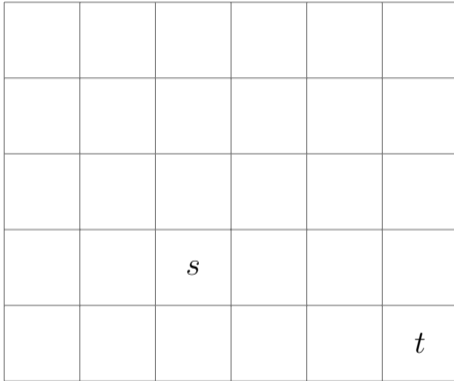


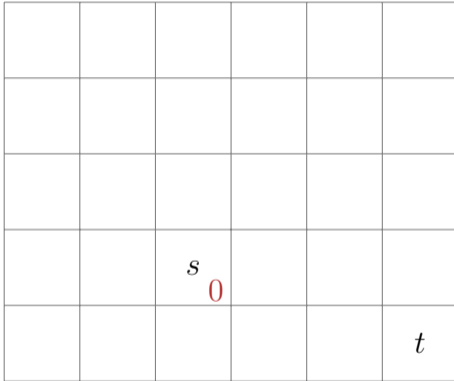
26.5 A*-Algorithm

Motivation A*



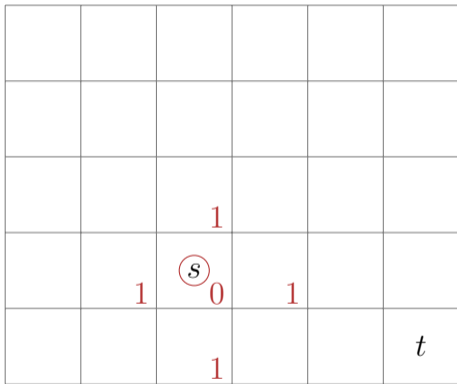
- Dijkstra Algorithm searches for all shortest paths, in all directions.

Motivation A*



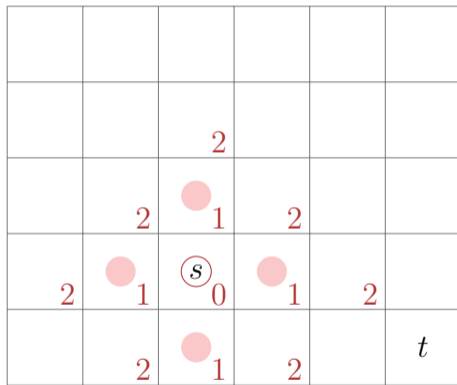
- Dijkstra Algorithm searches for all shortest paths, in all directions.

Motivation A*



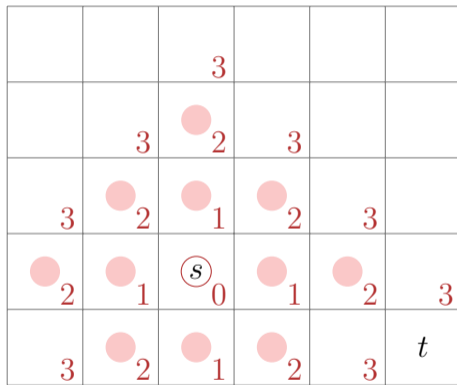
- Dijkstra Algorithm searches for all shortest paths, in all directions.

Motivation A*



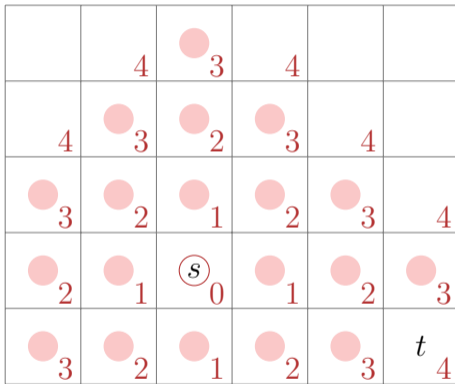
- Dijkstra Algorithm searches for all shortest paths, in all directions.

Motivation A*



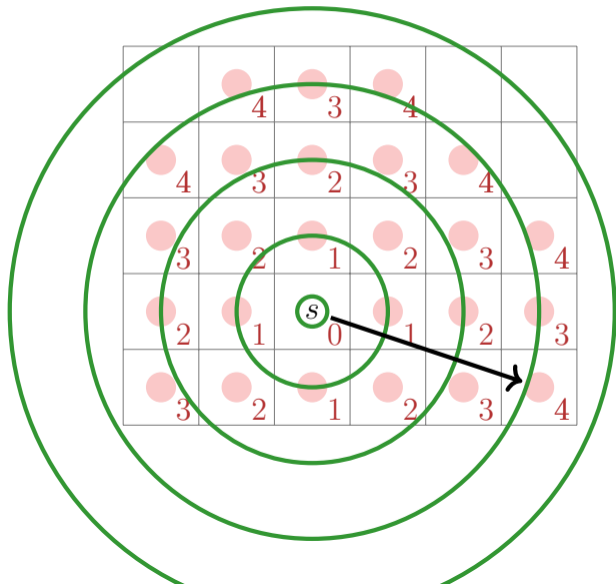
- Dijkstra Algorithm searches for all shortest paths, in all directions.

