University of Antwerp
Operations Research Group
ANT/OR

Shortest path problem and algorithms

Jochen Janssens

June 3, 2013, ANT/OR fORum, Antwerp

Problem description

If we assume a network where each edge has a value associated with it, the **shortest-path problem** can be described as the problem of finding the route between two vertices of the network which has the minimal cost. [20]

Use:

- Vehicle routing (shortest path from city A to city B)
- Computer networks
- Social networks (degree of separation)
- AI in video games
- Cutting or painting robot
- ...

A more formal definition

Given a directed graph G(V, A) with source node s, destination d

$$\begin{aligned} \min \quad & \sum w_{ij} x_{ij} \\ \text{s.t.} \quad & x_{ij} \geq 0 \ \forall (i,j) \in A \\ & \sum_j (x_{ij} - x_{ji}) = \begin{cases} 1, & \text{if i=s;} \\ -1, & \text{if i=d;} \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

where w_{ij} is the cost for each arc (i,j) in A [8]

Classification

- Single-pair shortest path problem (1 source to 1 destination)
- Single-source shortest path problem (1 source to n destinations)
- Single-destination shortest path problem (n sources to 1 destination)
- All-pairs shortest path problem (n sources to n destinations)

Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: s

Lowest cost from s until node (so far):

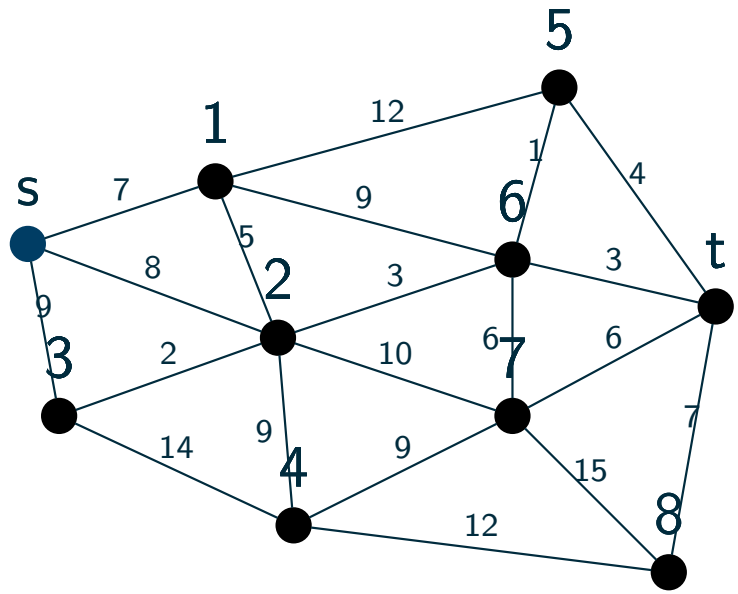
s	1	2	3	4	5	6	7	8	9	t
0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	-	-	-	-	-	-	-	-	-	-

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: s

Lowest cost from s until node (so far):

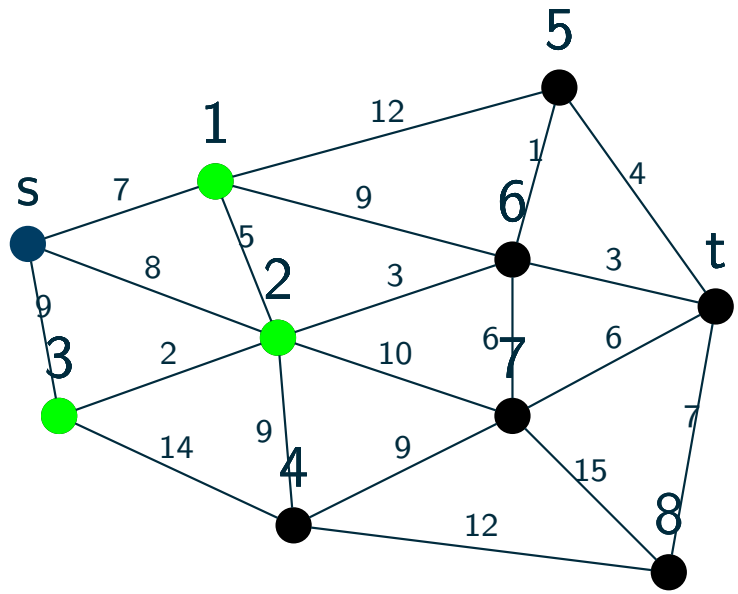
s	1	2	3	4	5	6	7	8	9	t
0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	-	-	-	-	-	-	-	-	-	-

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: s

Lowest cost from s until node (so far):

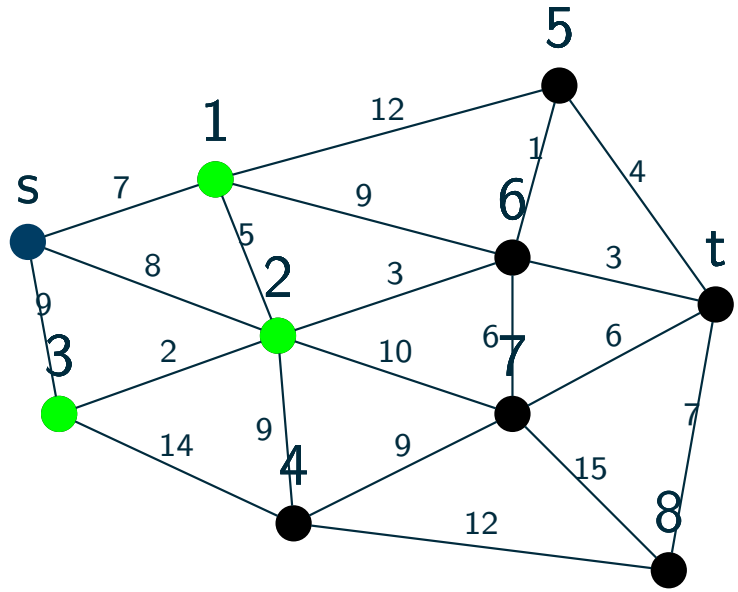
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	∞	∞	∞	∞	∞	∞	∞

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	-	-	-	-	-	-	-

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: s

Lowest cost from s until node (so far):

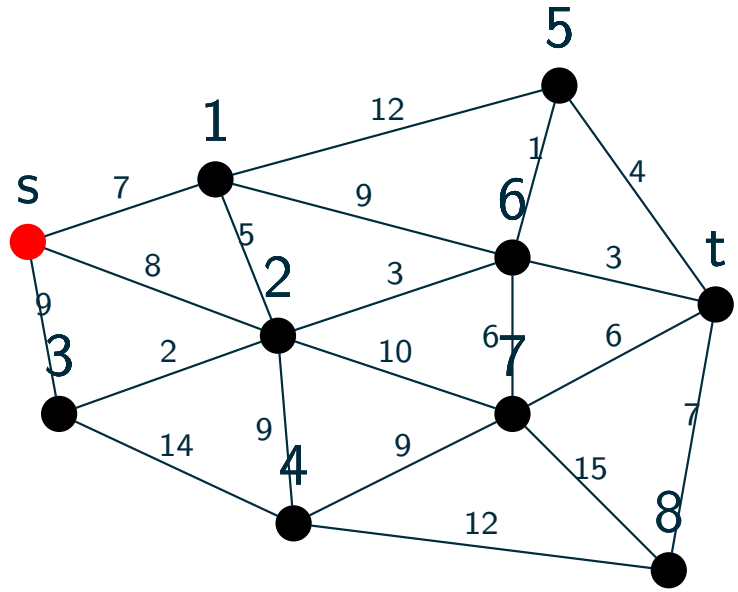
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	∞	∞	∞	∞	∞	∞	∞

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	-	-	-	-	-	-	-

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: 1

Lowest cost from s until node (so far):

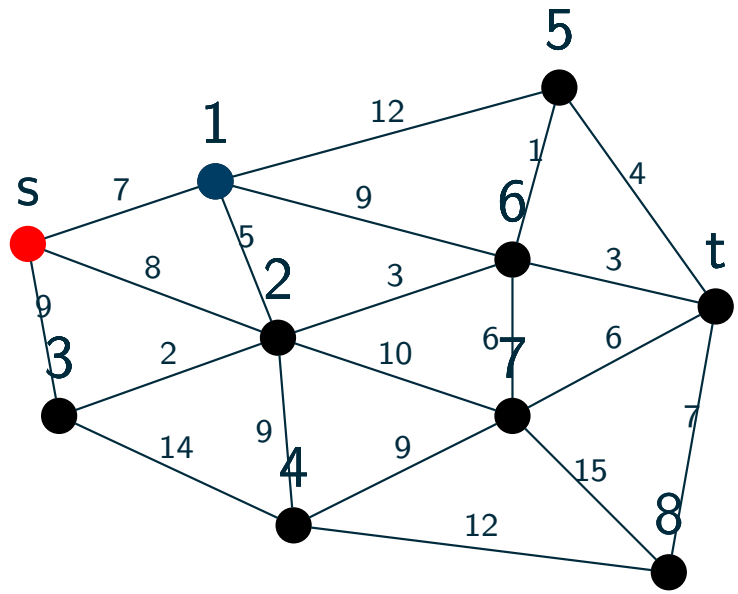
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	∞	∞	∞	∞	∞	∞	∞

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	-	-	-	-	-	-	-

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: 1

Lowest cost from s until node (so far):

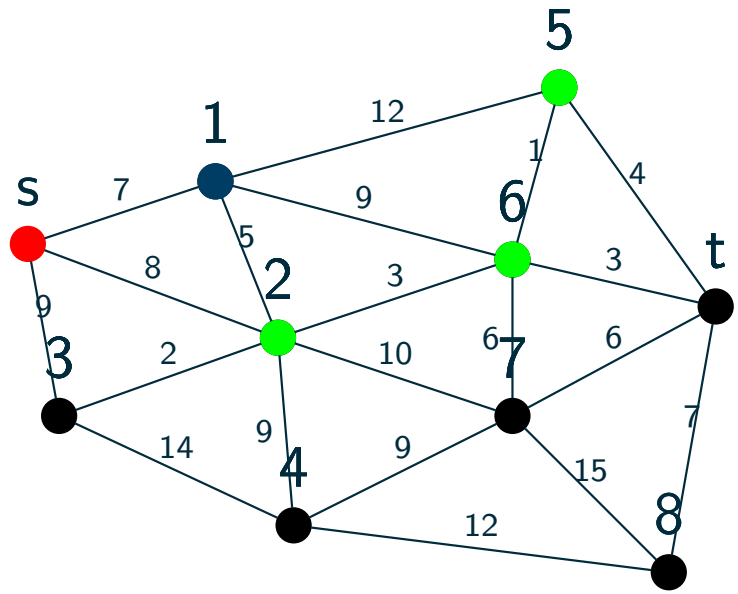
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	∞	19	16	∞	∞	∞	∞

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	-	1	1	-	-	-	-

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: 1

Lowest cost from s until node (so far):

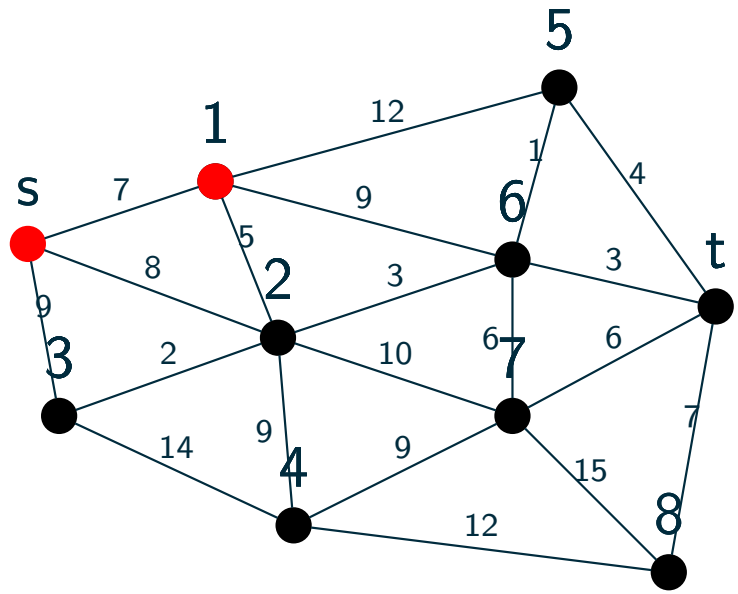
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	∞	19	16	∞	∞	∞	∞

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	-	1	1	-	-	-	-

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: 2

Lowest cost from s until node (so far):

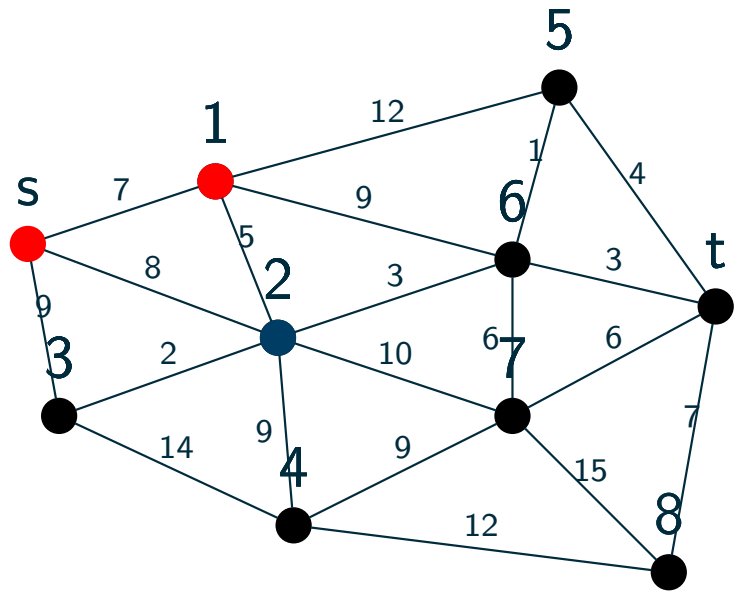
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	∞	19	16	∞	∞	∞	∞

