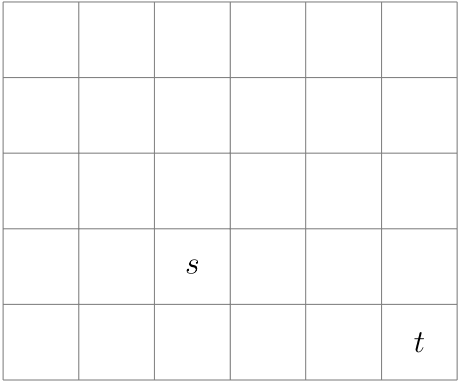
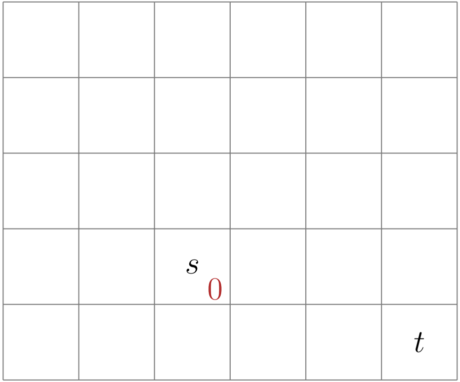
26.5 A*-Algorithm

Motivation A*



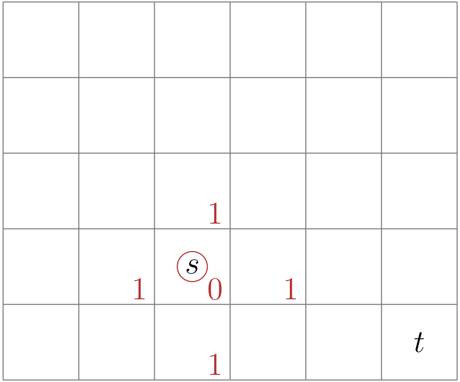
- Dijkstra Algorithm searches for all shortest paths, in all directions.

Motivation A*



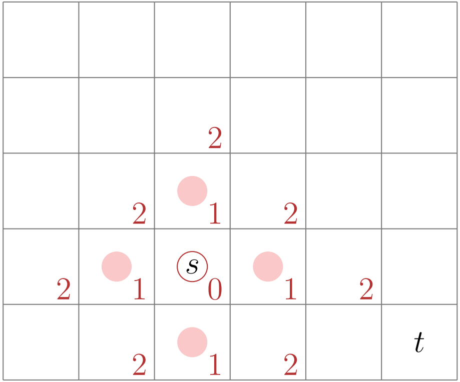
- Dijkstra Algorithm searches for all shortest paths, in all directions.

Motivation A*



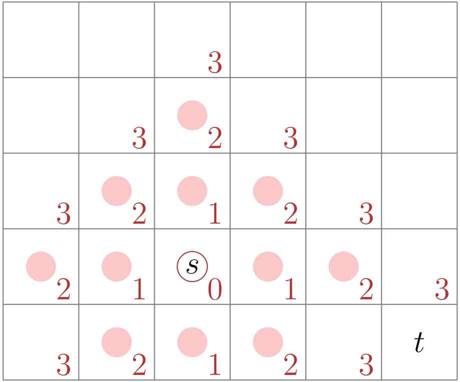
- Dijkstra Algorithm searches for all shortest paths, in all directions.

Motivation A*



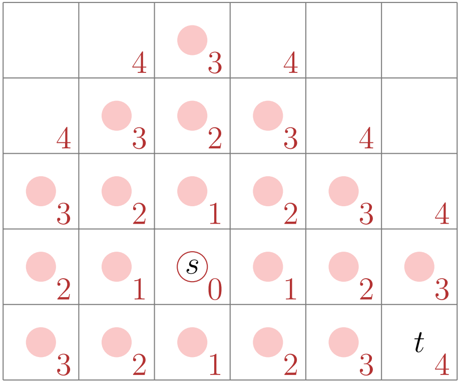
- Dijkstra Algorithm searches for all shortest paths, in all directions.

Motivation A*



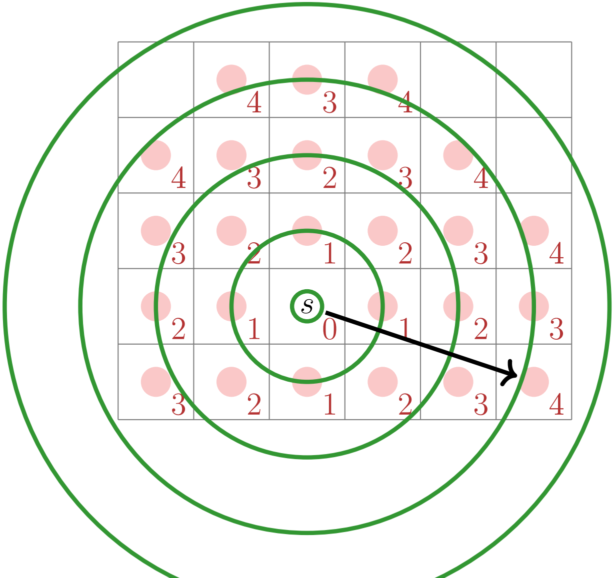
- Dijkstra Algorithm searches for all shortest paths, in all directions.

Motivation A*



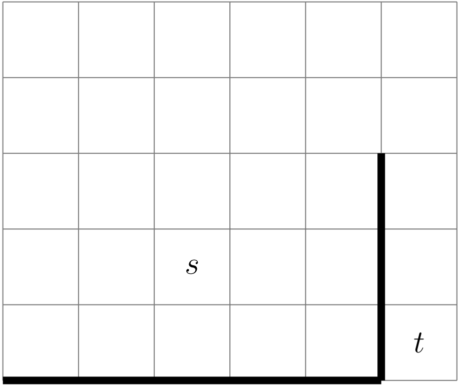
- Dijkstra Algorithm searches for all shortest paths, in all directions.

Motivation A*



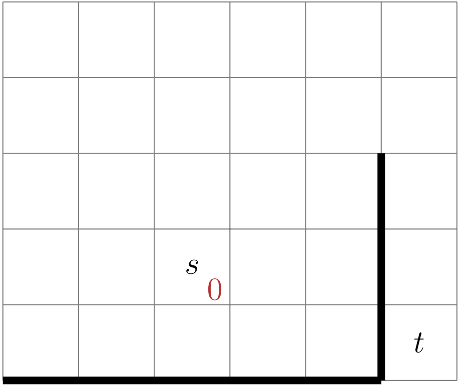
- Dijkstra Algorithm searches for all shortest paths, in all directions.

Motivation A*



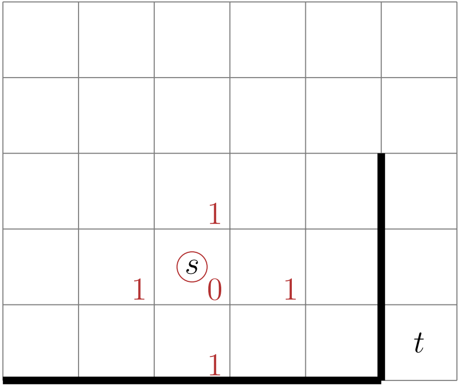
- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph’s structure.