Motivation A*



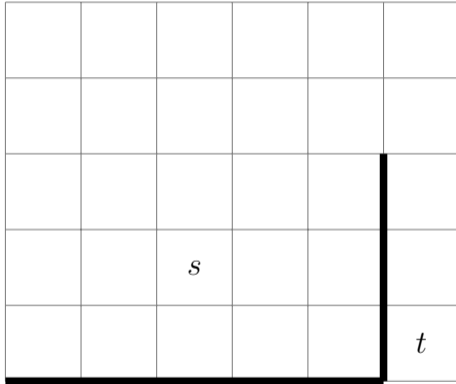
- Dijkstra Algorithm searches for all shortest paths, in all directions.

Motivation A*



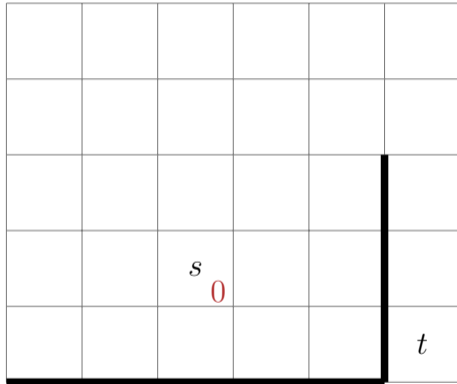
- Dijkstra Algorithm searches for all shortest paths, in all directions.

Motivation A*



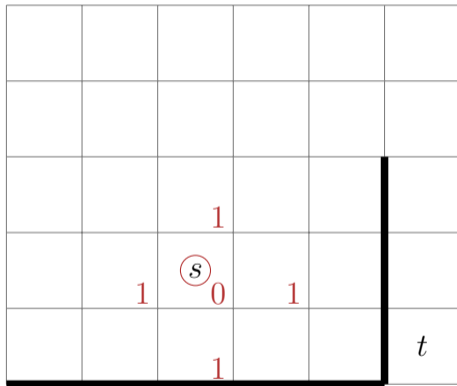
- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

Motivation A*



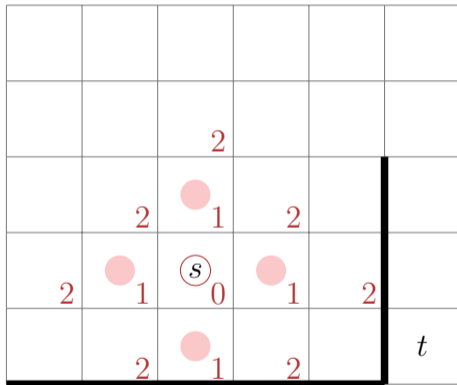
- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

Motivation A*



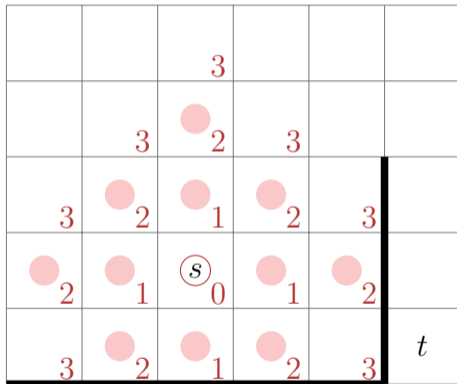
- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

Motivation A*



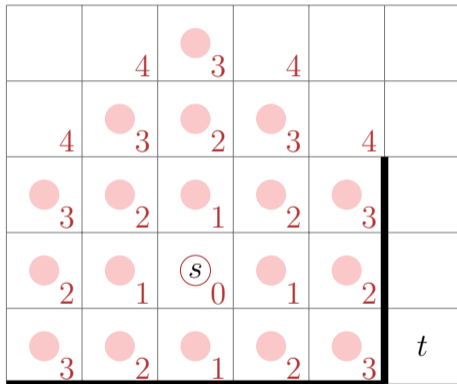
- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