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	-	1	1	-	-	-	-

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: 2

Lowest cost from s until node (so far):

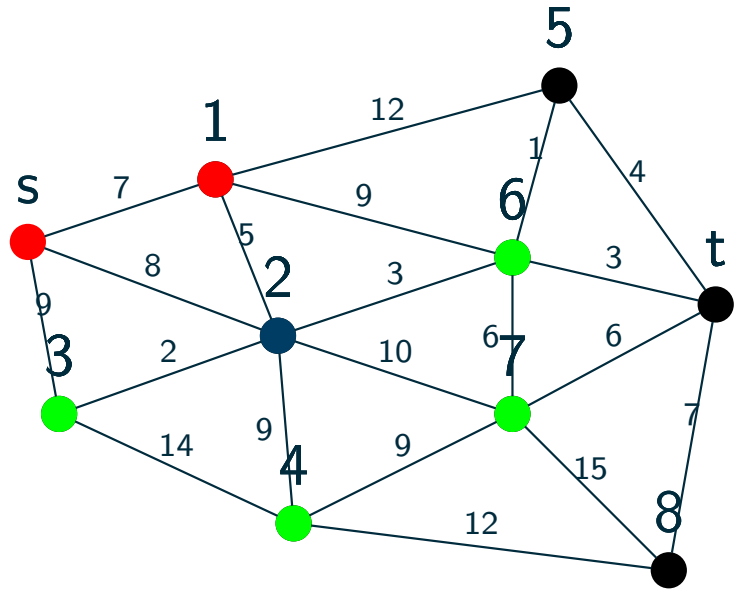
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	17	19	11	18	∞	∞	∞

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	2	1	2	2	-	-	-

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: 2

Lowest cost from s until node (so far):

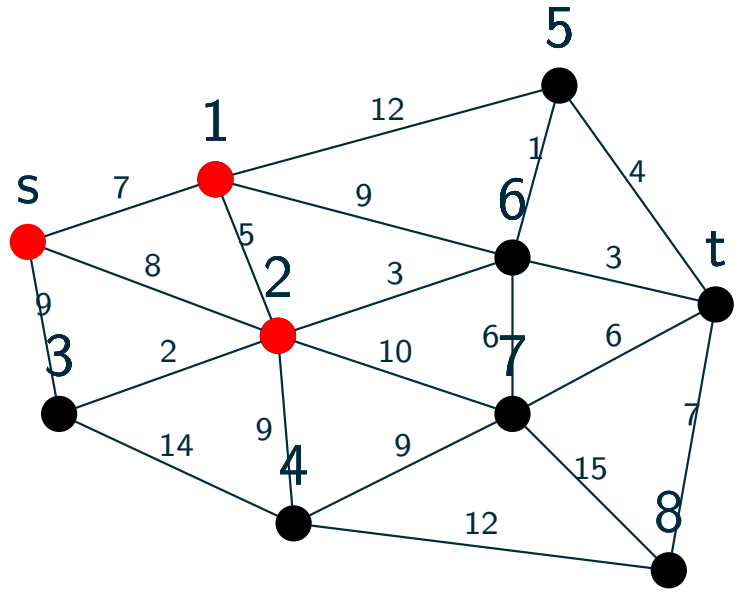
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	17	19	11	18	∞	∞	∞

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	2	1	2	2	-	-	-

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: 3

Lowest cost from s until node (so far):

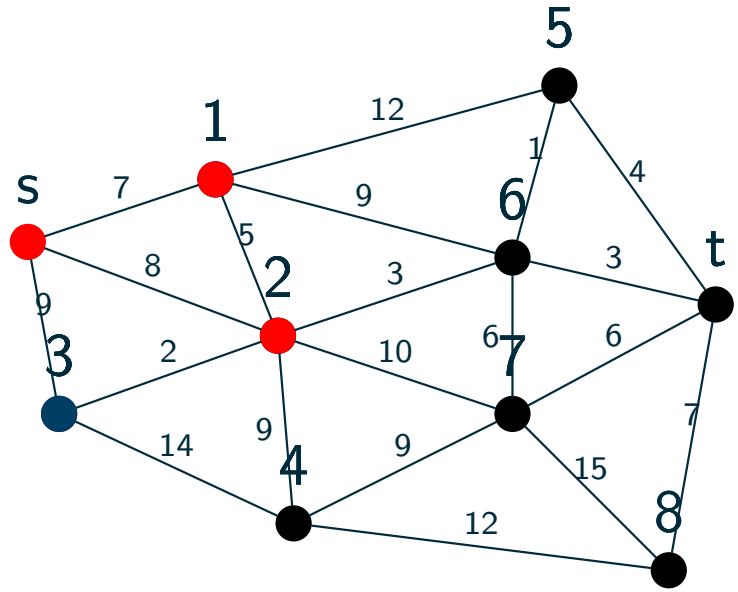
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	17	19	11	18	∞	∞	∞

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	2	1	2	2	-	-	-

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: 3

Lowest cost from s until node (so far):

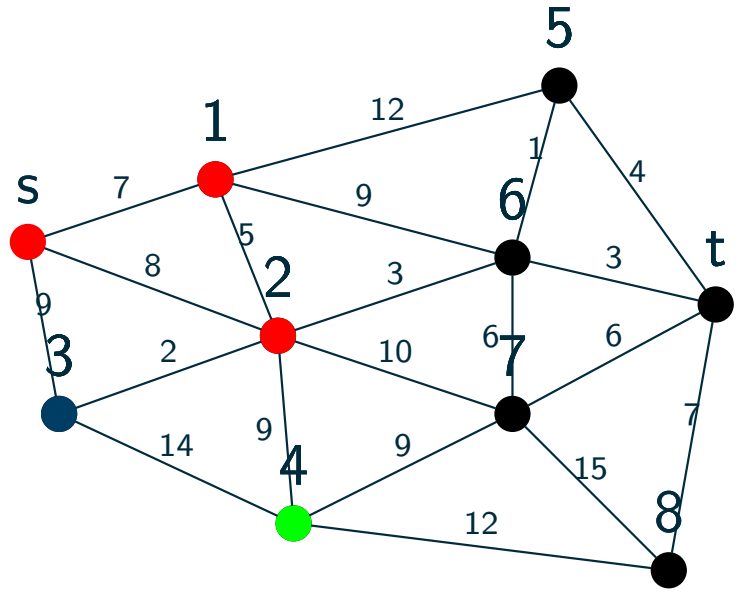
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	17	19	11	18	∞	∞	∞

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	2	1	2	2	-	-	-

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: 3

Lowest cost from s until node (so far):

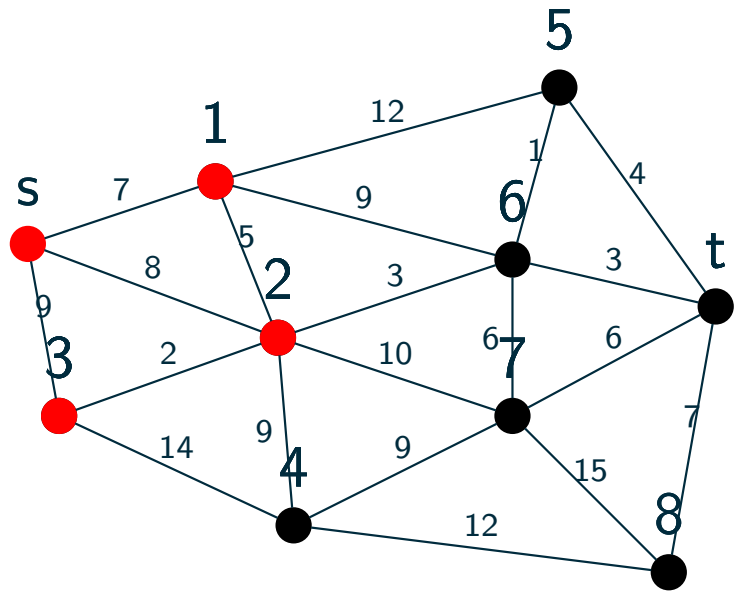
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	17	19	11	18	∞	∞	∞

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	2	1	2	2	-	-	-

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: 6

Lowest cost from s until node (so far):

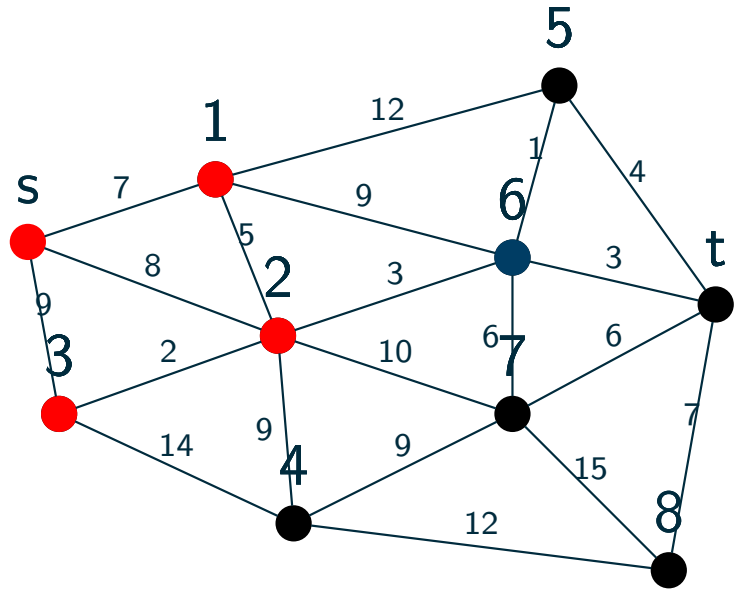
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	17	19	11	18	∞	∞	∞

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	2	1	2	2	-	-	-

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: 6

Lowest cost from s until node (so far):

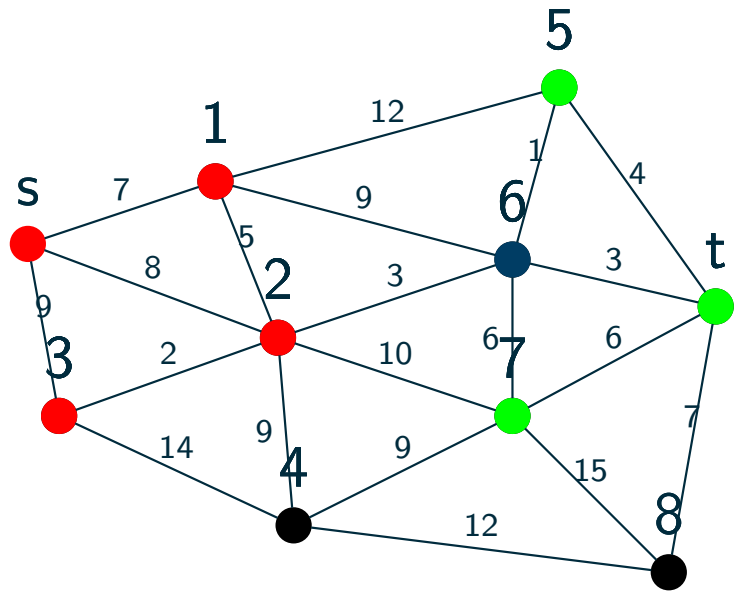
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	17	12	11	17	∞	∞	14

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	2	6	2	6	-	-	6

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: 6

Lowest cost from s until node (so far):

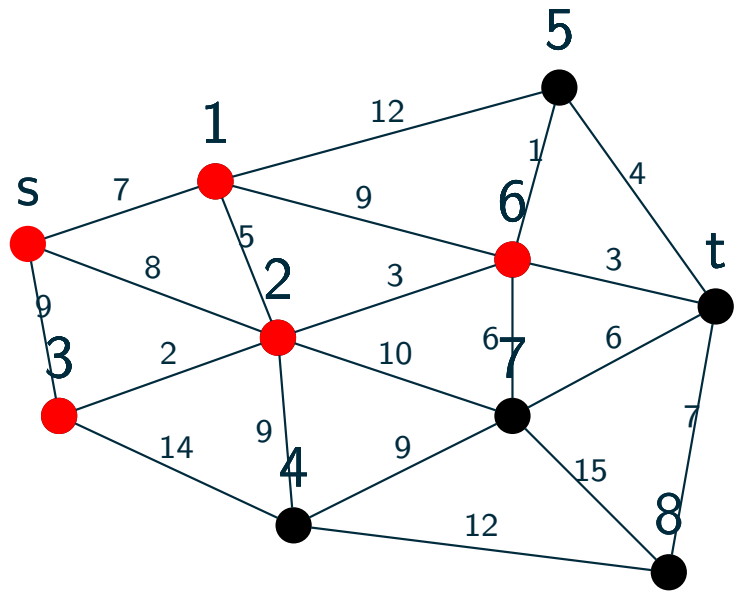
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	17	12	11	17	∞	∞	14

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	2	6	2	6	-	-	6

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: 5

Lowest cost from s until node (so far):