Motivation A*



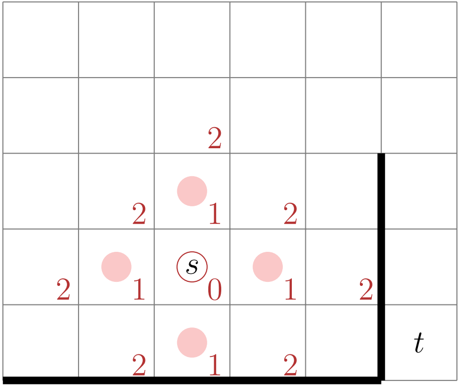
- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

Motivation A*



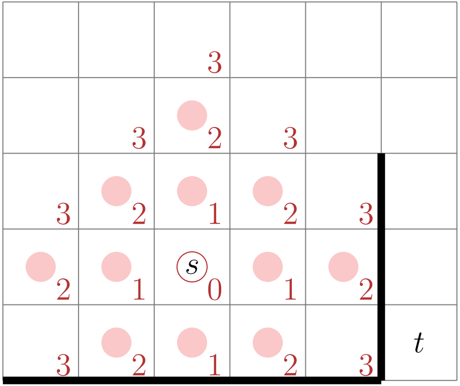
- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

Motivation A*



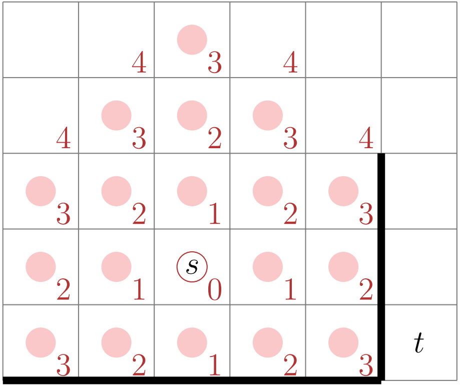
- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

Motivation A*



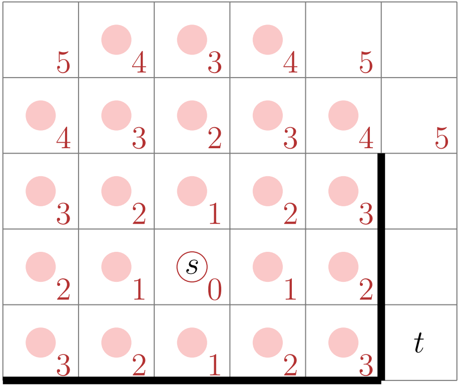
- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

Motivation A*



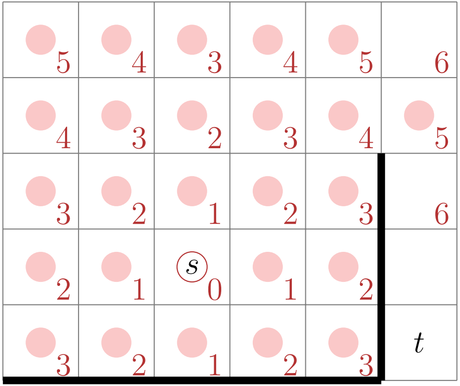
- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

Motivation A*



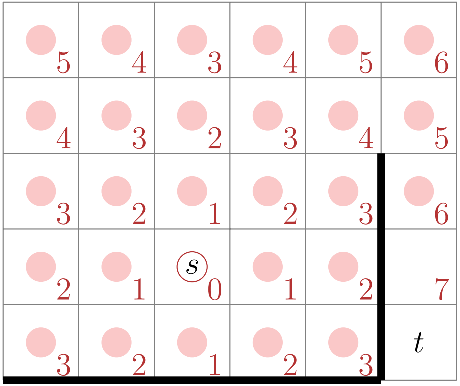
- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

Motivation A*



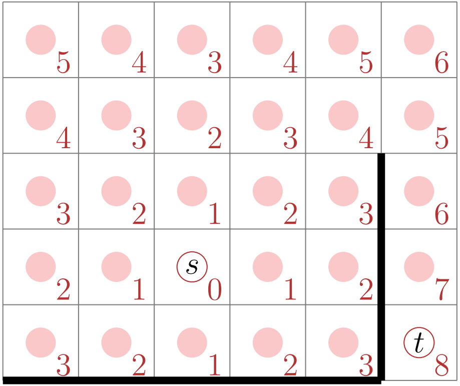
- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

Motivation A*



- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

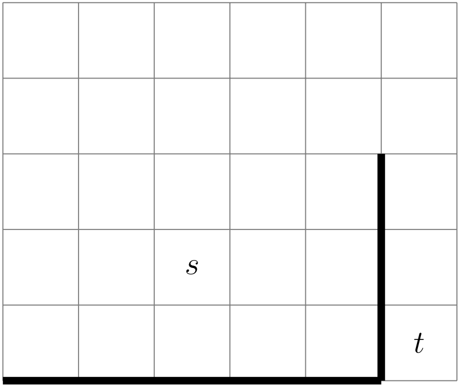
Motivation A*



- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

A* in Action

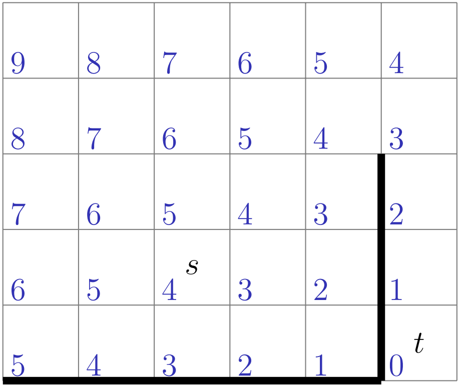
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distance)



- Idea: equip algorithm with a preferred direction by ways of a distance heuristic $\hat{h}$
- The value of this heuristics needs to underestimate the distance to t and is added to the found distance d_s to s

A* in Action

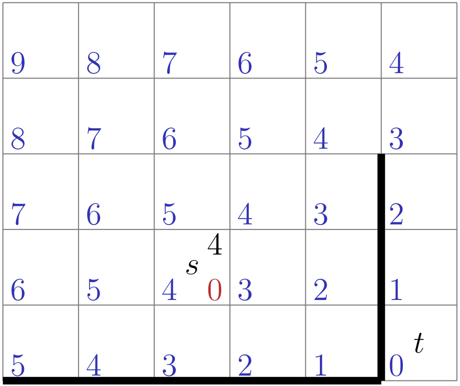
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ $\quad$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distance)



- Idea: equip algorithm with a preferred direction by ways of a distance heuristic $\hat{h}$
- The value of this heuristics needs to underestimate the distance to t and is added to the found distance d_s to s

A* in Action

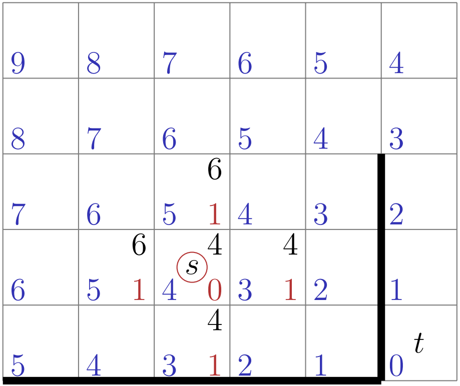
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distance)



- Idea: equip algorithm with a preferred direction by ways of a distance heuristic $\hat{h}$
- The value of this heuristics needs to underestimate the distance to t and is added to the found distance d_s to s