Motivation A*



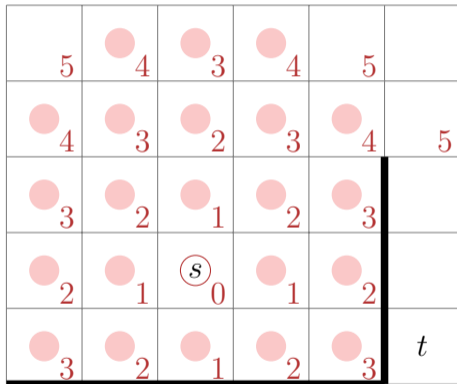
- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

Motivation A*



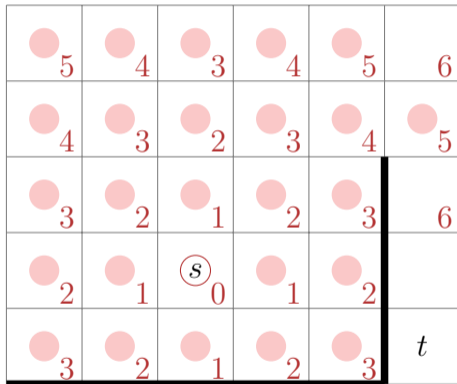
- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

Motivation A*



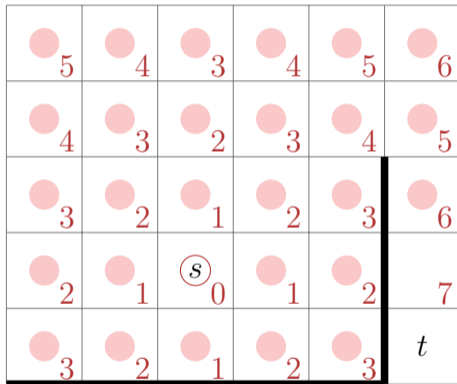
- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

Motivation A*



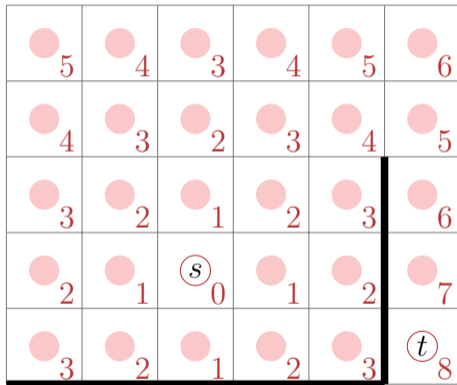
- Dijkstra Algorithm searches for all shortest paths, in all directions.
- which is correct, because the algorithm does not know about the graph's structure.

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

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



- 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

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

9	8	7	6	5	4
8	7	6	5	4	3
7	6	5	4	3	2
6	5	4 ^s	3	2	1
5	4	3	2	1	0 ^t

- 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

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

9	8	7	6	5	4
8	7	6	5	4	3
7	6	5	4	3	2
6	5	4 ^s	0	3	2
5	4	3	2	1	0 ^t

- 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

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

9	8	7	6	5	4		
8	7	6	5	4	3		
		6					
7	6	5	1	4	3	2	
	6	4	4				
6	5	1	4	0	3	1	2
		4					
5	4	3	1	2	1		0

s is located at the cell with value 0. *t* is located at the cell with value 0 in the bottom right corner.

- 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

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

9	8	7	6	5	4					
8	7	6	5	4	3					
		6	6							
7	6	5	1	4	2	3	2			
	6	4	4	4						
6	5	1	4	0	3	1	2	2	1	
	6	4	4							
5	4	2	3	1	2	2	1		0	<i>t</i>

- 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

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

9	8	7	6	5	4					
8	7	6	5	4	3					
		6	6	6						
7	6	5	1	4	2	3	2			
	6	4	4	4						
6	5	1	4	0	3	1	2	2	1	
	6	4	4	4						
5	4	2	3	1	2	2	1	3	0	t

- 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

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

9	8	7	6	5	4					
		8	8	8						
8	7	6	2	5	3	4	4	3		
	8	6	6	6	6					
7	6	2	5	1	4	2	3	3	2	
	8	6	4	4	4					
6	2	5	1	4	0	3	1	2	2	1
	8	6	4	4	4					
5	3	4	2	3	1	2	2	1	3	0

- 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

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

			10	10	10				
9	8	7	3	6	4	5	5	4	
		10	8	8	8	8	8	8	8
8	7	3	6	2	5	3	4	4	3
	10	8	6	6	6	6	6	6	
7	3	6	2	5	1	4	2	3	3
	8	6	4	4	4	4	4	4	4
6	2	5	1	4	0	3	1	2	2
	8	6	4	4	4	4	4	4	4
5	3	4	2	3	1	2	2	1	3
									0

- 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

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

			10	10	10			
9	8	7	3	6	4	5	5	4
	10	8	8	8	8	8	8	8
8	7	3	6	2	5	3	4	4
	10	8	6	6	6	6	6	8
7	3	6	2	5	1	4	2	3
	8	6	4	4	4	4	4	
6	2	5	1	4	0	3	1	2
	8	6	4	4	4	4	4	
5	3	4	2	3	1	2	2	1
								0

The table shows a grid of values representing the estimated cost function $\hat{f}(u)$. The start node s is at the cell with value 0 (row 6, column 5). The target node t is at the cell with value 0 (row 8, column 9). Red circles are placed on several cells, and a thick black line highlights the path from s to t .