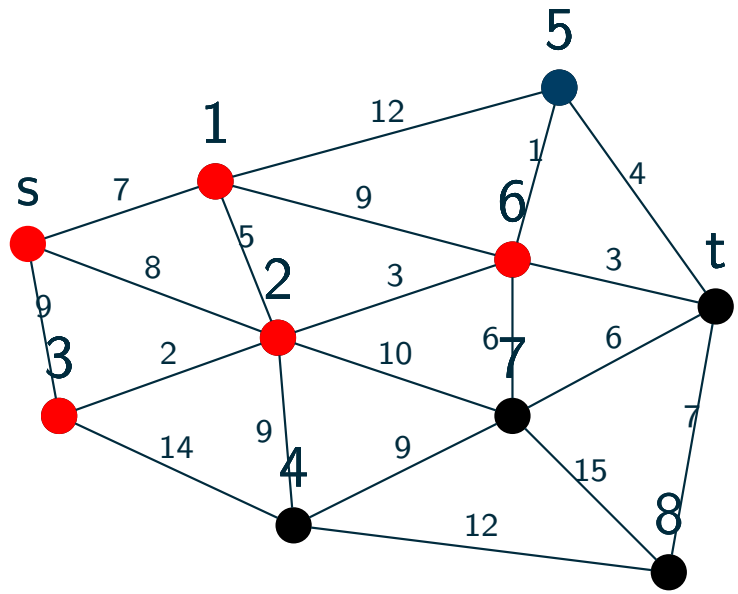
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	17	12	11	17	∞	∞	14

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	2	6	2	6	-	-	6

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: 5

Lowest cost from s until node (so far):

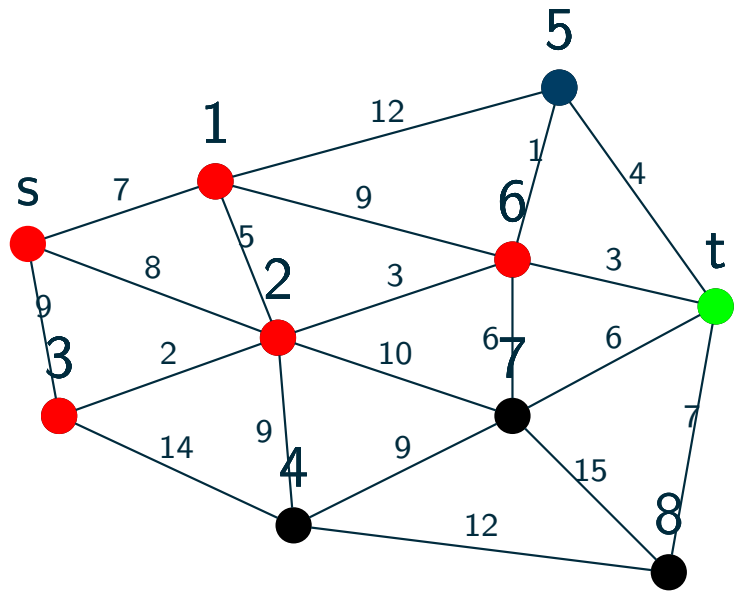
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	17	12	11	17	∞	∞	14

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	2	6	2	6	-	-	6

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: 5

Lowest cost from s until node (so far):

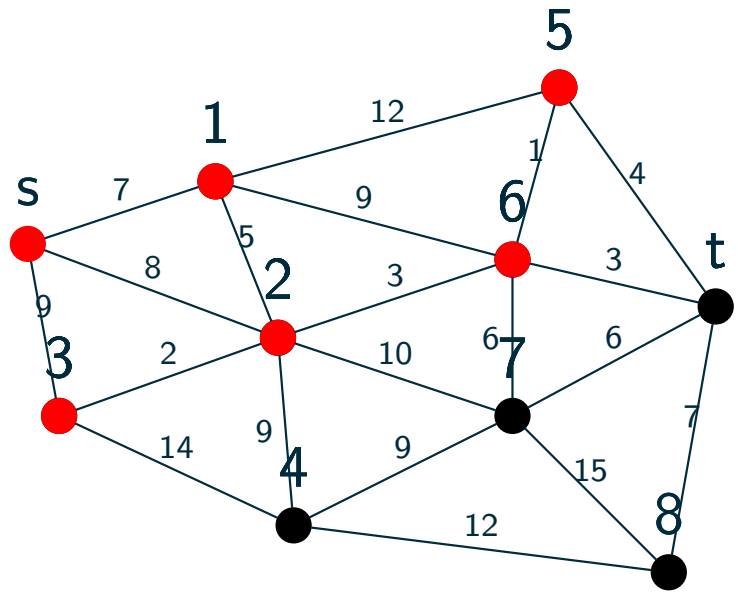
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	17	12	11	17	∞	∞	14

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	2	6	2	6	-	-	6

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: t

Lowest cost from s until node (so far):

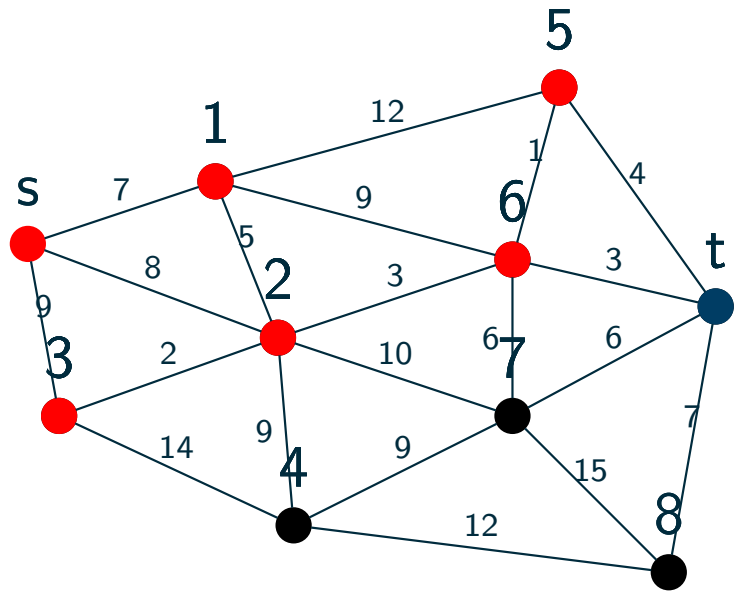
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	17	12	11	17	∞	∞	14

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	2	6	2	6	-	-	6

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: t

Lowest cost from s until node (so far):

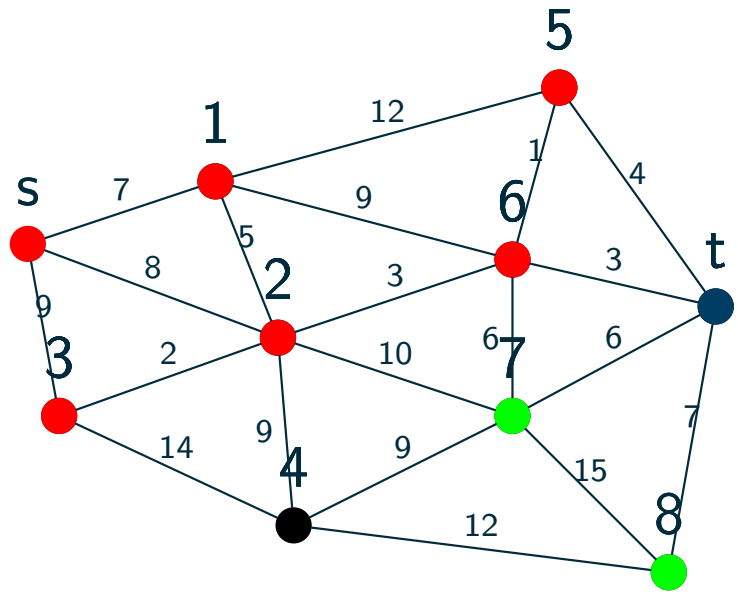
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	17	12	11	17	∞	∞	14

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	2	6	2	6	-	-	6

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked: t

Lowest cost from s until node (so far):

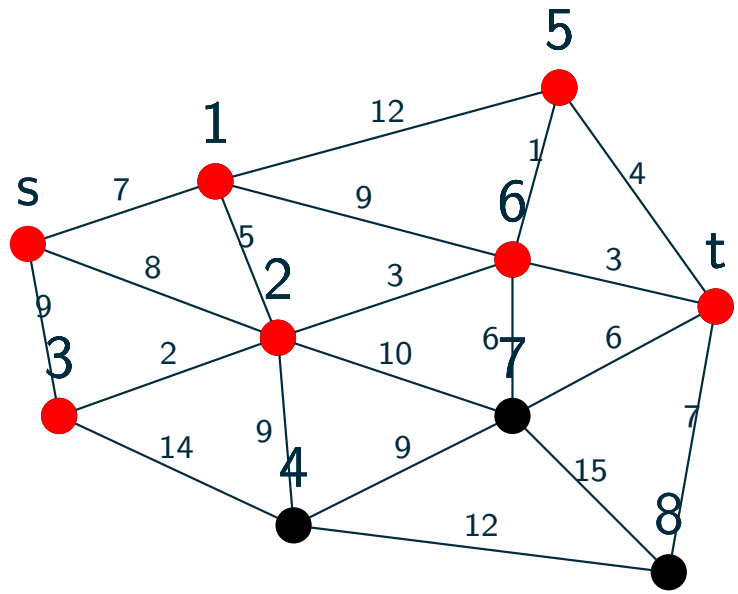
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	17	12	11	17	∞	∞	14

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	2	6	2	6	-	-	6

Already visited:

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



Dijkstra's algorithm

Can be both single-source (1 to n) as single-pair (1 to 1).
Can be speed up by using Fibonacci heap

1. assign to each node distance ∞, to source node distance 0;
2. mark all nodes unvisited, set source node as current node;
3. for the current node, consider all of its unvisited neighbours and calculate their *tentative* distances, if the distance is less than the previous one, overwrite that distance;
4. when all neighbours are checked, mark current node as visited;
5. if destination node is marked as visited or smallest tentative distance of unvisited nodes is ∞, stop;
6. select unvisited node with lowest tentative distance and set it as current node, go to step 3;

Currently checked:
Lowest cost from s until node (so far):

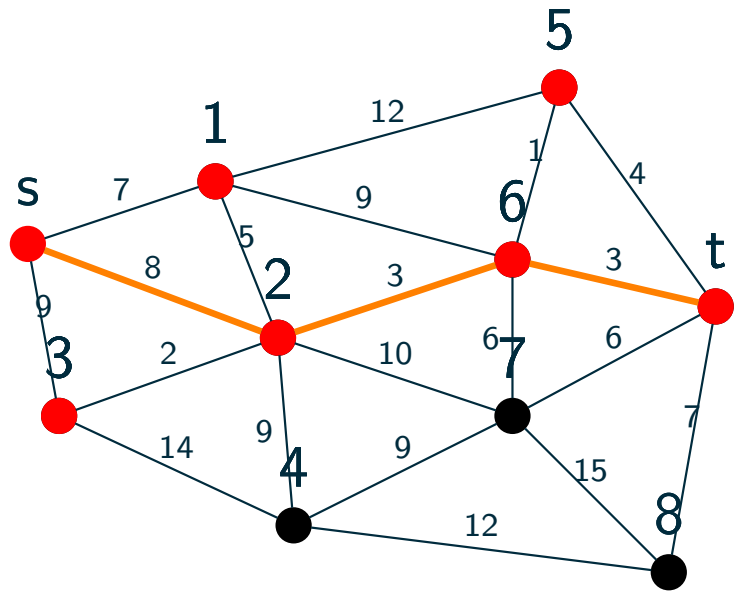
s	1	2	3	4	5	6	7	8	9	t
0	7	8	9	17	12	11	17	∞	∞	14

Previous node:

s	1	2	3	4	5	6	7	8	9	t
-	s	s	s	2	6	2	6	-	-	6

Already visited:

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



Bellman-Ford algorithm

Is single source (1 to n).

Slower than Dijkstra's algorithm.

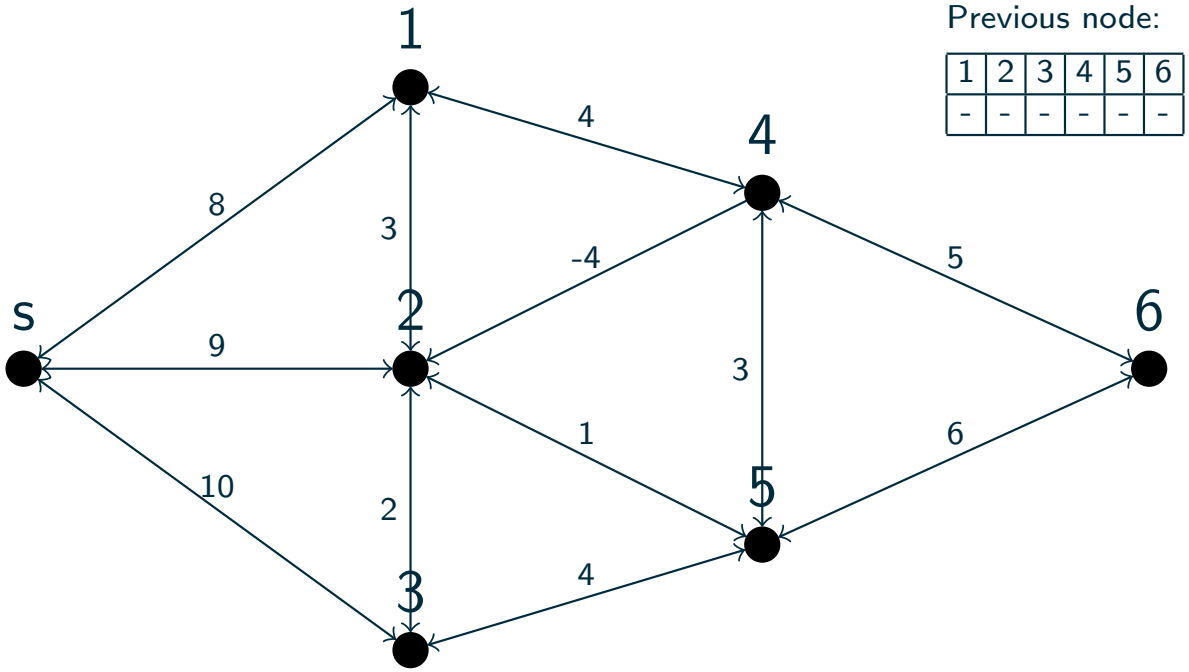
Can deal with negative edge weights (able to detect negative cycles).