A* in Action

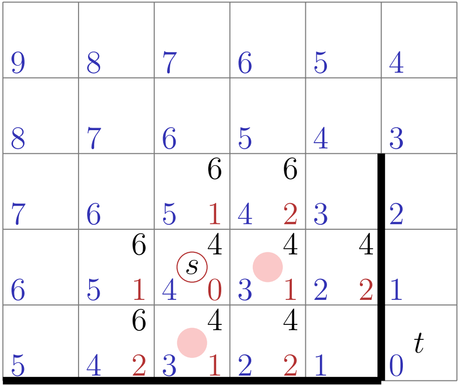
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distance)



- Idea: equip algorithm with a preferred direction by ways of a distance heuristic $\hat{h}$
- The value of this heuristics needs to underestimate the distance to t and is added to the found distance d_s to s

A* in Action

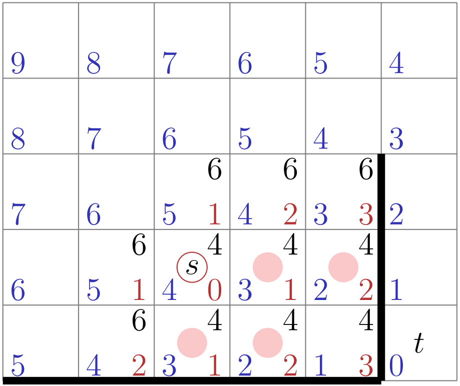
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distance)



- Idea: equip algorithm with a preferred direction by ways of a distance heuristic $\hat{h}$
- The value of this heuristics needs to underestimate the distance to t and is added to the found distance d_s to s

A* in Action

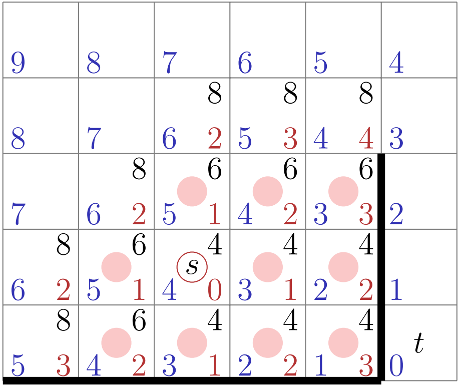
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ $(\hat{h} = \delta_x + \delta_y$ Manhattan-Distance)



- Idea: equip algorithm with a preferred direction by ways of a distance heuristic $\hat{h}$
- The value of this heuristics needs to underestimate the distance to t and is added to the found distance d_s to s

A* in Action

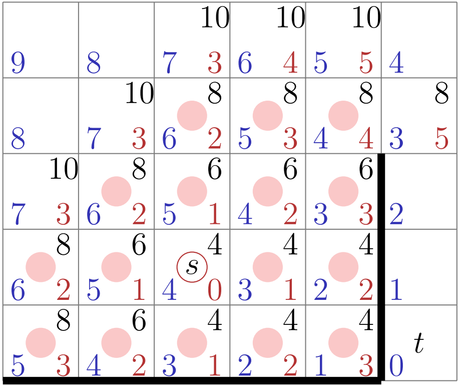
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distance)



- Idea: equip algorithm with a preferred direction by ways of a distance heuristic $\hat{h}$
- The value of this heuristics needs to underestimate the distance to t and is added to the found distance d_s to s

A* in Action

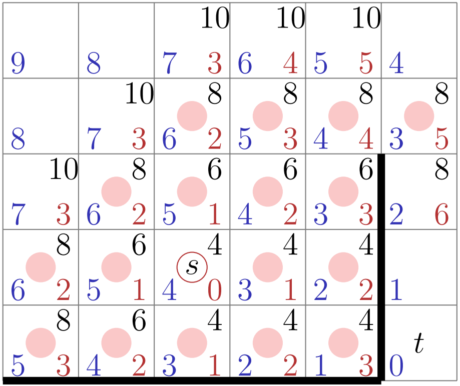
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distance)



- Idea: equip algorithm with a preferred direction by ways of a distance heuristic $\hat{h}$
- The value of this heuristics needs to underestimate the distance to t and is added to the found distance d_s to s

A* in Action

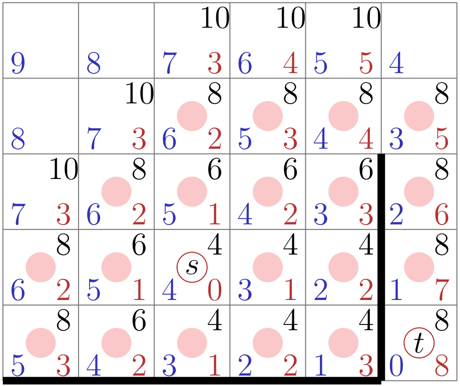
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ $(\hat{h} = \delta_x + \delta_y$ Manhattan-Distance)



- Idea: equip algorithm with a preferred direction by ways of a distance heuristic $\hat{h}$
- The value of this heuristics needs to underestimate the distance to t and is added to the found distance d_s to s

A* in Action

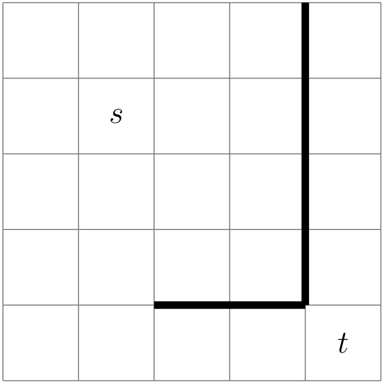
$\hat{f}(u) = d_s[u] + \hat{h}(u)$ $\quad$ ($\hat{h} = \delta_x + \delta_y$ Manhattan-Distance)



- Idea: equip algorithm with a preferred direction by ways of a distance heuristic $\hat{h}$
- The value of this heuristics needs to underestimate the distance to t and is added to the found distance d_s to s

Keep backward path

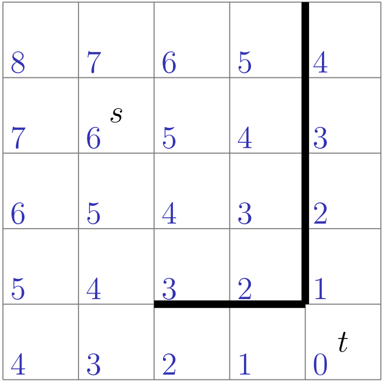
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x + \delta_y)$$



- The algorithm works like the Dijkstra-algorithm
- For finding the next candidate of R instead of the value d_s the value of $\hat{f} = \hat{h} + d_s$ is used

Keep backward path

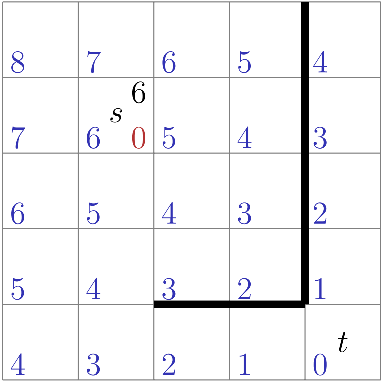
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x + \delta_y)$$



- The algorithm works like the Dijkstra-algorithm
- For finding the next candidate of R instead of the value d_s the value of $\hat{f} = \hat{h} + d_s$ is used