- 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

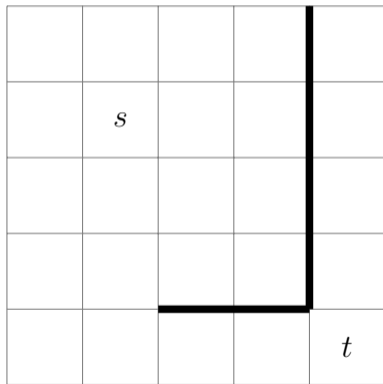
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x + \delta_y \text{ Manhattan-Distance})$$

			10	10	10				
9	8	7	3	6	4	5	5	4	
		10	8	8	8	8	8	8	
8	7	3	6	2	5	3	4	4	3
	10	8	6	6	6	6	6	6	8
7	3	6	2	5	1	4	2	3	3
	8	6	4	4	4	4	4	4	8
6	2	5	1	4	0	3	1	2	2
	8	6	4	4	4	4	4	4	8
5	3	4	2	3	1	2	2	1	3
									0
									8

- 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

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



- 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

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

8	7	6	5	4
7	6 ^s	5	4	3
6	5	4	3	2
5	4	3	2	1
4	3	2	1	0 ^t

- 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

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

8	7	6	5	4
7	6 ^s	5	4	3
6	5	4	3	2
5	4	3	2	1
4	3	2	1	0 ^t

- 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

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

		8						
8	7	1	6	5	4			
	8	6	6					
7	1	6	s	0	5	1	4	3
		6						
6	5	1	4	3	2			
5	4	3	2	1				
4	3	2	1	0	<i>t</i>			

- 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

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

		8	8					
8		7	1	6	2	5		4
	8		6		6		6	
7	1	6	0	5	1	4	2	3
	8		6		6			
6	2	5	1	4	2	3		2
			6					
5	4	2	3		2			1
4	3	2	1		0			t

- 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

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

		8	8	8				
8		7	1	6	2	5	3	4
	8		6	6	6			
7	1	6	0	5	1	4	2	3
	8		6	6	6			
6	2	5	1	4	2	3	3	2
	8		6	6				
5	3	4	2	3	3	2		1
		6						
4	3	3	2		1			0 ^t

- 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

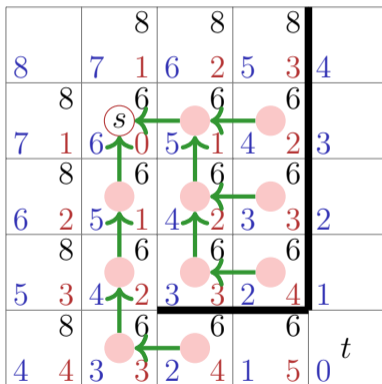
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x + \delta_y)$$

		8	8	8				
8		7	1	6	2	5	3	4
	8		6	6	6			
7	1	6	0	5	1	4	2	3
	8		6	6	6			
6	2	5	1	4	2	3	3	2
	8		6	6	6			
5	3	4	2	3	3	2	4	1
	8		6	6				
4	4	3	3	2	4	1		0 ^t

- 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

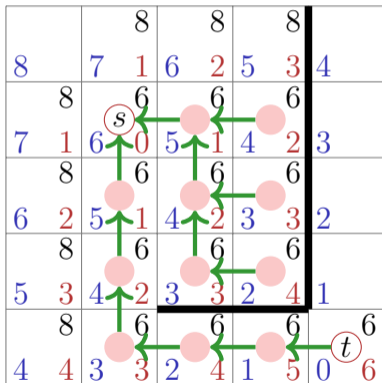
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x + \delta_y)$$



- 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

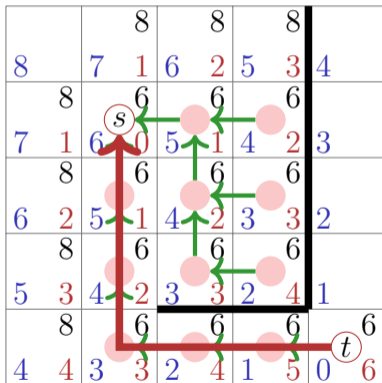
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x + \delta_y)$$