Dijkstra greedily selects the minimum-weight node that has not yet been processed, Bellman-Ford relaxes *all* edges.

```
procedure BellmanFord(list vertices, list edges, vertex source){
// This implementation takes in a graph, represented as lists of vertices and edges,
// and fills two arrays (distance and predecessor) with shortest-path information

// Step 1: initialize graph
foreach vertex v in vertices: do
    if v is source then
        distance[v] := 0
    else
        distance[v] := infinity
    end
    predecessor[v] := null
end
// Step 2: relax edges repeatedly
for  i from 1 to size(vertices)-1: do
    foreach edge (u, v) with weight w in edges: do
        if distance[u] + w < distance[v]: then
            distance[v] := distance[u] + w
            predecessor[v] := u
        end
    end
end
// Step 3: check for negative-weight cycles
foreach edge (u, v) with weight w in edges: do
    if distance[u] + w < distance[v]: then
        error "Graph contains a negative-weight cycle"
    end
end
}
```

Bellman-Ford algorithm



To be updated: { s }

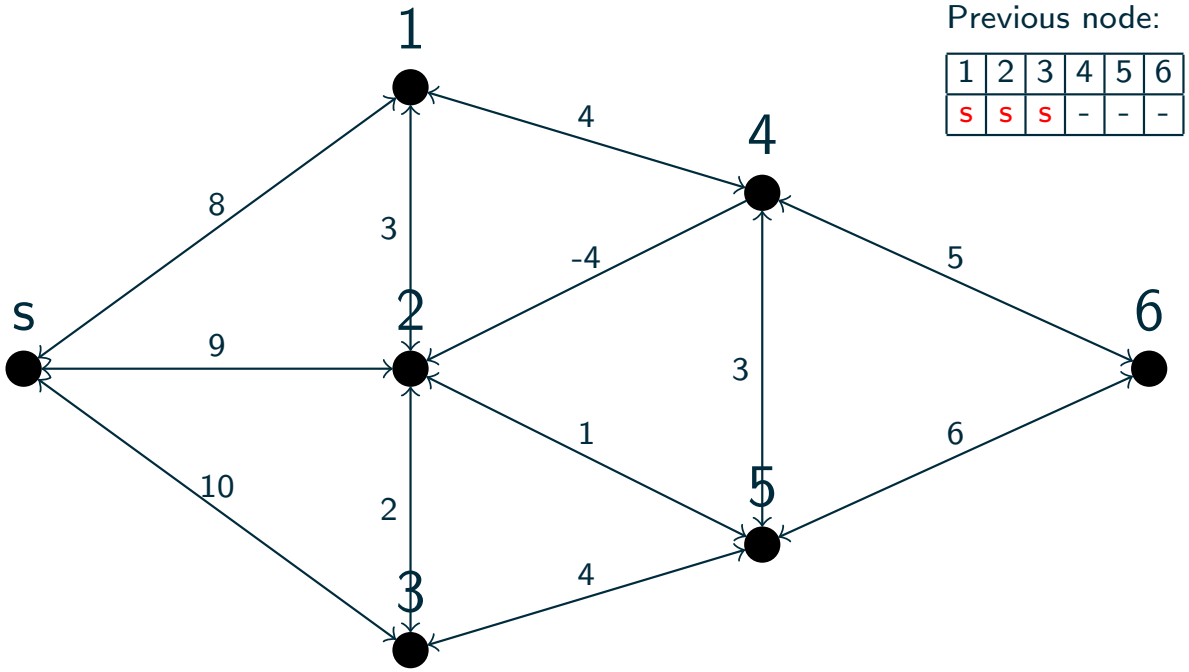
Lowest cost from s until node (so far):

1	2	3	4	5	6
∞	∞	∞	∞	∞	∞

Previous node:

1	2	3	4	5	6
-	-	-	-	-	-

Bellman-Ford algorithm



To be updated: { s }

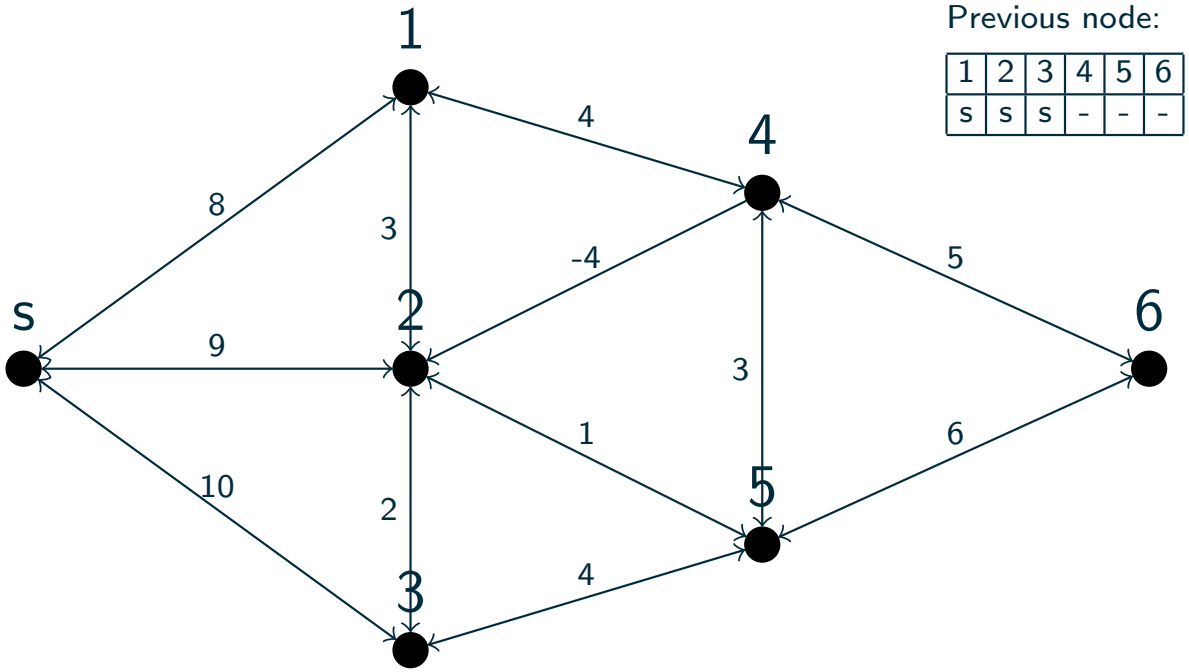
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	9	10	∞	∞	∞

Previous node:

1	2	3	4	5	6
s	s	s	-	-	-

Bellman-Ford algorithm



To be updated: { 1,2,3 }

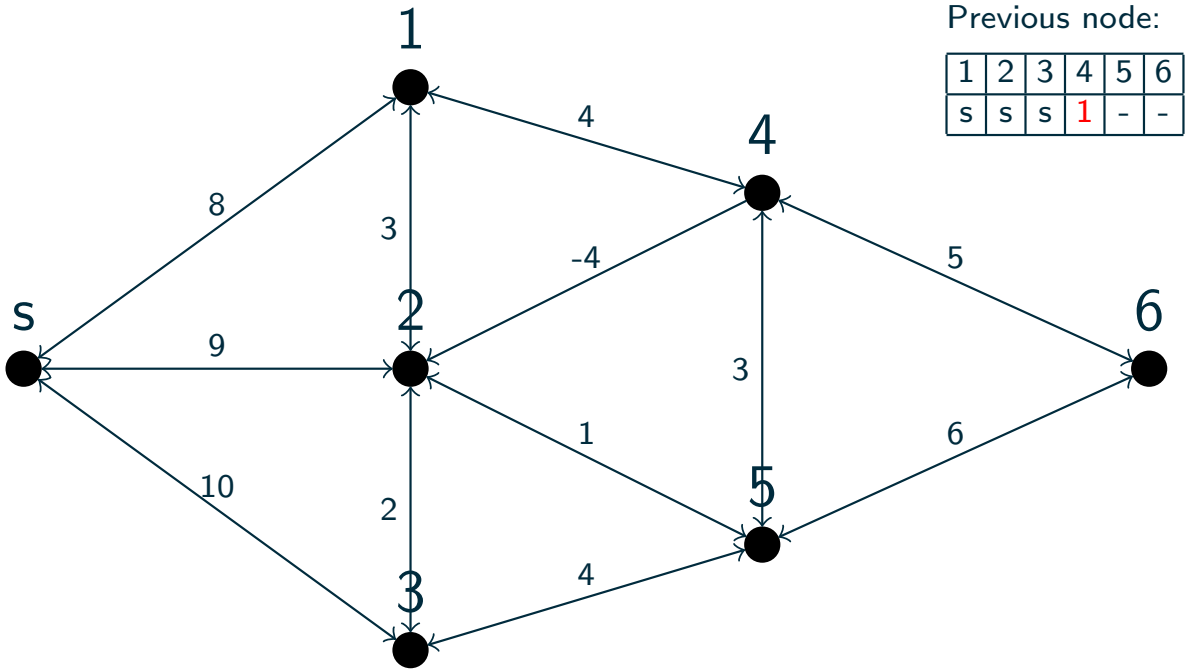
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	9	10	∞	∞	∞

Previous node:

1	2	3	4	5	6
s	s	s	-	-	-

Bellman-Ford algorithm



To be updated: { 1,2,3 }

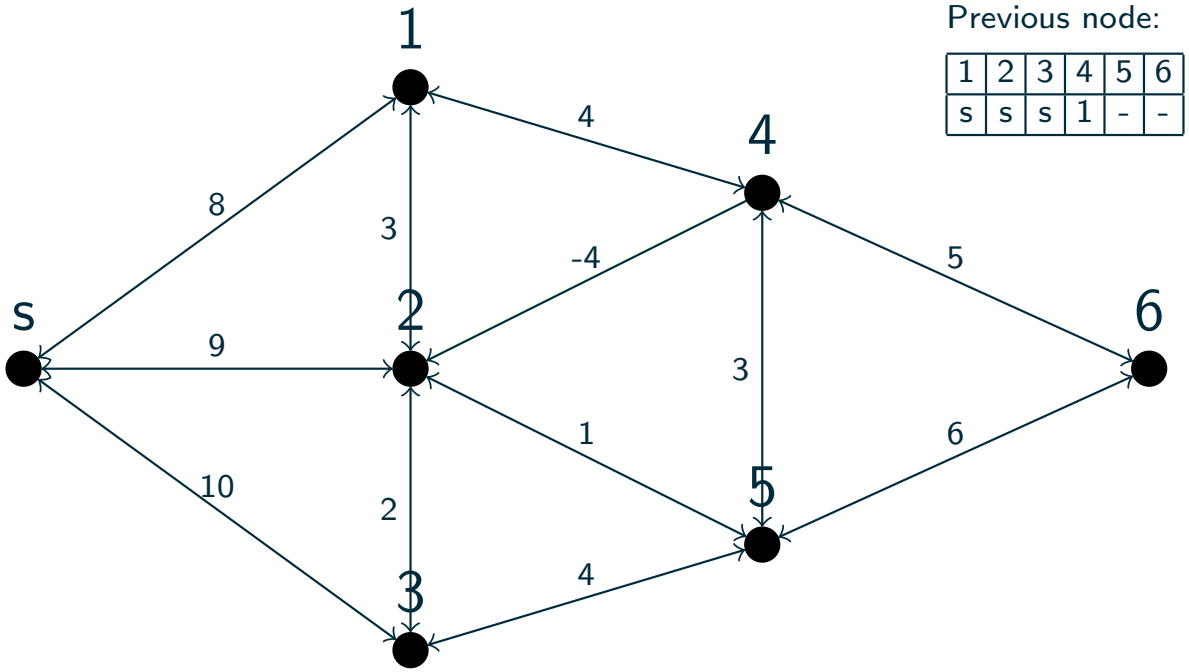
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	9	10	12	∞	∞

Previous node:

1	2	3	4	5	6
s	s	s	1	-	-

Bellman-Ford algorithm



To be updated: { 2,3,4 }

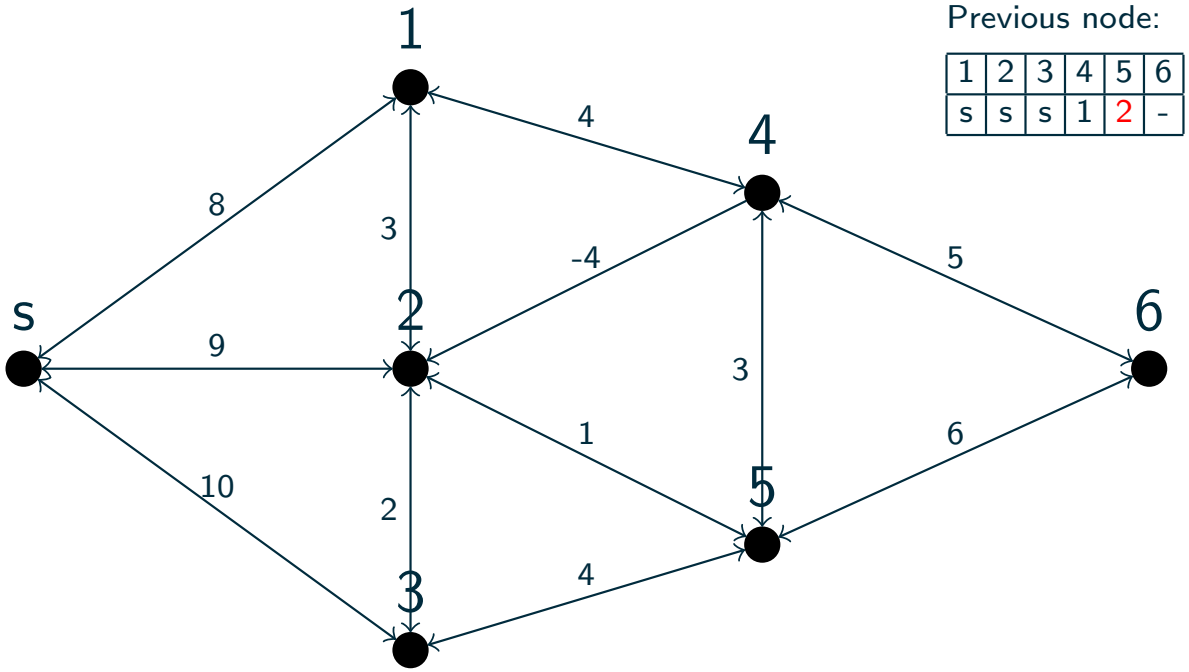
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	9	10	12	∞	∞