Keep backward path

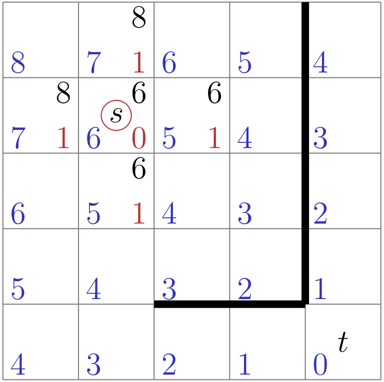
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x + \delta_y)$$



- The algorithm works like the Dijkstra-algorithm
- For finding the next candidate of R instead of the value d_s the value of $\hat{f} = \hat{h} + d_s$ is used

Keep backward path

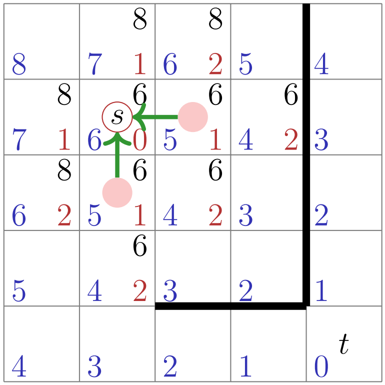
$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x + \delta_y)$



- The algorithm works like the Dijkstra-algorithm
- For finding the next candidate of R instead of the value d_s the value of $\hat{f} = \hat{h} + d_s$ is used

Keep backward path

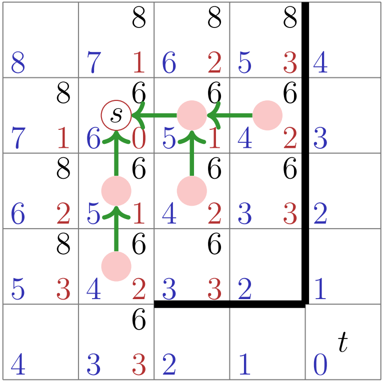
$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x + \delta_y)$



- The algorithm works like the Dijkstra-algorithm
- For finding the next candidate of R instead of the value d_s the value of $\hat{f} = \hat{h} + d_s$ is used

Keep backward path

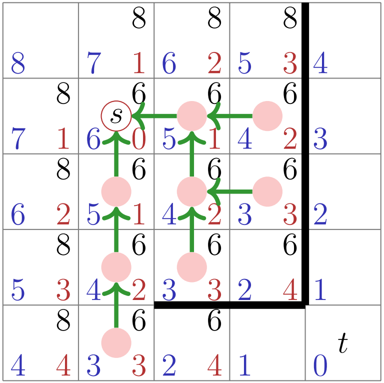
$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x + \delta_y)$



- The algorithm works like the Dijkstra-algorithm
- For finding the next candidate of R instead of the value d_s the value of $\hat{f} = \hat{h} + d_s$ is used

Keep backward path

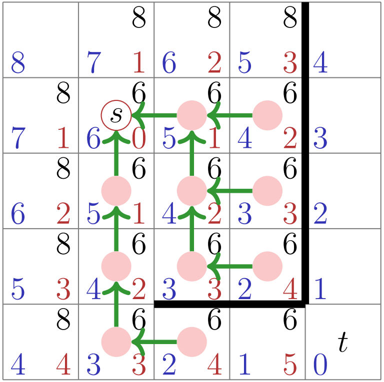
$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x + \delta_y)$



- The algorithm works like the Dijkstra-algorithm
- For finding the next candidate of R instead of the value d_s the value of $\hat{f} = \hat{h} + d_s$ is used

Keep backward path

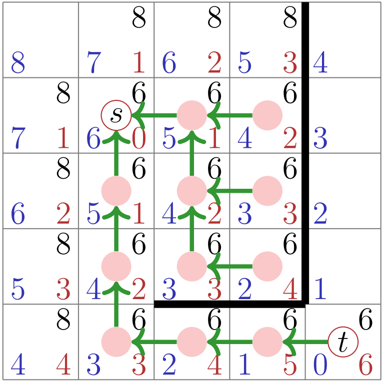
$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x + \delta_y)$



- The algorithm works like the Dijkstra-algorithm
- For finding the next candidate of R instead of the value d_s the value of $\hat{f} = \hat{h} + d_s$ is used

Keep backward path

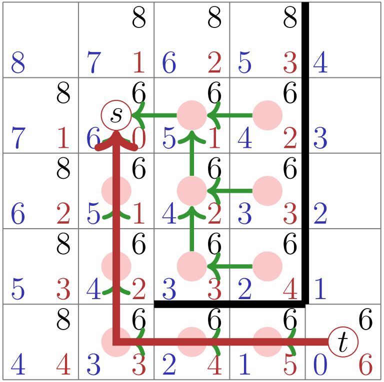
$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x + \delta_y)$



- The algorithm works like the Dijkstra-algorithm
- For finding the next candidate of R instead of the value d_s the value of $\hat{f} = \hat{h} + d_s$ is used

Keep backward path

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



- The algorithm works like the Dijkstra-algorithm
- For finding the next candidate of R instead of the value d_s the value of $\hat{f} = \hat{h} + d_s$ is used

A*-Algorithm

Prerequisites

- Positively weighted, finite graph $G = (V, E, c)$
- $s \in V$, $t \in V$
- Distance estimate $\widehat{h}_t(v) \leq h_t(v) := \delta(v, t)\ \forall\, v \in V$.
- Wanted: shortest path $p : s \rightsquigarrow t$

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

Input: Positively weighted Graph $G = (V, E, c)$, starting point $s \in V$, end point $t \in V$, estimate $\widehat{h}(v) \leq \delta(v, t)$

Output: Existence and value of a shortest path from s to t

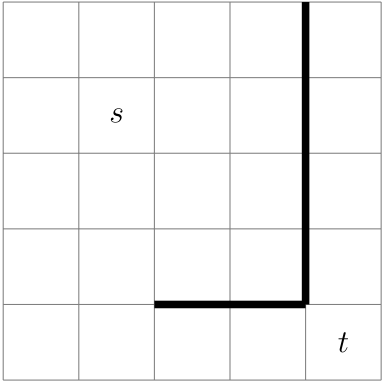
foreach $u \in V$ **do**
  $d[u] \leftarrow \infty$; $\widehat{f}[u] \leftarrow \infty$; $\pi[u] \leftarrow$ null

$d[s] \leftarrow 0$; $\widehat{f}[s] \leftarrow \widehat{h}(s)$; $N \leftarrow \{s\}$; $K \leftarrow \{\}$
while $N \neq \emptyset$ **do**
  $u \leftarrow$ 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

What if $\hat{h}$ does not underestimate

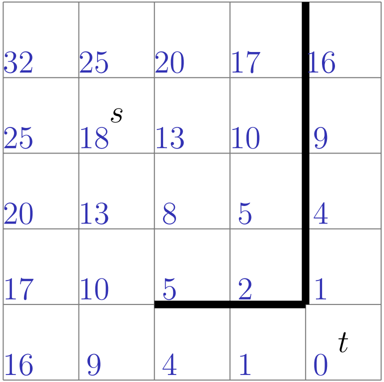
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- Algorithm can terminate with the wrong result when $\hat{h}$ does not under-estimate the distance to t.
- although the heuristics looks reasonable otherwise (it is monotonic, for instance)