- 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

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



- 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 $\hat{h}_t(v) \leq h_t(v) := \delta(v, t) \forall v \in V$.
- 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 $\hat{h}(v) \leq \delta(v, t)$

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

foreach $u \in V$ **do**

$d[u] \leftarrow \infty$; $\hat{f}[u] \leftarrow \infty$; $\pi[u] \leftarrow \text{null}$

$d[s] \leftarrow 0$; $\hat{f}[s] \leftarrow \hat{h}(s)$; $N \leftarrow \{s\}$; $K \leftarrow \{\}$

while $N \neq \emptyset$ **do**

$u \leftarrow \text{ExtractMin}_{\hat{f}}(N)$; $K \leftarrow K \cup \{u\}$

if $u = t$ **then return** success

foreach $v \in N^+(u)$ with $d[v] > d[u] + c(u, v)$ **do**

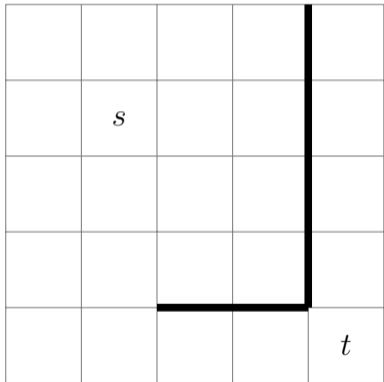
$d[v] \leftarrow d[u] + c(u, v)$; $\hat{f}[v] \leftarrow d[v] + \hat{h}(v)$; $\pi[v] \leftarrow u$

$N \leftarrow N \cup \{v\}$; $K \leftarrow K - \{v\}$

return failure

What if \hat{h} does not underestimate

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



- 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

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

32	25	20	17	16
25	18 ^s	13	10	9
20	13	8	5	4
17	10	5	2	1
16	9	4	1	0 ^t

- 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

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

32	25	20	17	16	
	18 ^s				
25	18	0	13	10	9
20	13	8	5	4	
17	10	5	2	1	
16	9	4	1	0 ^t	

- 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

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

		26					
32	25	1	20	17	16		
	26	18	14				
25	1	18	0	13	1	10	9
		14					
20	13	1	8	5	4		
17	10		5	2	1		
16	9		4	1	0	t	

- 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

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

		26	22					
32	25	1	20	2	17	16		
	26	18	14	12				
25	1	18	0	13	1	10	2	9
		14	10					
20	13	1	8	2	5	4		
17	10		5	2	1			
16	9		4	1	0	t		

- 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

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

		26	22			
32	25	1	20	2	17	16
	26	18	14		12	
25	1	18	0	13	1	10
		14	10		13	
20	13	1	8	2	5	3
			8			
17	10		5	3	2	1
16	9	4	1			0 ^{<i>t</i>}

Diagram description: A grid with values. A thick black vertical line is between columns 4 and 5. A thick black horizontal line is between rows 7 and 8. A red circle 's' is at (row 3, col 2). A red circle is at (row 3, col 3). A red circle is at (row 4, col 3). Green arrows point from (row 3, col 3) to (row 3, col 2) and from (row 4, col 3) to (row 3, col 3).

- 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

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

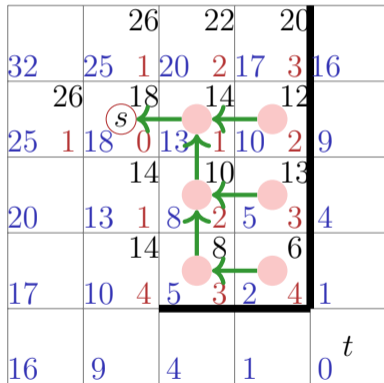
		26	22					
32	25	1	20	2	17	16		
	26	18	14		12			
25	1	18	0	13	1	10	2	9
		14	10		13			
20	13	1	8	2	5	3	4	
		14	8		6			
17	10	4	5	3	2	4	1	
16	9	4	1	0	t			

Diagram illustrating a search space grid. The grid contains numerical values representing distances. A path is highlighted with green arrows and pink circles, starting from a node labeled 's' (circled in red) and moving towards a node labeled 't'. The path consists of nodes (18, 14), (18, 13), (10, 13), (8, 13), and (8, 8). A thick black vertical line is drawn between columns 4 and 5, and a thick black horizontal line is drawn between rows 7 and 8.