Previous node:

1	2	3	4	5	6
s	s	s	1	-	-

Bellman-Ford algorithm



To be updated: { 2,3,4 }

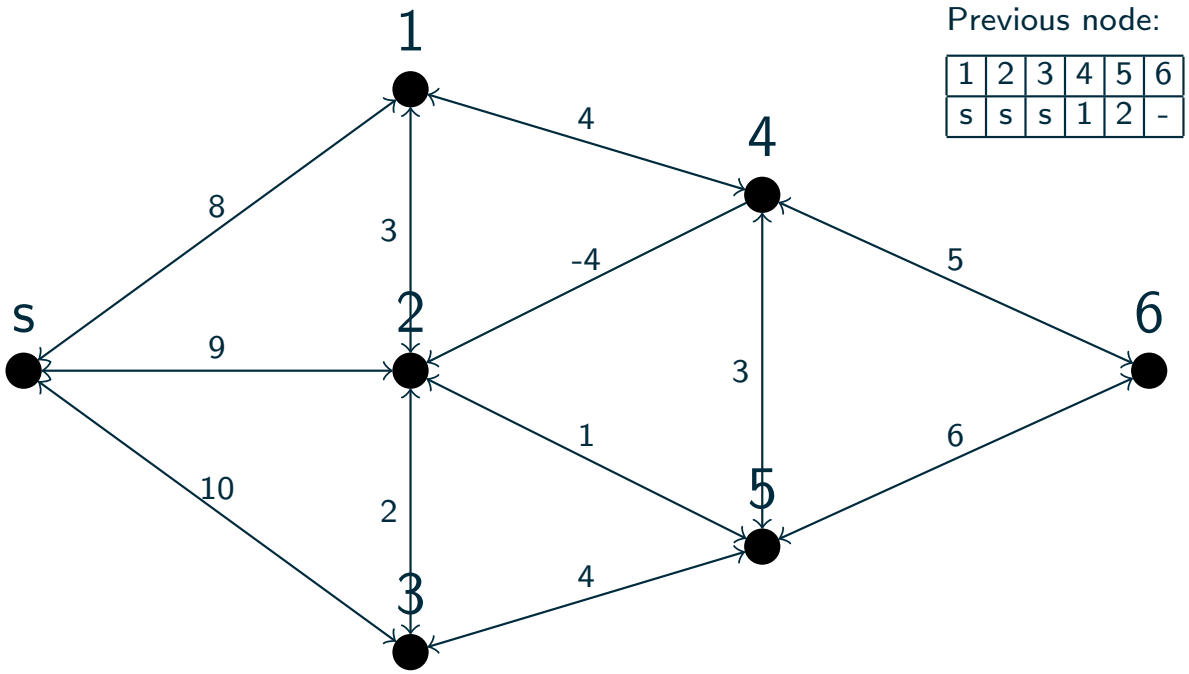
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	9	10	12	10	∞

Previous node:

1	2	3	4	5	6
s	s	s	1	2	-

Bellman-Ford algorithm



To be updated: { 3,4,5 }

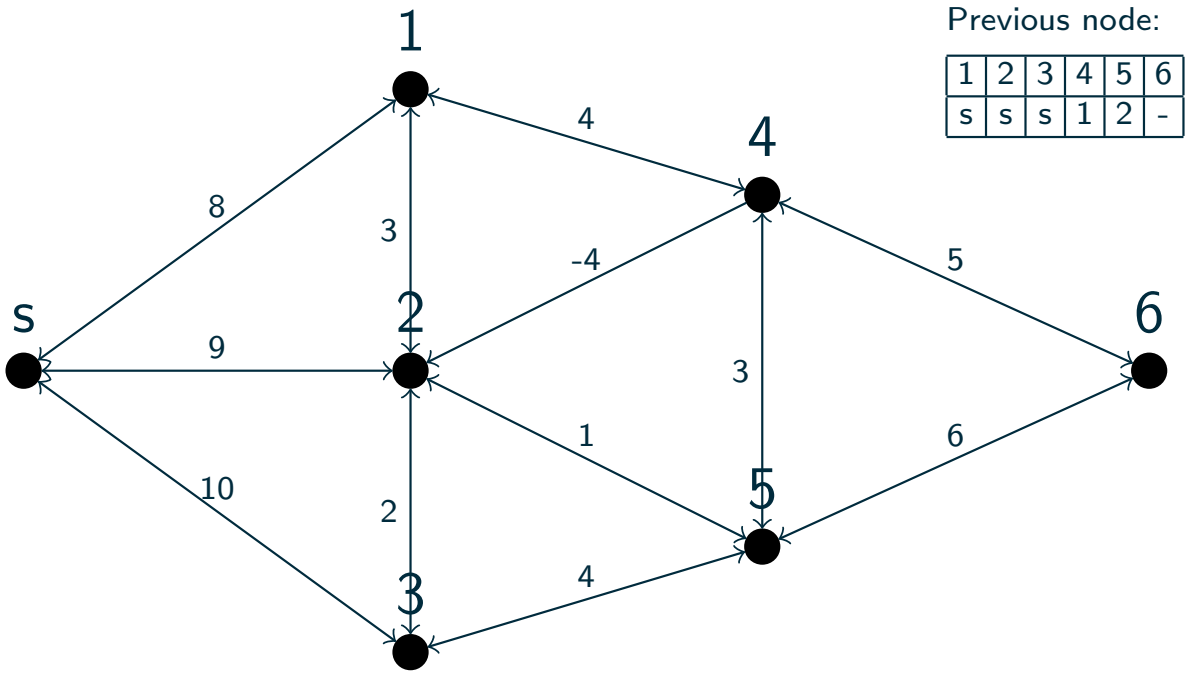
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	9	10	12	10	∞

Previous node:

1	2	3	4	5	6
s	s	s	1	2	-

Bellman-Ford algorithm



To be updated: { 3,4,5 }

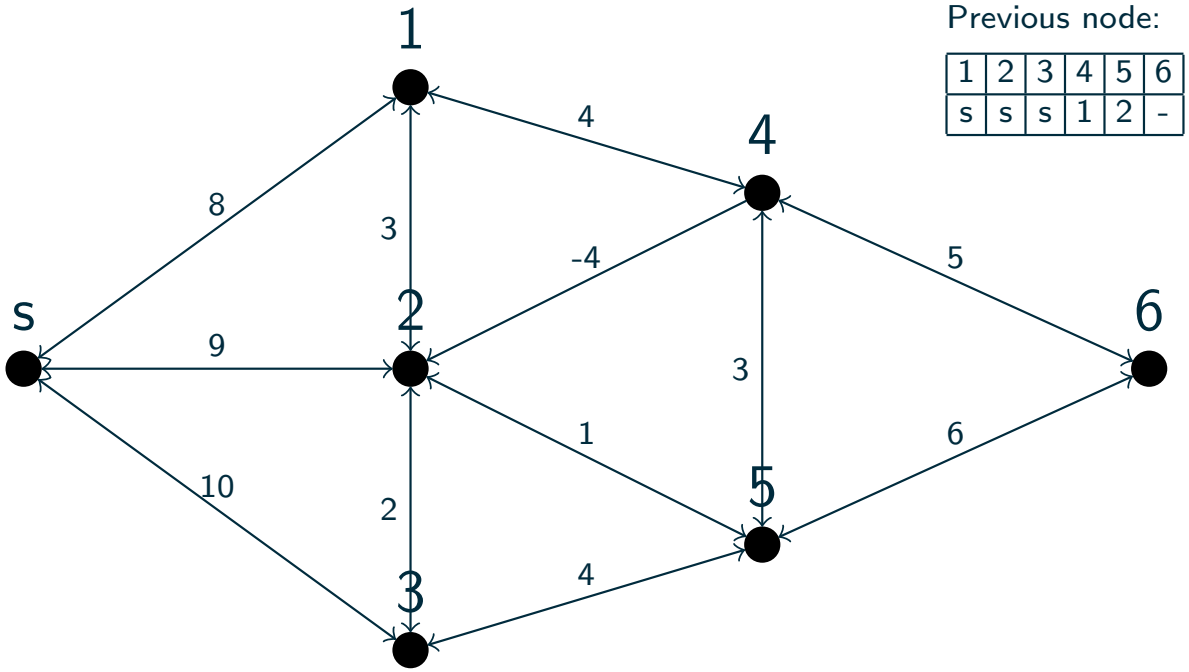
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	9	10	12	10	∞

Previous node:

1	2	3	4	5	6
s	s	s	1	2	-

Bellman-Ford algorithm



To be updated: { 4,5 }

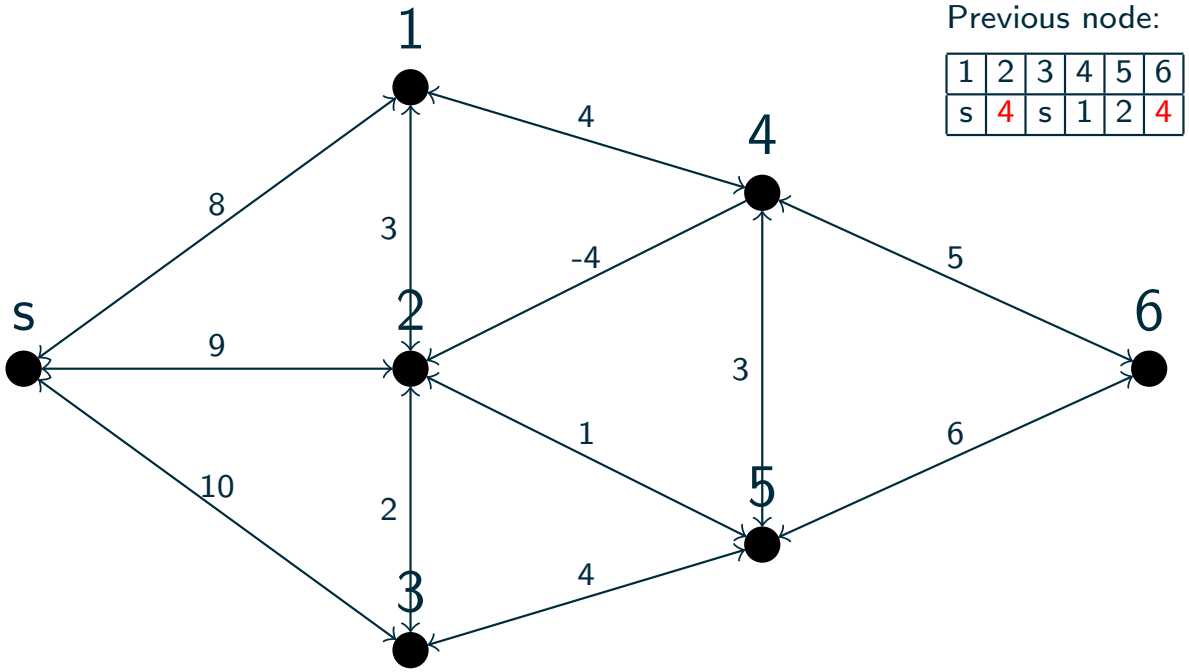
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	9	10	12	10	∞

Previous node:

1	2	3	4	5	6
s	s	s	1	2	-

Bellman-Ford algorithm



To be updated: { 4,5 }

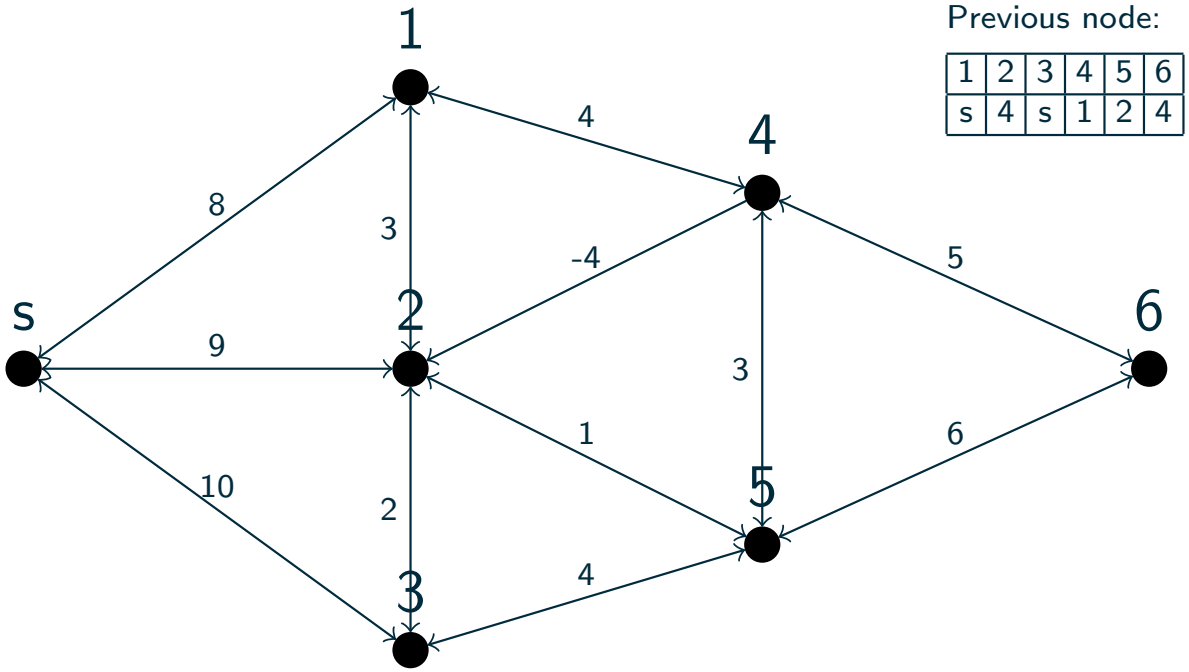
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	8	10	12	10	17