What if $\hat{h}$ does not underestimate

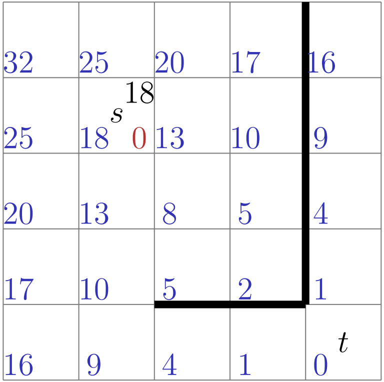
$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x^2 + \delta_y^2)$



- Algorithm can terminate with the wrong result when $\hat{h}$ does not under-estimate the distance to t.
- although the heuristics looks reasonable otherwise (it is monotonic, for instance)

What if $\hat{h}$ does not underestimate

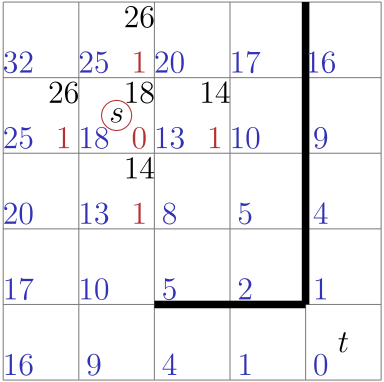
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- Algorithm can terminate with the wrong result when $\hat{h}$ does not under-estimate the distance to t.
- although the heuristics looks reasonable otherwise (it is monotonic, for instance)

What if $\hat{h}$ does not underestimate

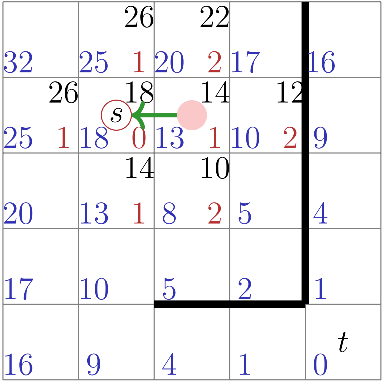
$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x^2 + \delta_y^2)$



- Algorithm can terminate with the wrong result when $\hat{h}$ does not under-estimate the distance to t.
- although the heuristics looks reasonable otherwise (it is monotonic, for instance)

What if $\hat{h}$ does not underestimate

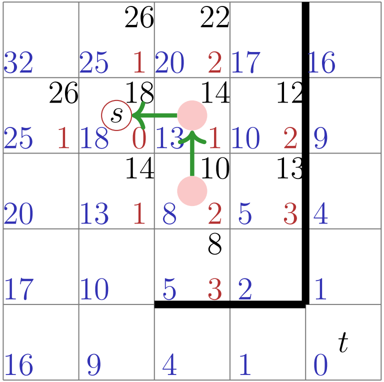
$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x^2 + \delta_y^2)$



- Algorithm can terminate with the wrong result when $\hat{h}$ does not under-estimate the distance to t.
- although the heuristics looks reasonable otherwise (it is monotonic, for instance)

What if $\hat{h}$ does not underestimate

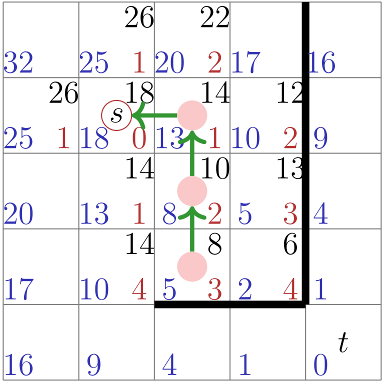
$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x^2 + \delta_y^2)$



- Algorithm can terminate with the wrong result when $\hat{h}$ does not under-estimate the distance to t.
- although the heuristics looks reasonable otherwise (it is monotonic, for instance)

What if $\hat{h}$ does not underestimate

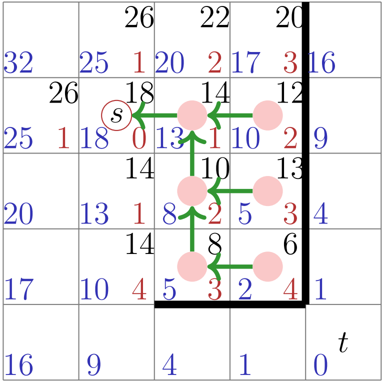
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- Algorithm can terminate with the wrong result when $\hat{h}$ does not under-estimate the distance to t.
- although the heuristics looks reasonable otherwise (it is monotonic, for instance)

What if $\hat{h}$ does not underestimate

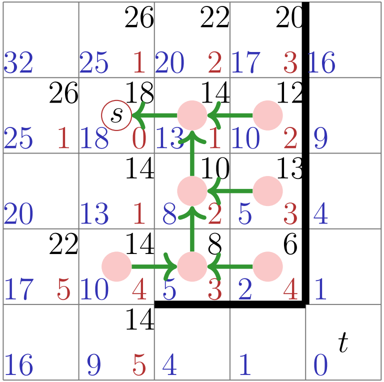
$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x^2 + \delta_y^2)$



- Algorithm can terminate with the wrong result when $\hat{h}$ does not under-estimate the distance to t.
- although the heuristics looks reasonable otherwise (it is monotonic, for instance)

What if $\hat{h}$ does not underestimate

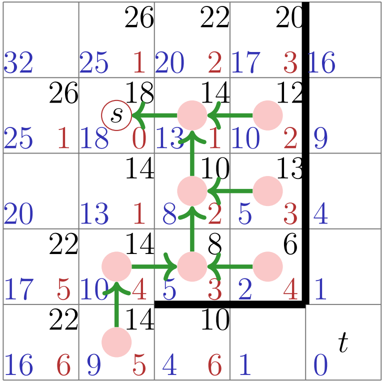
$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x^2 + \delta_y^2)$



- Algorithm can terminate with the wrong result when $\hat{h}$ does not under-estimate the distance to t.
- although the heuristics looks reasonable otherwise (it is monotonic, for instance)

What if $\hat{h}$ does not underestimate

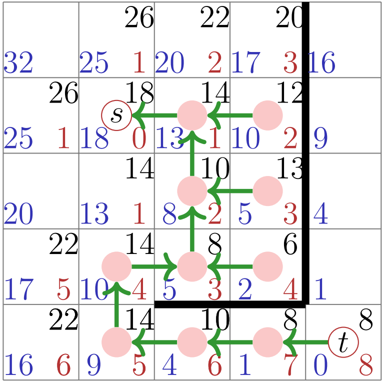
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- Algorithm can terminate with the wrong result when $\hat{h}$ does not under-estimate the distance to t.
- although the heuristics looks reasonable otherwise (it is monotonic, for instance)

What if $\hat{h}$ does not underestimate

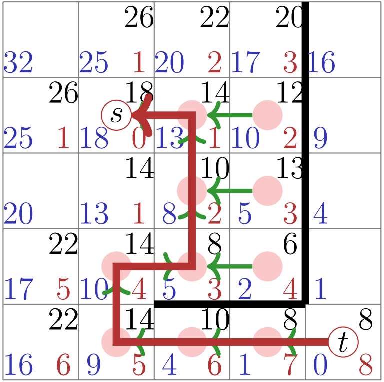
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \qquad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- Algorithm can terminate with the wrong result when $\hat{h}$ does not under-estimate the distance to t.
- although the heuristics looks reasonable otherwise (it is monotonic, for instance)