- 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

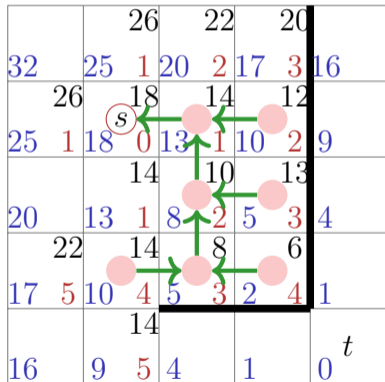
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- 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

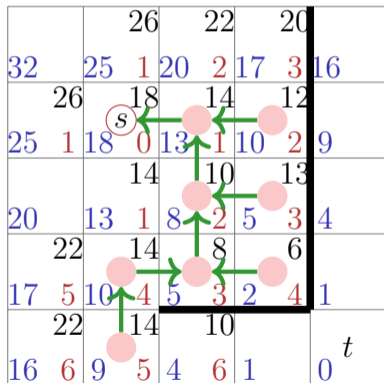
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- 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

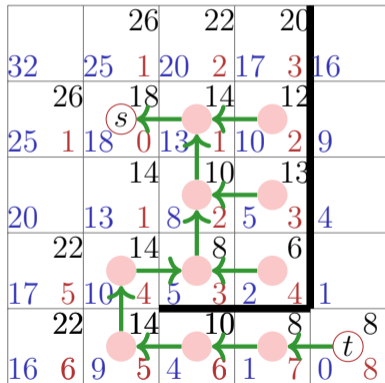
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- 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

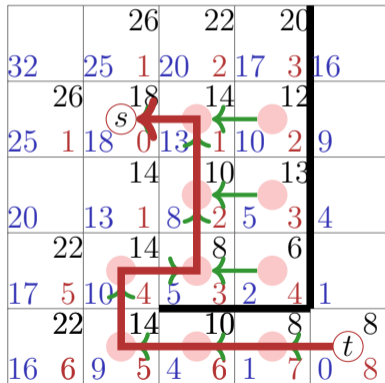
$$\hat{f}(u) = d_s[u] + \hat{h}(u) \quad (\hat{h} = \delta_x^2 + \delta_y^2)$$



- 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

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



- 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 \hat{h} , in addition to being admissible ($\hat{h}(v) \leq h(v)$ for all $v \in V$), fulfils monotonicity, i.e. if for all $(u, u') \in E$:

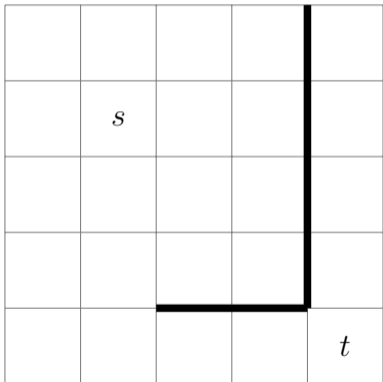
$$\hat{h}(u') \leq \hat{h}(u) + c(u', u)$$

then the A*-Algorithm is equivalent to the Dijkstra-algorithm with edge weights $\tilde{c}(u, v) = c(u, v) + \hat{h}(u) - \hat{h}(v)$, 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}

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

8	7	6	5	4
7	0 ^s	0	0	3
6	5	1	0	2
5	0	0	0	1
4	0	2	1	0 ^t

- 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}

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

8	7	6	5	4
7	0 ⁰ s	0	0	3
6	5	1	0	2
5	0	0	0	1
4	0	2	1	0 ^t

- 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}

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

		8					
8	7	1	6	5	4		
	8	0		1			
7	1	0	0	0	1	0	3
		6					
6	5	1	1	0	2		
5	0	0	0	0	1		
4	0	2	1	0	t		

- 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}

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

		8	8							
8		7	1	6	2	5	4			
	8		0		1		2			
7	1	0	s	0	0	1	0	2	3	
		6		2						
6		5	1	1	2	0			2	
5		0		0	0				1	
4		0		2	1				0	<i>t</i>

- 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}

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

		8	8	8			
8		7	1	6	2	5	3
	8		0		1		2
7	1	0	0	0	1	0	2
		6			2		3
6	5	1	1	2	0	3	2
			3				
5	0	0	3	0			1
4	0	2	1				t

- 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}

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

		8	8	8					
8		7	1	6	2	5	3	4	
	8		0		1		2		
7	1	0	(s)	0	0	1	0	2	3
		6		2		3			
6	5	1	1	2	0	3	2		
		4		3		4			
5	0	4	0	3	0	4	1		
4	0	2	1	0	t				

- 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}

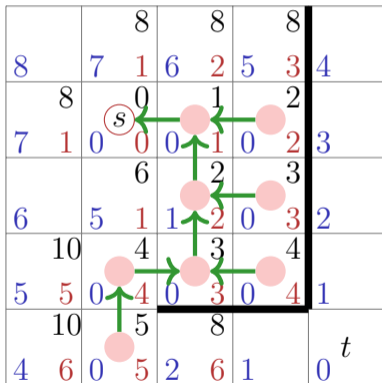
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$