Previous node:

1	2	3	4	5	6
s	4	s	1	2	4

Bellman-Ford algorithm



To be updated: { 5,2,6 }

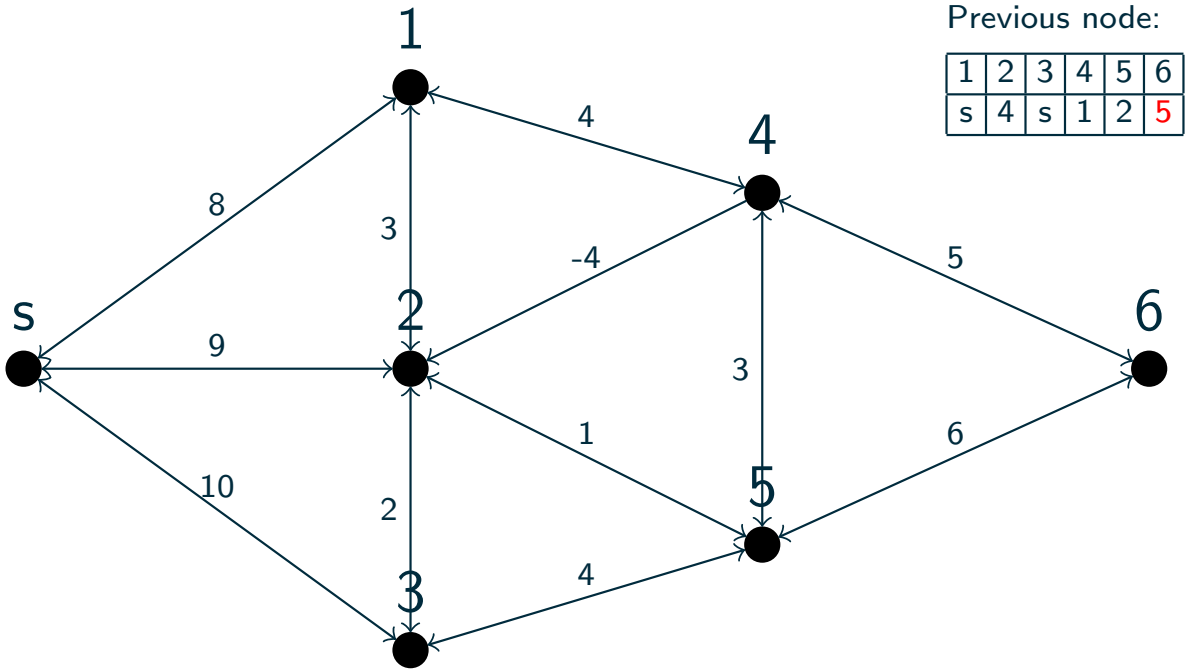
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	8	10	12	10	17

Previous node:

1	2	3	4	5	6
s	4	s	1	2	4

Bellman-Ford algorithm



To be updated: { 5,2,6 }

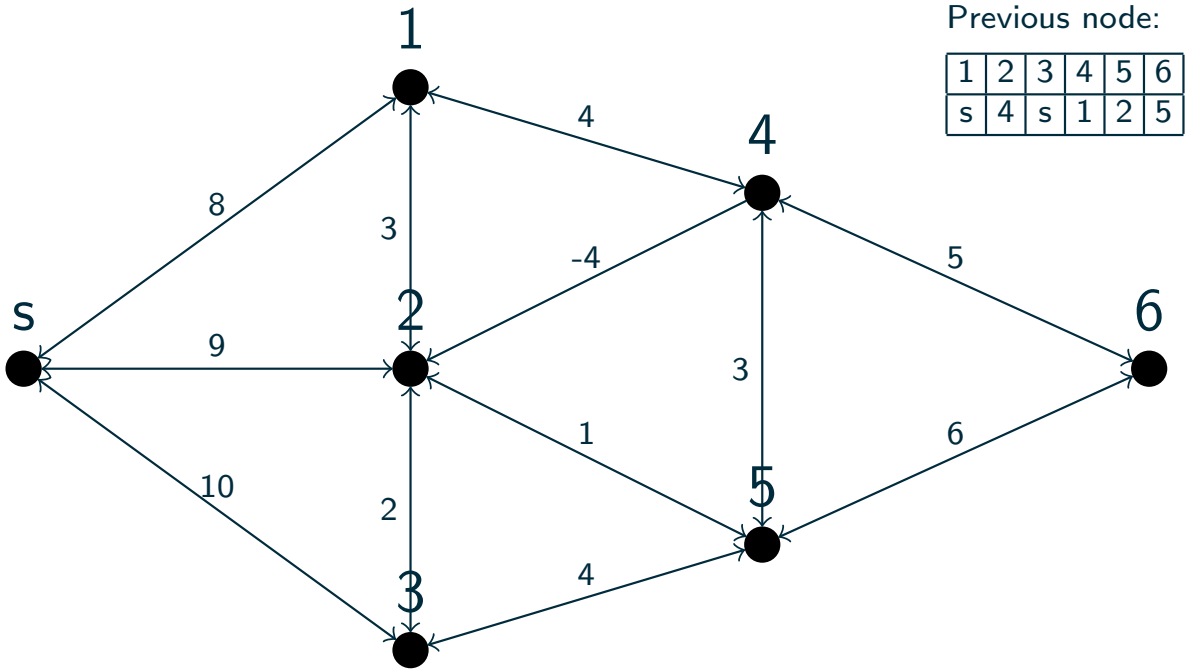
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	8	10	12	10	16

Previous node:

1	2	3	4	5	6
s	4	s	1	2	5

Bellman-Ford algorithm



To be updated: { 2,6 }

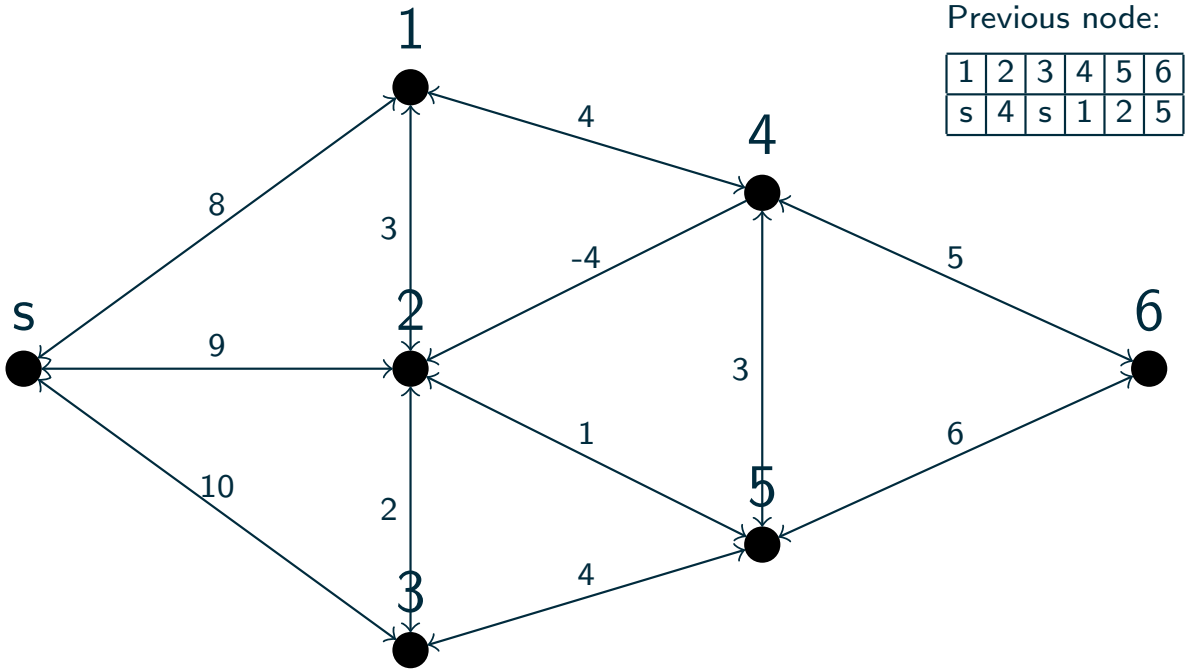
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	8	10	12	10	16

Previous node:

1	2	3	4	5	6
s	4	s	1	2	5

Bellman-Ford algorithm



To be updated: { 2,6 }

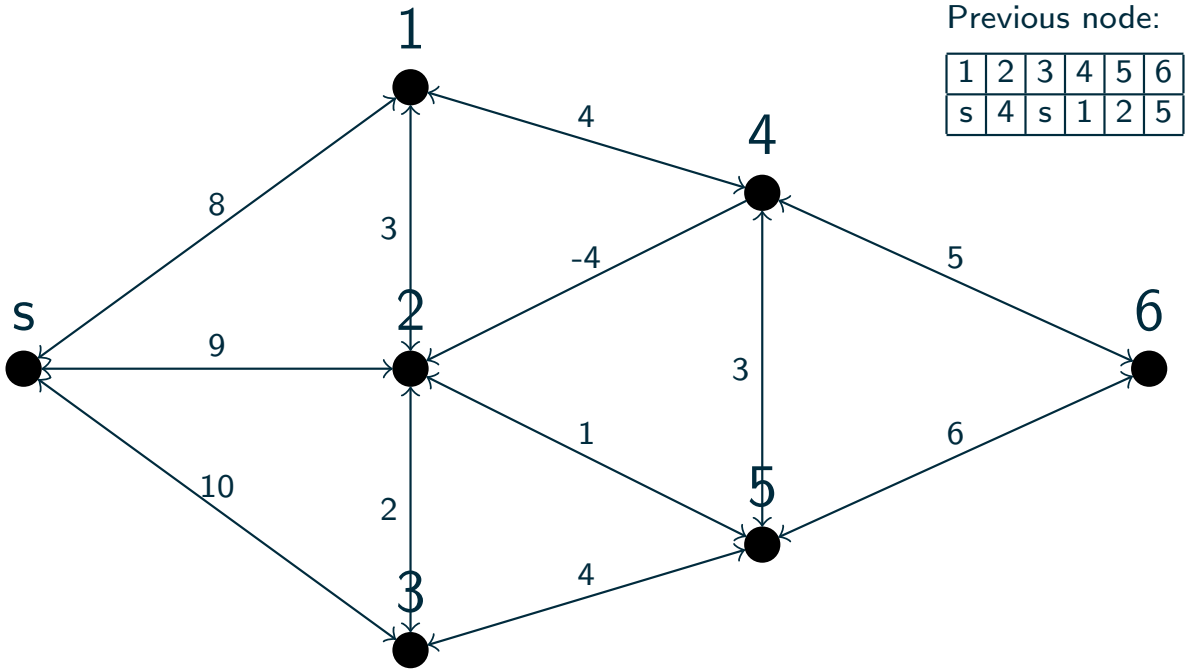
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	8	10	12	9	16

Previous node:

1	2	3	4	5	6
s	4	s	1	2	5

Bellman-Ford algorithm



To be updated: { 6,5 }

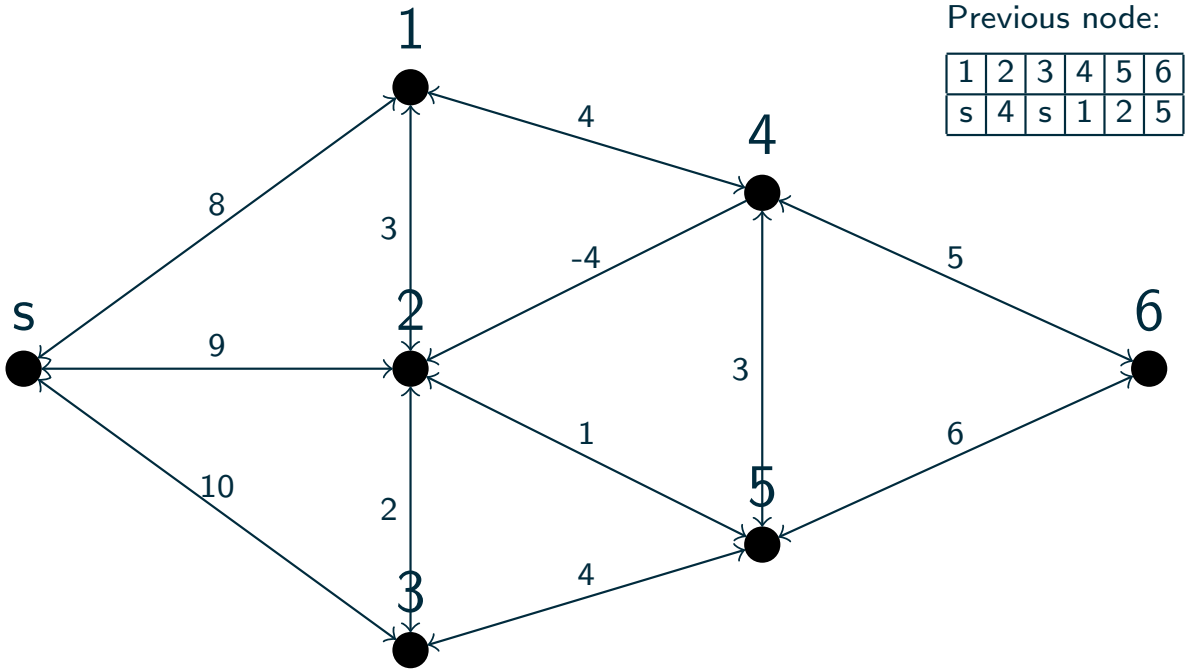
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	8	10	12	9	16