What if $\hat{h}$ does not underestimate

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



- Algorithm can terminate with the wrong result when $\hat{h}$ does not under-estimate the distance to t.
- although the heuristics looks reasonable otherwise (it is monotonic, for instance)

Revisiting nodes

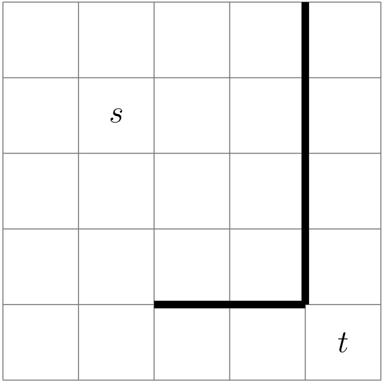
- The A*-algorithm can re-insert nodes that had been extracted from R before.
- This can lead to suboptimal behavior (w.r.t. running time of the algorithm).
- If $\widehat{h}$, in addition to being admissible ($\widehat{h}(v) \leq h(v)$ for all $v \in V$), fulfils monotonicity, i.e. if for all $(u, u') \in E$:

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

  then the A*-Algorithm is equivalent to the Dijsktra-algorithm with edge weights $\tilde{c}(u, v) = c(u, v) + \widehat{h}(u) - \widehat{h}(v)$, and no node is re-inserted into R.
- It is not always possible to find monotone heuristics.

A crazy $\hat{h}$

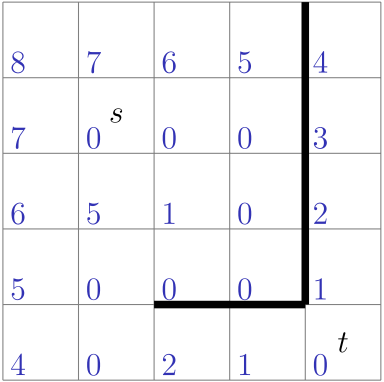
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithm terminates correctly even if the distance heuristic is not monotonic
- It is then possible that nodes are removed and re-inserted into R multiple times.

A crazy $\hat{h}$

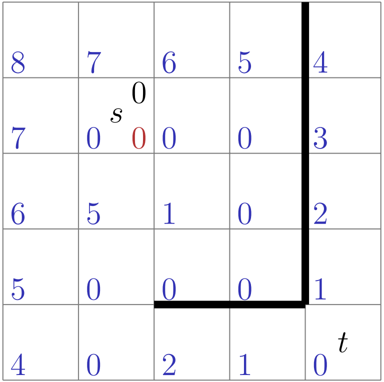
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithm terminates correctly even if the distance heuristic is not monotonic
- It is then possible that nodes are removed and re-inserted into R multiple times.

A crazy $\hat{h}$

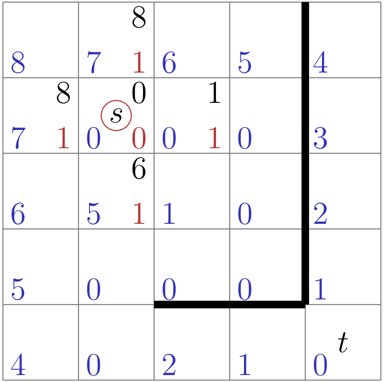
$\hat{f}(u) = d_s[u] + \hat{h}(u)$



- Algorithm terminates correctly even if the distance heuristic is not monotonic
- It is then possible that nodes are removed and re-inserted into R multiple times.

A crazy $\hat{h}$

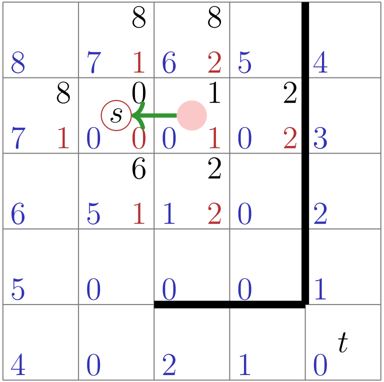
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithm terminates correctly even if the distance heuristic is not monotonic
- It is then possible that nodes are removed and re-inserted into R multiple times.

A crazy $\hat{h}$

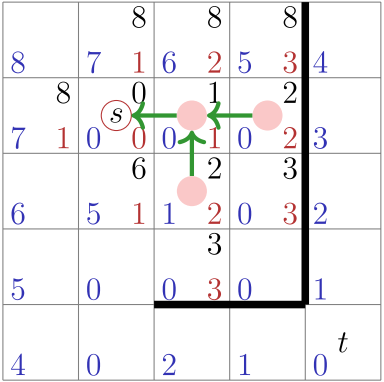
$\hat{f}(u) = d_s[u] + \hat{h}(u)$



- Algorithm terminates correctly even if the distance heuristic is not monotonic
- It is then possible that nodes are removed and re-inserted into R multiple times.

A crazy $\hat{h}$

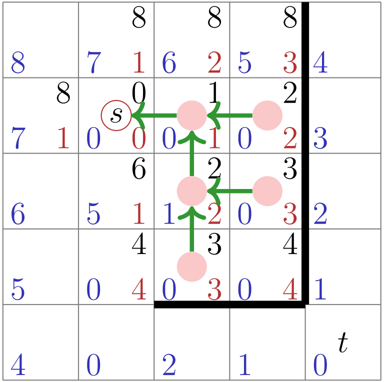
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithm terminates correctly even if the distance heuristic is not monotonic
- It is then possible that nodes are removed and re-inserted into R multiple times.

A crazy $\hat{h}$

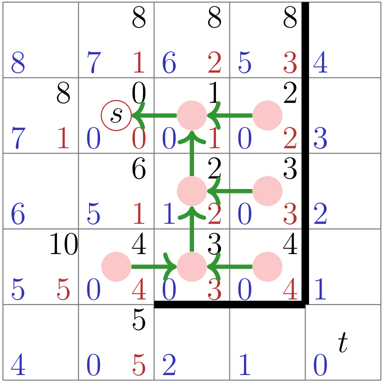
$\hat{f}(u) = d_s[u] + \hat{h}(u)$



- Algorithm terminates correctly even if the distance heuristic is not monotonic
- It is then possible that nodes are removed and re-inserted into R multiple times.

A crazy $\hat{h}$

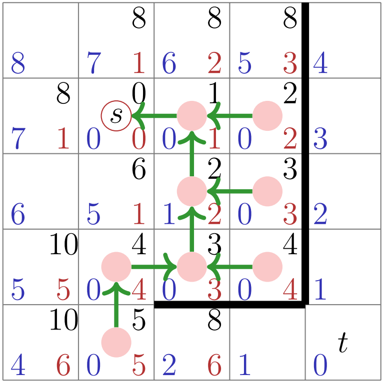
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithm terminates correctly even if the distance heuristic is not monotonic
- It is then possible that nodes are removed and re-inserted into R multiple times.