		8	8	8			
8	7	1	6	2	5	3	4
	8	0	1		2		
7	1	0	0	0	1	0	2
		6	2		3		
6	5	1	1	2	0	3	2
	10	4	3		4		
5	5	0	4	0	3	0	4
		5					
4	0	5	2	1	0		t

- 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}

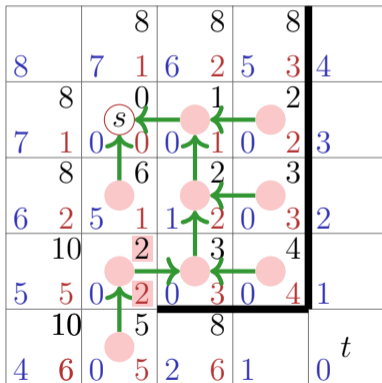
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- 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}

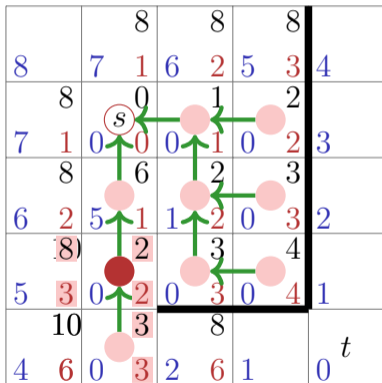
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- 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}

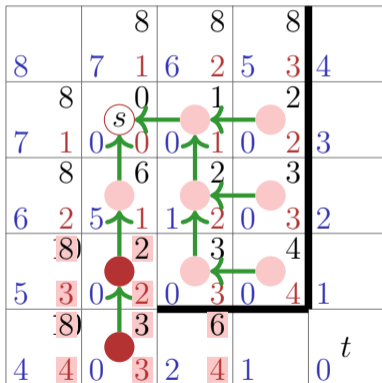
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- 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}

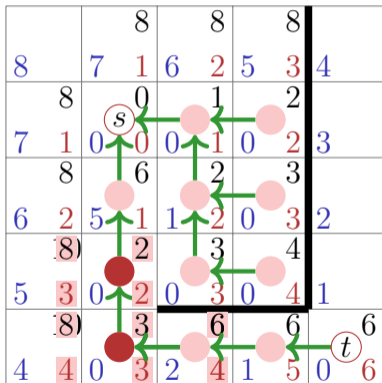
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- 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}

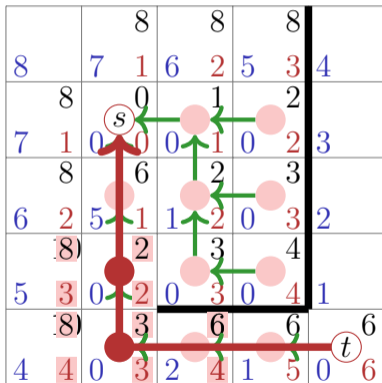
$$\hat{f}(u) = d_s[u] + \hat{h}(u)$$



- 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

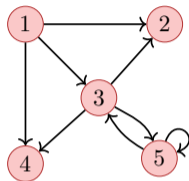
- The A*-Algorithm is an extension of the Dijkstra algorithm by a distance heuristic \hat{h} .
- A* = Dijkstra if $\hat{h} \equiv 0$
- If \hat{h} underestimates the real distance, the algorithm works correctly.
- If \hat{h} is monotone in addition, then the algorithm works efficiently.
- In practical 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 $A_G \hat{=} \text{Relation } E \subseteq V \times V \text{ over } V$

Graphs and Relations

Graph $G = (V, E)$

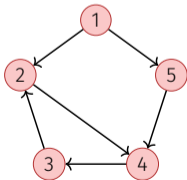
adjacencies $A_G \hat{=} \text{Relation } E \subseteq V \times V \text{ over } V$

- **reflexive** $\Leftrightarrow a_{i,i} = 1$ for all $i = 1, \dots, n$. (loops)
- **symmetric** $\Leftrightarrow a_{i,j} = a_{j,i}$ for all $i, j = 1, \dots, n$ (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 .