Previous node:

1	2	3	4	5	6
s	4	s	1	2	5

Bellman-Ford algorithm



To be updated: { 6,5 }

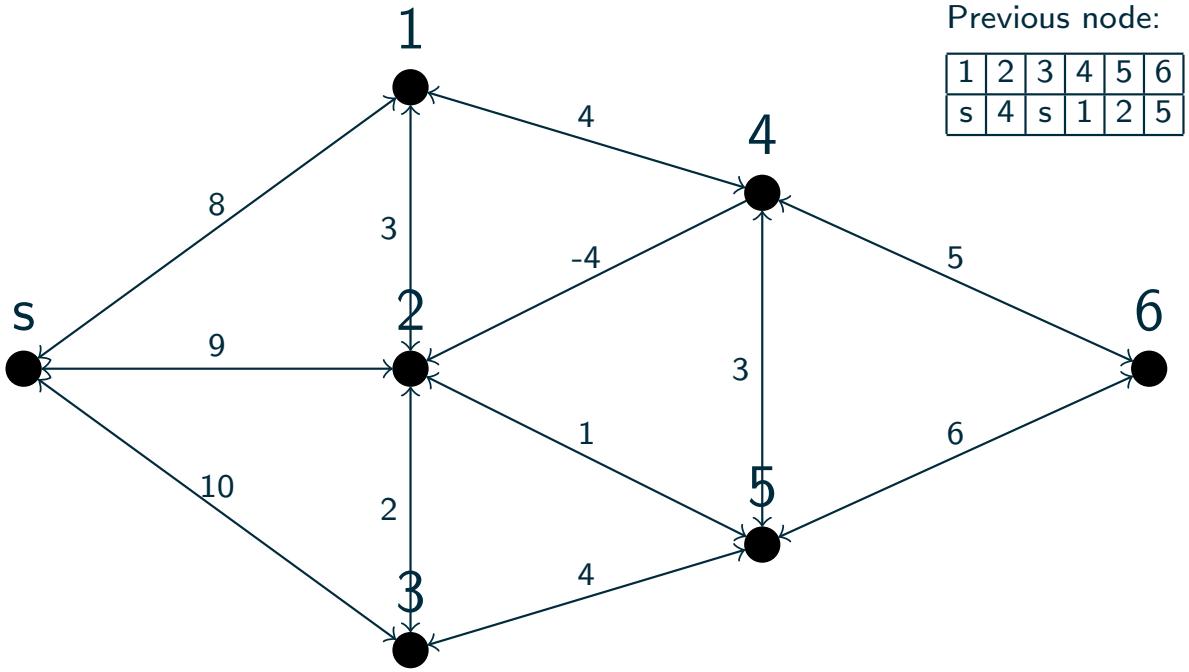
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	8	10	12	9	16

Previous node:

1	2	3	4	5	6
s	4	s	1	2	5

Bellman-Ford algorithm



To be updated: { 5 }

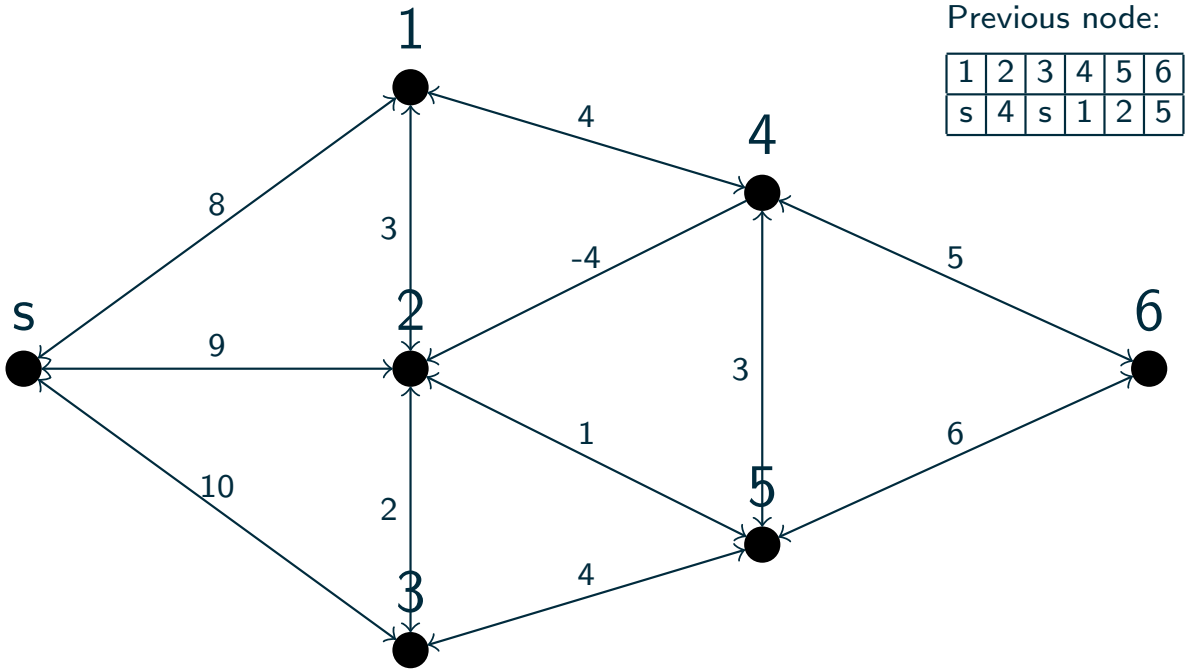
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	8	10	12	9	16

Previous node:

1	2	3	4	5	6
s	4	s	1	2	5

Bellman-Ford algorithm



To be updated: { 5 }

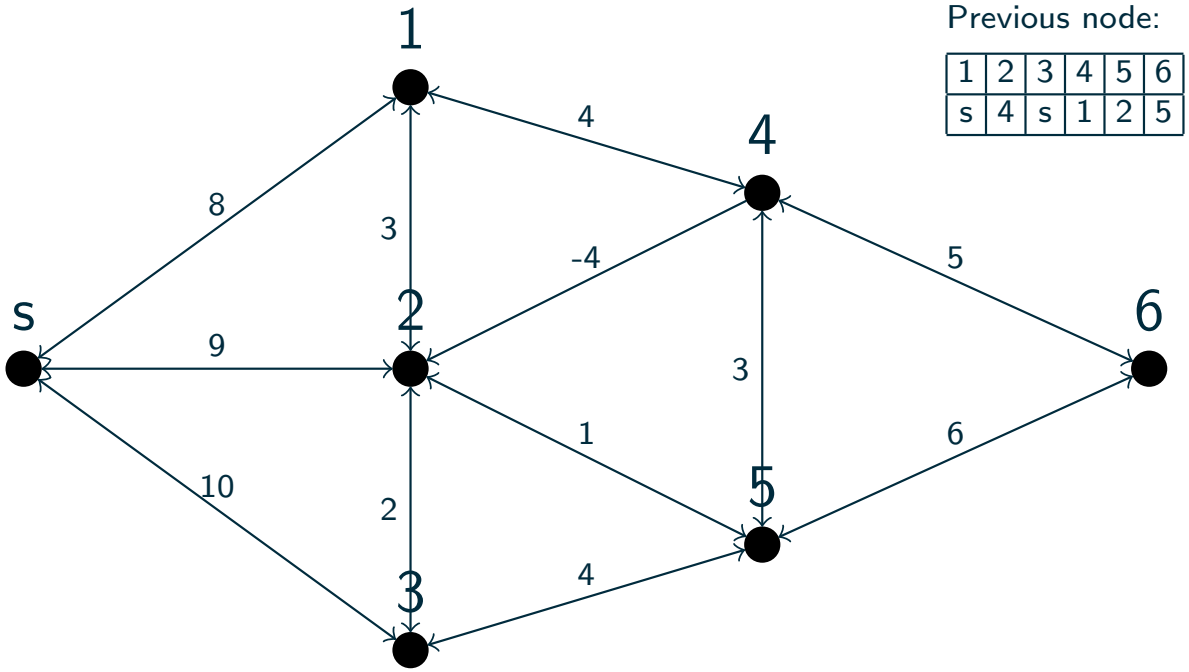
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	8	10	12	9	15

Previous node:

1	2	3	4	5	6
s	4	s	1	2	5

Bellman-Ford algorithm



To be updated: { 6 }

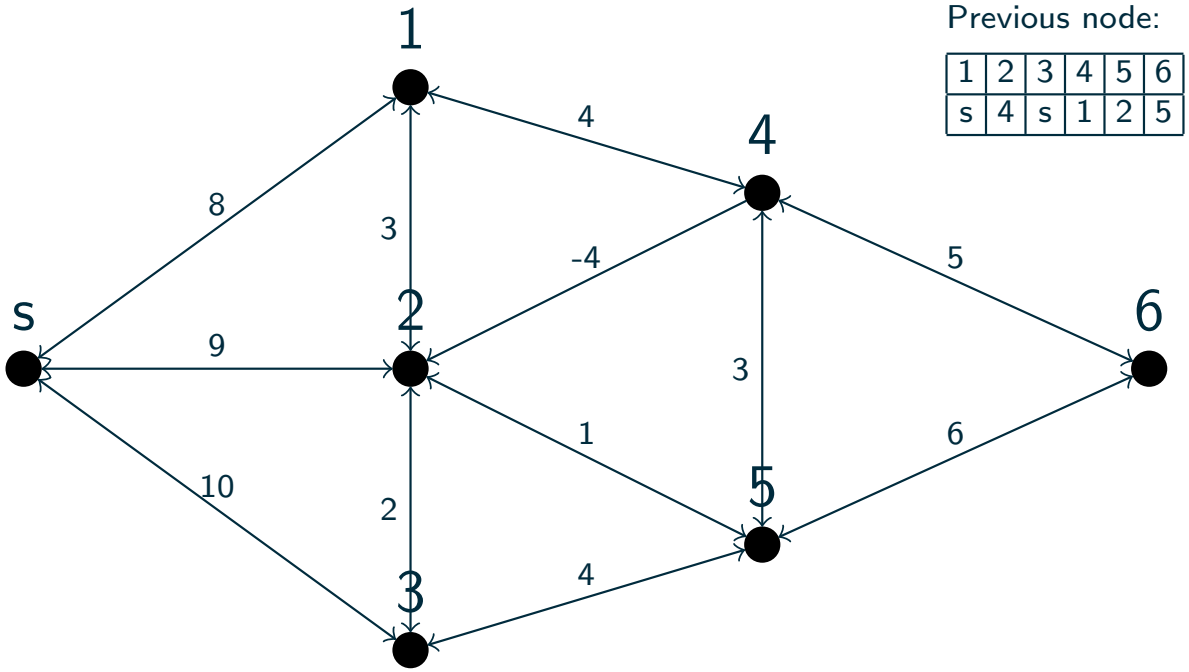
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	8	10	12	9	15

Previous node:

1	2	3	4	5	6
s	4	s	1	2	5

Bellman-Ford algorithm



To be updated: { 6 }

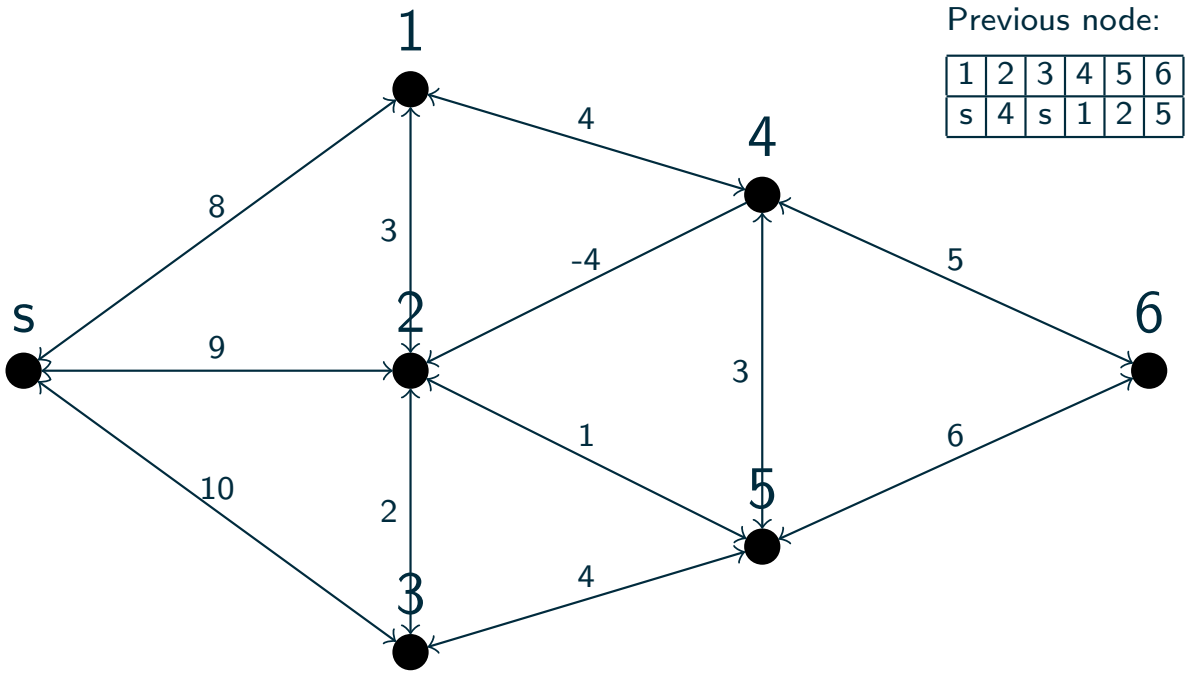
Lowest cost from s until node (so far):