A crazy $\hat{h}$

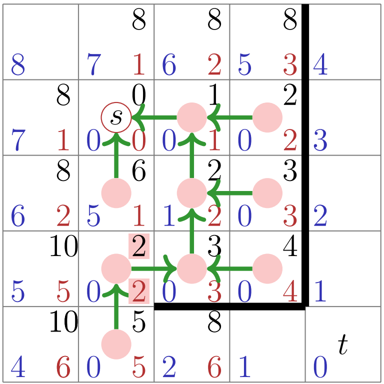
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithm terminates correctly even if the distance heuristic is not monotonic
- It is then possible that nodes are removed and re-inserted into R multiple times.

A crazy $\hat{h}$

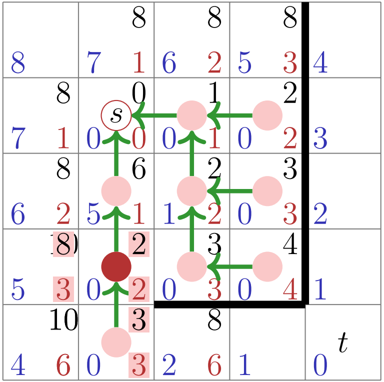
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithm terminates correctly even if the distance heuristic is not monotonic
- It is then possible that nodes are removed and re-inserted into R multiple times.

A crazy $\hat{h}$

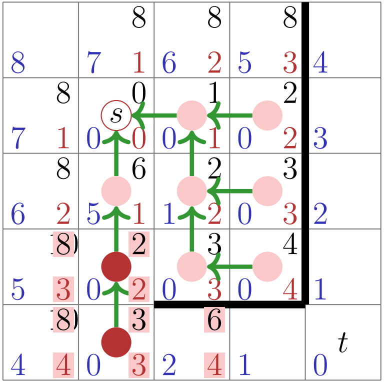
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithm terminates correctly even if the distance heuristic is not monotonic
- It is then possible that nodes are removed and re-inserted into R multiple times.

A crazy $\hat{h}$

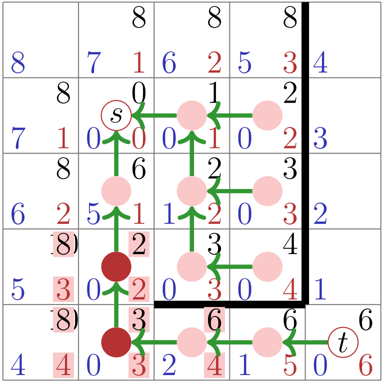
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- Algorithm terminates correctly even if the distance heuristic is not monotonic
- It is then possible that nodes are removed and re-inserted into R multiple times.

A crazy $\hat{h}$

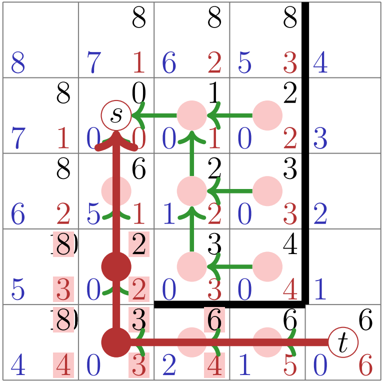
$\hat{f}(u) = d_s[u] + \hat{h}(u)$



- Algorithm terminates correctly even if the distance heuristic is not monotonic
- It is then possible that nodes are removed and re-inserted into R multiple times.

A crazy $\hat{h}$

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



- Algorithm terminates correctly even if the distance heuristic is not monotonic
- It is then possible that nodes are removed and re-inserted into R multiple times.

Conclusion

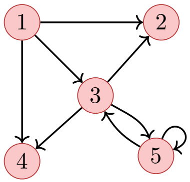
- The A*-Algorithm is an extension of the Dijkstra algortihm by a distance heuristic $\hat{h}$.
- A* = Dijkstra if $\hat{h} \equiv 0$
- If $\hat{h}$ underestimates the real distance, the algorithm works correctly.
- If $\hat{h}$ is monotone in addition, then the algorithm works efficiently.
- In practical applications (e.g. routing), the choice of $\hat{h}$ is often intuitive and leads to a significant improvement over Dijkstra.
- Correctness proof in the handout

27. Transitive Closure, All Pairs Shortest Paths

Reflexive transitive closure [Ottman/Widmayer, Kap. 9.2 Cormen et al, Kap. 25.2] Floyd-Warshall Algorithm [Ottman/Widmayer, Kap. 9.5.3 Cormen et al, Kap. 25.2]

Adjacency Matrix Product

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



Interpretation

Theorem 27

Let $G = (V, E)$ *be a graph and* $k \in \mathbb{N}$*. Then the element* $a_{i,j}^{(k)}$ *of the matrix* $(a_{i,j}^{(k)})_{1 \leq i,j \leq n} = (A_G)^k$ *provides the number of paths with length* k *from* v_i *to* v_j *.*

Graphs and Relations

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

Graphs and Relations

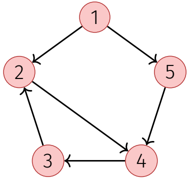
Graph $G = (V, E)$
adjacencies $A_G \mathrel{\widehat{=}}$ Relation $E \subseteq V \times V$ over V

- **reflexive** $\Leftrightarrow a_{i,i} = 1$ for all $i = 1, \ldots, n$. (loops)
- **symmetric** $\Leftrightarrow a_{i,j} = a_{j,i}$ for all $i, j = 1, \ldots, n$ (undirected)
- **transitive** $\Leftrightarrow (u, v) \in E$, $(v, w) \in E \Rightarrow (u, w) \in E$. (reachability)

Reflexive Transitive Closure

Reflexive transitive closure of G $\Leftrightarrow$ **Reachability relation** E^*: $(v, w) \in E^*$ iff $\exists$ path from node v to w.

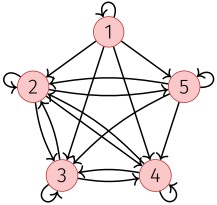
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$



$G = (V, E)$

$\Rightarrow$

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



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

Algorithm $A \cdot A$

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

```
B ← 0
for r ← 1 to n do
    for c ← 1 to n do
        for k ← 1 to n do
            b_rc ← b_rc + a_rk · a_kc          // Number of Paths
return B
```

Counts number of paths of length 2

Algorithm $A \otimes A$

Input: Adjacency-Matrix $A = (a_{ij})_{i,j=1\ldots n}$
Output: Modified Matrix Product $B = (b_{ij})_{i,j=1\ldots n} = A \otimes A$

```
B ← A                                          // Keep paths
for r ← 1 to n do
    for c ← 1 to n do
        for k ← 1 to n do
            b_rc ← max{b_rc, a_rk · a_kc}      // Path: yes/no
return B
```

Computes which paths of length 1 and 2 exist

Computation of the Reflexive Transitive Closure

Goal: computation of $B = (b_{ij})_{1 \leq i,j \leq n}$ with $b_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E^*$

Computation of the Reflexive Transitive Closure

Goal: computation of $B = (b_{ij})_{1 \leq i,j \leq n}$ with $b_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E^*$ First idea:

- Start with $B \leftarrow A$ and set $b_{ii} = 1$ for each i (Reflexivity.).
- Compute

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

  how

Computation of the Reflexive Transitive Closure

Goal: computation of $B = (b_{ij})_{1 \le i,j \le n}$ with $b_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E^*$ First idea:

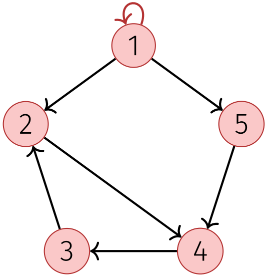
- Start with $B \leftarrow A$ and set $b_{ii} = 1$ for each i (Reflexivity.).
- Compute

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

with powers of 2 $B_2 := B \otimes B$, $B_4 := B_2 \otimes B_2$, $B_8 = B_4 \otimes B_4$ ...
$\Rightarrow$ running time $n^3 \lceil \log_2 n \rceil$

Improvement: Algorithm of Warshall (1962)

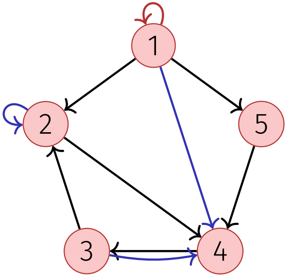
Inductive procedure: all paths known over nodes from $\{v_i : i < k\}$. Add node v_k.