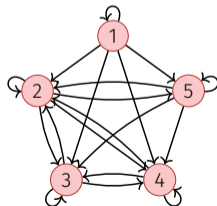
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$



$G = (V, E)$



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



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

Algorithm $A \cdot A$

Input: (Adjacency-)Matrix $A = (a_{ij})_{i,j=1\dots n}$

Output: Matrix Product $B = (b_{ij})_{i,j=1\dots n} = A \cdot A$

$B \leftarrow 0$

for $r \leftarrow 1$ **to** n **do**

for $c \leftarrow 1$ **to** n **do**

for $k \leftarrow 1$ **to** n **do**

$b_{rc} \leftarrow b_{rc} + a_{rk} \cdot a_{kc}$

// Number of Paths

return B

Counts number of paths of length 2

Algorithm $A \otimes A$

Input: Adjacency-Matrix $A = (a_{ij})_{i,j=1\dots n}$

Output: Modified Matrix Product $B = (b_{ij})_{i,j=1\dots n} = A \otimes A$

```
 $B \leftarrow A$  // Keep paths
for  $r \leftarrow 1$  to  $n$  do
  for  $c \leftarrow 1$  to  $n$  do
    for  $k \leftarrow 1$  to  $n$  do
       $b_{rc} \leftarrow \max\{b_{rc}, a_{rk} \cdot a_{kc}\}$  // 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 $B = (b_{ij})_{1 \leq i, j \leq n}$ with $b_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E^*$ First idea:

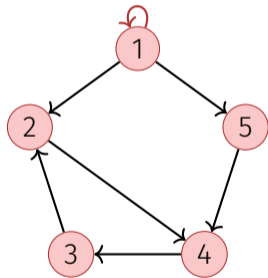
- Start with $B \leftarrow A$ and set $b_{ii} = 1$ for each i (Reflexivity.).
- Compute

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

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

Improvement: Algorithm of Warshall (1962)

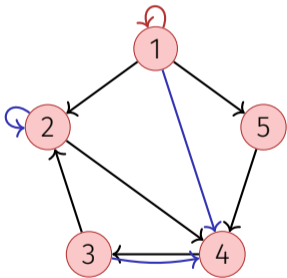
Inductive procedure: all paths known over nodes from $\{v_i : i < k\}$. Add node v_k .



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

Improvement: Algorithm of Warshall (1962)

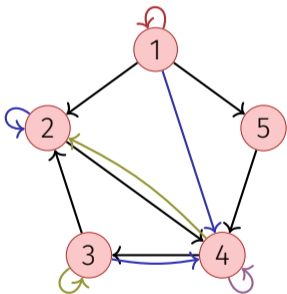
Inductive procedure: all paths known over nodes from $\{v_i : i < k\}$. Add node v_k .



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

Improvement: Algorithm of Warshall (1962)

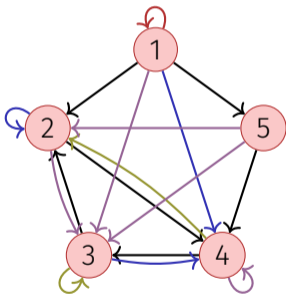
Inductive procedure: all paths known over nodes from $\{v_i : i < k\}$. Add node v_k .



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

Improvement: Algorithm of Warshall (1962)

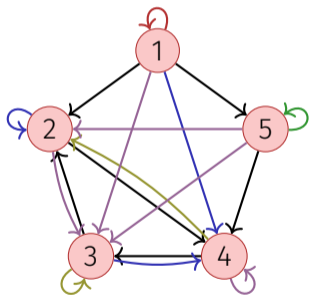
Inductive procedure: all paths known over nodes from $\{v_i : i < k\}$. Add node v_k .



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

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

$B \leftarrow A_G$

for $k \leftarrow 1$ **to** n **do**

$b_{kk} \leftarrow 1$

// Reflexivity

for $r \leftarrow 1$ **to** n **do**

for $c \leftarrow 1$ **to** n **do**

$b_{rc} \leftarrow \max\{b_{rc}, b_{rk} \cdot b_{kc}\}$

// All paths via v_k

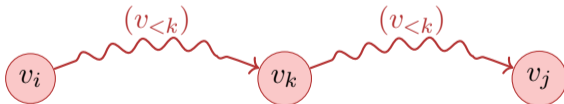
return B

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 ($k \rightarrow k + 1$):** 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 in⁴² $V^k := \{v_1, \dots, v_k\}$, 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} .

⁴²like 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 $h : V \rightarrow \mathbb{R}$
- 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 $G = (V, E, c)$ iff it is shortest path in $\tilde{G} = (V, E, \tilde{c})$

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

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: $\tilde{c}(v_0, \dots, v_k = v_0) = c(v_0, \dots, v_k = v_0)$

Johnson's Algorithm

Add a new node $s \notin V$:

$$G' = (V', E', c')$$

$$V' = V \cup \{s\}$$

$$E' = E \cup \{(s, v) : v \in V\}$$

$$c'(u, v) = c(u, v), \quad u \neq s$$

$$c'(s, v) = 0 (v \in V)$$

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**

└ $\tilde{d}(u, \cdot) \leftarrow \text{Dijkstra}(\tilde{G}', u)$

foreach $v \in V$ **do**

└ $D(u, v) \leftarrow \tilde{d}(u, v) + h(v) - h(u)$

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

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