1	2	3	4	5	6
8	8	10	12	9	15

Previous node:

1	2	3	4	5	6
s	4	s	1	2	5

Bellman-Ford algorithm



To be updated: { }

Lowest cost from s until node (so far):

1	2	3	4	5	6
8	8	10	12	9	15

Previous node:

1	2	3	4	5	6
s	4	s	1	2	5

Other

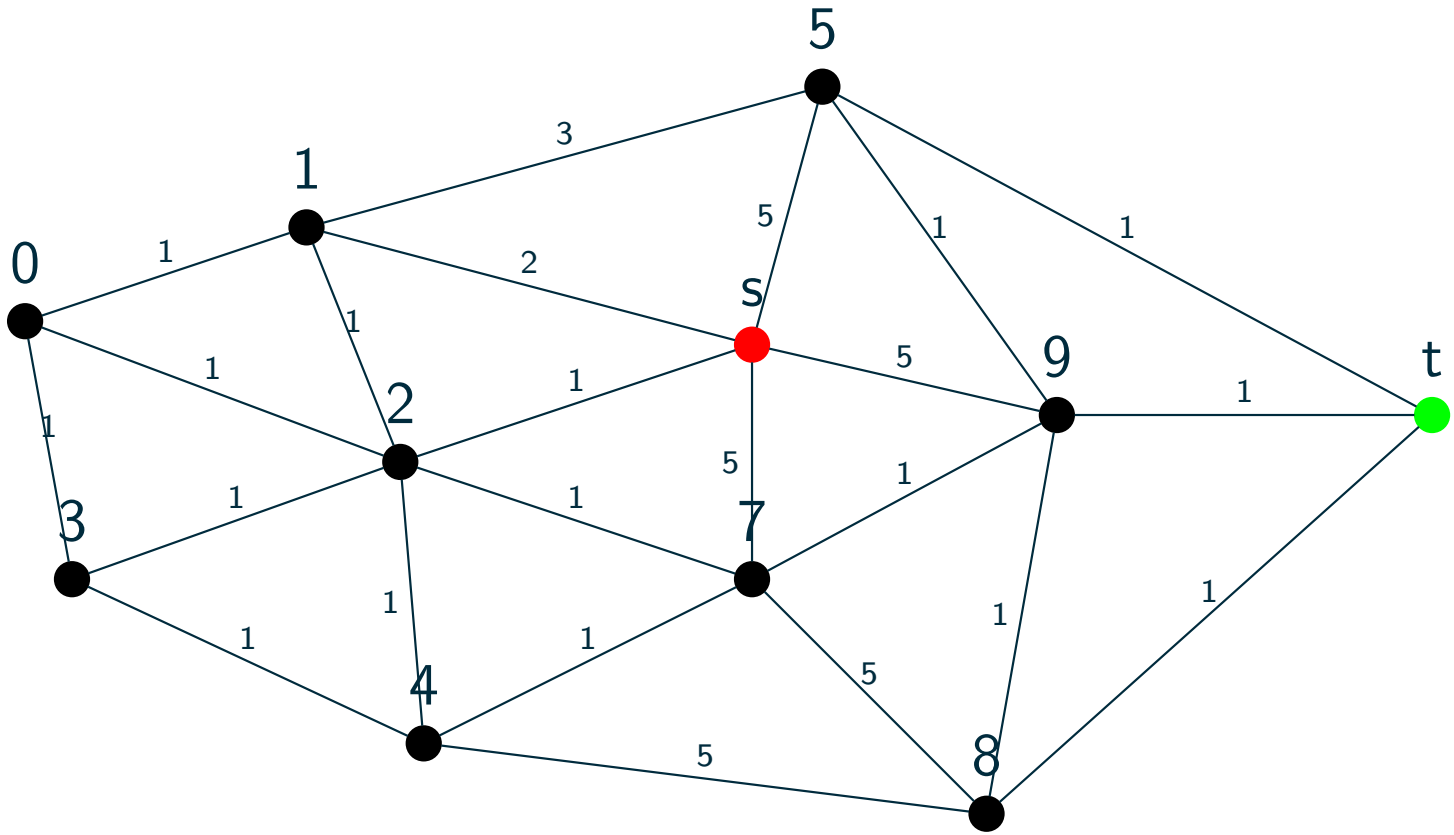
All-pairs shortest path (n to n):

- Floyd-Warshall algorithm (in $O(|V|^3)$)
  - incrementally updates matrix of shortest paths
  - function shortestPath(i,j,k) $\rightarrow$ returns shortest path from i to j using only vertices from set {1,2,...,k}
  - goal: find shortest path from each i to each j using only vertices 1 to k+1
  - shortestPath(i,j,k+1) =
    min(shortestPath(i,j,k),ShortestPath(i,k+1,k)+shortestPath(k+1,j,k))

- Johnson's algorithm (in $O(V^2 \log V + VE)$[17])
  1. add a node q to the graph, connected by a zero weight edge
  2. execute bellman-ford algorithm to find shortest path h(v) to each other vertex
  3. edges of the original graph are reweighted (edge with weight w(u,v) is now given weight w(u,v)+h(u)-h(v))
  4. remove q and apply Dijkstra's algorithm for each node

A* search algorithm

Dijkstra $\rightarrow$ explore node with lowest tentative cost, but what happens in the example below?



A* search algorithm

A* $\rightarrow$ uses a different evaluation function f(n) to determine good candidate nodes to explore.

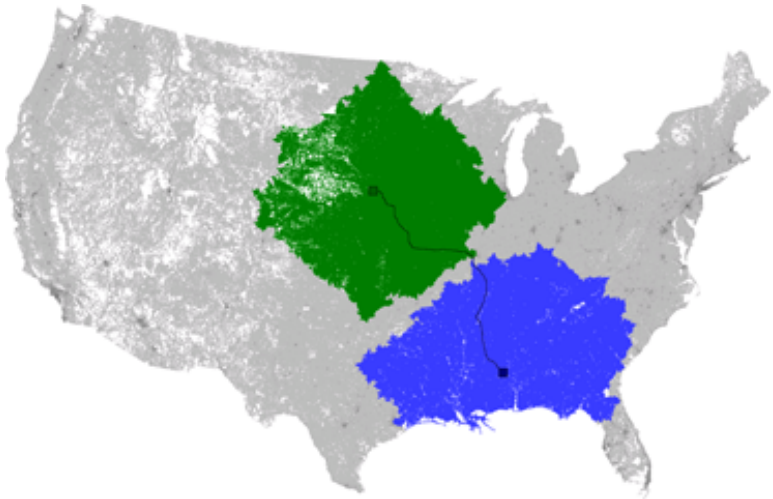
$$f(n) = g(n) + h(n)$$

where g(n) is the cost from the source node to node n
h(n) is the heuristic cost from node n to the target node

The heuristic cost must always underestimate the real distance to the target node. (e.g. "as the crow flies" (straight-line distance))

Dijkstra vs A*

Dijkstra (bidirectional)

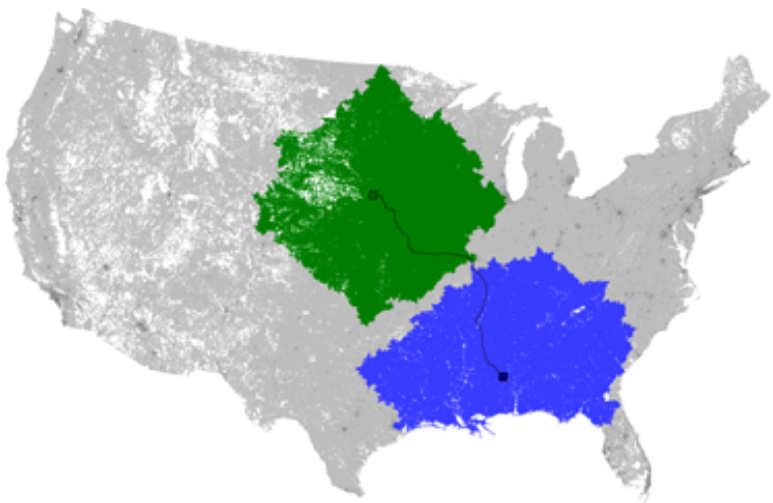


A* (with landmarks)

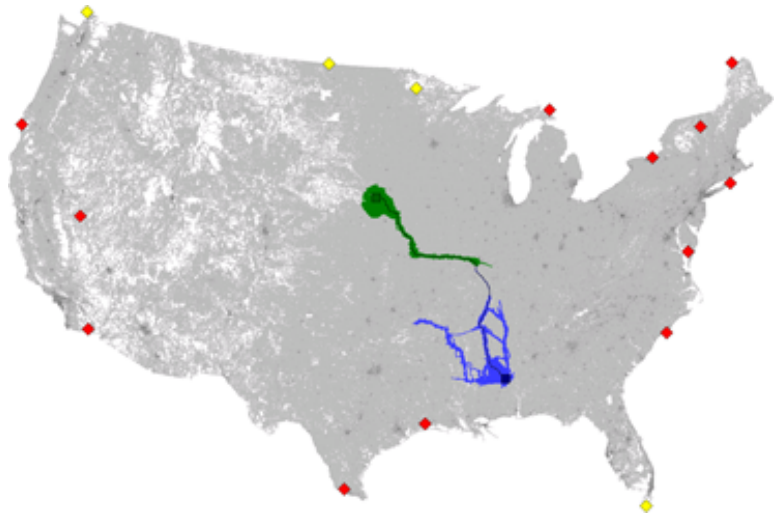
source: http://research.microsoft.com/en-us/news/features/shortestpath-070709.aspx

Dijkstra vs A*

Dijkstra (bidirectional)



A* (with landmarks)



source: http://research.microsoft.com/en-us/news/features/shortestpath-070709.aspx

Further improvement

Use of preprocessing:

- Landmarks [10][12][13][14][19]
- Highway-hierarchy $\rightarrow$ takes advantage of inherent road hierarchy to restrict the search to a smaller subgraph [18]
- Reach-based pruning $\rightarrow$ pruning the shortest path search[11][15]

References I

[1] A* search algorithm @ONLINE.
visited on: may 27 2013.
http://en.wikipedia.org/wiki/A*_search_algorithm.

[2] Bellmanford algorithm @ONLINE.
visited on: may 27 2013.
http://en.wikipedia.org/wiki/Bellman-Ford_algorithm.

[3] Dijkstra's algorithm @ONLINE.
visited on: may 27 2013.
http://en.wikipedia.org/wiki/Dijkstra's_algorithm.

[4] Floydwarshall algorithm @ONLINE.
visited on: may 27 2013.
http://en.wikipedia.org/wiki/Floyd-Warshall_algorithm.

[5] Johnson's algorithm @ONLINE.
visited on: may 27 2013.
http://en.wikipedia.org/wiki/Johnson's_algorithm.

[6] Shortest-path problem @ONLINE.
visited on: may 27 2013.
http://en.wikipedia.org/wiki/Shortest_path_problem.

[7] Richard Bellman.
On a routing problem.
Technical report, DTIC Document, 1956.

[8] R Chandrasekaran and UT Dallas.
Network flows and combinatorial optimization.
1996.

References II

[9] Edsger W Dijkstra.
A note on two problems in connexion with graphs.
Numerische mathematik, 1(1):269–271, 1959.

[10] Andrew V Goldberg and Chris Harrelson.
Computing the shortest path: A* search meets graph theory.
In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 156–165. Society for Industrial and Applied Mathematics, 2005.

[11] Andrew V Goldberg, Haim Kaplan, and Renato Werneck.
Reach for a*: Efficient point-to-point shortest path algorithms.
In *Workshop on Algorithm Engineering & Experiments, Miami*, pages 129–143, 2006.

[12] Andrew V Goldberg, Haim Kaplan, and Renato F Werneck.
Better landmarks within reach.
In *Experimental Algorithms*, pages 38–51. Springer, 2007.

[13] Andrew V Goldberg and Renato F Werneck.
Computing point-to-point shortest paths from external memory.
In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX05)*, pages 26–40, 2005.

[14] Kevin Grant and David Mould.
Lpi: Approximating shortest paths using landmarks.
In *Eighteenth European Conference on Artificial Intelligence-Workshop on AI and Games*, 2008.

[15] Ron Gutman.
Reach-based routing: A new approach to shortest path algorithms optimized for road networks.
In *6th Workshop on Algorithm Engineering and Experiments*, pages 100–111, 2004.

References III

[16] Peter E Hart, Nils J Nilsson, and Bertram Raphael.
A formal basis for the heuristic determination of minimum cost paths.
Systems Science and Cybernetics, IEEE Transactions on, 4(2):100–107, 1968.

[17] Donald B Johnson.
Efficient algorithms for shortest paths in sparse networks.
Journal of the ACM (JACM), 24(1):1–13, 1977.

[18] Peter Sanders and Dominik Schultes.
Highway hierarchies hasten exact shortest path queries.
In *Algorithms–Esa 2005*, pages 568–579. Springer, 2005.

[19] Konstantin Tretyakov, Abel Armas-Cervantes, Luciano García-Bañuelos, Jaak Vilo, and Marlon Dumas.
Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs.
In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1785–1794. ACM, 2011.

[20] Wayne L Winston.
Operations research: Applications and algorithms.

University of Antwerp
Operations Research Group
ANT/OR

Shortest path problem and algorithms

Jochen Janssens

June 3, 2013, ANT/OR fORum, Antwerp

jochen.janssens@ua.ac.be