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

Improvement: Algorithm of Warshall (1962)

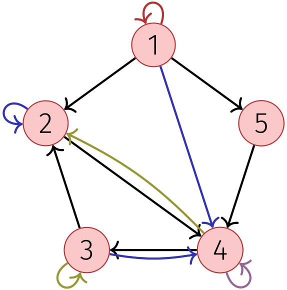
Inductive procedure: all paths known over nodes from $\{v_i : i < k\}$. Add node v_k.



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

Improvement: Algorithm of Warshall (1962)

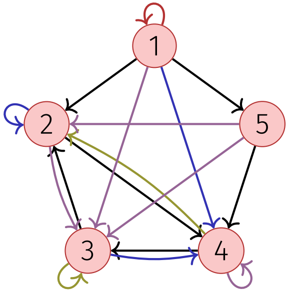
Inductive procedure: all paths known over nodes from $\{v_i : i < k\}$. Add node v_k.



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

Improvement: Algorithm of Warshall (1962)

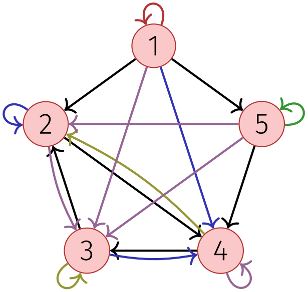
Inductive procedure: all paths known over nodes from $\{v_i : i < k\}$. Add node v_k.



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

Improvement: Algorithm of Warshall (1962)

Inductive procedure: all paths known over nodes from $\{v_i : i < k\}$. Add node v_k.



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

Algorithm TransitiveClosure(A_G)

Input: Adjacency matrix $A_G = (a_{ij})_{i,j=1\dots n}$
Output: Reflexive transitive closure $B = (b_{ij})_{i,j=1\dots n}$ of G

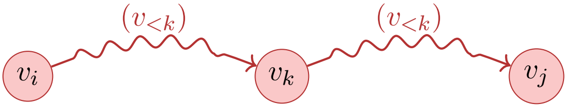
```
B ← A_G
for k ← 1 to n do
    b_kk ← 1                                   // Reflexivity
    for r ← 1 to n do
        for c ← 1 to n do
            b_rc ← max{b_rc, b_rk · b_kc}      // All paths via v_k
return B
```

Runtime $\Theta(n^3)$.

Correctness of the Algorithm (Induction)

Invariant (k): all paths via nodes with maximal index $< k$ considered.

- **Base case ($k = 1$)**: All directed paths (all edges) in A_G considered.
- **Hypothesis**: invariant (k) fulfilled.
- **Step** ($k \to k+1$): For each path from v_i to v_j via nodes with maximal index k: by the hypothesis $b_{ik} = 1$ and $b_{kj} = 1$. Therefore in the k-th iteration: $b_{ij} \leftarrow 1$.



All shortest Paths

Compute the weight of a shortest path for each pair of nodes.

- $|V|\times$ Application of Dijkstra's Shortest Path algorithm $\mathcal{O}(|V|\cdot(|E|+|V|)\cdot\log|V|)$ (with Fibonacci Heap: $\mathcal{O}(|V|^2\log|V|+|V|\cdot|E|)$)
- $|V|\times$ Application of Bellman-Ford: $\mathcal{O}(|E|\cdot|V|^2)$
- There are better ways!

Induction via node number

Consider weights of all shortest paths S^k with intermediate nodes in[42] $V^k := \{v_1, \dots, v_k\}$, provided that weights for all shortest paths S^{k-1} with intermediate nodes in V^{k-1} are given.

- v_k no intermediate node of a shortest path of $v_i \rightsquigarrow v_j$ in V^k: Weight of a shortest path $v_i \rightsquigarrow v_j$ in S^{k-1} is then also weight of shortest path in S^k.
- v_k intermediate node of a shortest path $v_i \rightsquigarrow v_j$ in V^k: Sub-paths $v_i \rightsquigarrow v_k$ and $v_k \rightsquigarrow v_j$ contain intermediate nodes only from S^{k-1}.

[42] like for the algorithm of the reflexive transitive closure of Warshall

Induction via node number

$d^k(u,v)$ = Minimal weight of a path $u \rightsquigarrow v$ with intermediate nodes in V^k

Induktion

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

Algorithm Floyd-Warshall(G)

Input: Graph $G = (V, E, c)$ without negative weight cycles.
Output: Minimal weights of all paths 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**
            $d^k(v_i, v_j) = \min\{d^{k-1}(v_i, v_j), d^{k-1}(v_i, v_k) + d^{k-1}(v_k, v_j)\}$

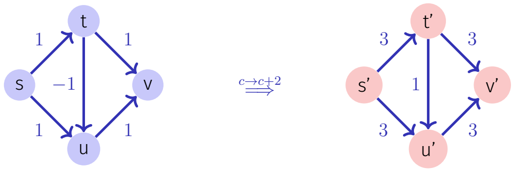
Runtime: $\Theta(|V|^3)$

Remark: Algorithm can be executed with a single matrix d (in place).

Reweighting

Idea: Reweighting the graph in order to apply Dijkstra's algorithm.

The following does **not** work. The graphs are not equivalent in terms of shortest paths.



Reweighting

Other Idea: “Potential” (Height) on the nodes

- $G = (V, E, c)$ a weighted graph.
- Mapping $h : V \to \mathbb{R}$
- New weights

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

Reweighting

Observation: A path p is shortest path in in $G = (V, E, c)$ iff it is shortest path in in $\tilde{G} = (V, E, \tilde{c})$

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

Thus $\tilde{c}(p)$ minimal in all $v_0 \rightsquigarrow v_k \iff c(p)$ minimal in all $v_0 \rightsquigarrow v_k$.

Weights of cycles are invariant: $\tilde{c}(v_0, \ldots, v_k = v_0) = c(v_0, \ldots, v_k = v_0)$

Johnson's Algorithm

Add a new node $s \notin V$:

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

Johnson's Algorithm

If no negative cycles, choose as height function the weight of the shortest paths from s,

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

For a minimal weight d of a path the following triangular inequality holds:

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

Substitution yields $h(v) \leq h(u) + c(u, v)$. Therefore

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

Algorithm Johnson(G)

Input: Weighted Graph $G = (V, E, c)$
Output: Minimal weights of all paths D.

New node s. Compute $G' = (V', E', c')$
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**
    $D(u, v) \leftarrow \tilde{d}(u, v) + h(v) - h(u)$

Analysis

Runtimes

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

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