EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
SCIENCE
Applied Bioinformatics

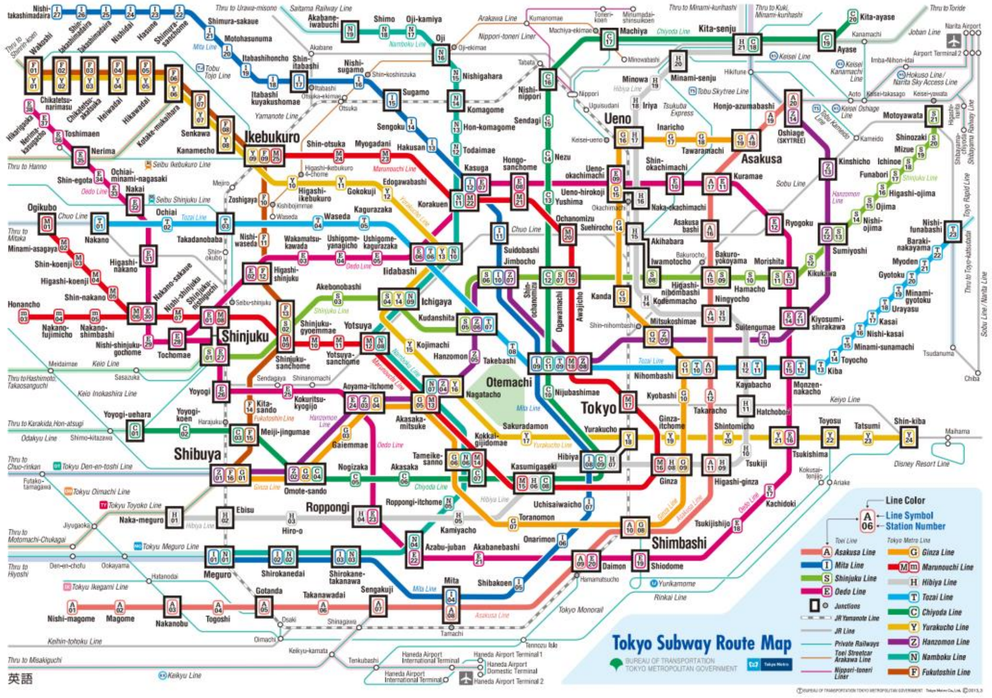
Programming in C/C++

- Graphs (2) – Shortest Paths -

Shortest Path

- Often, we don't only want to just find a path between two nodes, but the we want to find the **shortest** path between two nodes $\mathbf{v}_1, \mathbf{v}_2$ in a graph

- Shortest path - path with smallest path weight
  - for unweighted graphs: number of edges in the path
  - for weighted graphs: sum of edge weights in the path



Dijkstra’s Algorithm

Dijkstra's Algorithm

Basic idea:

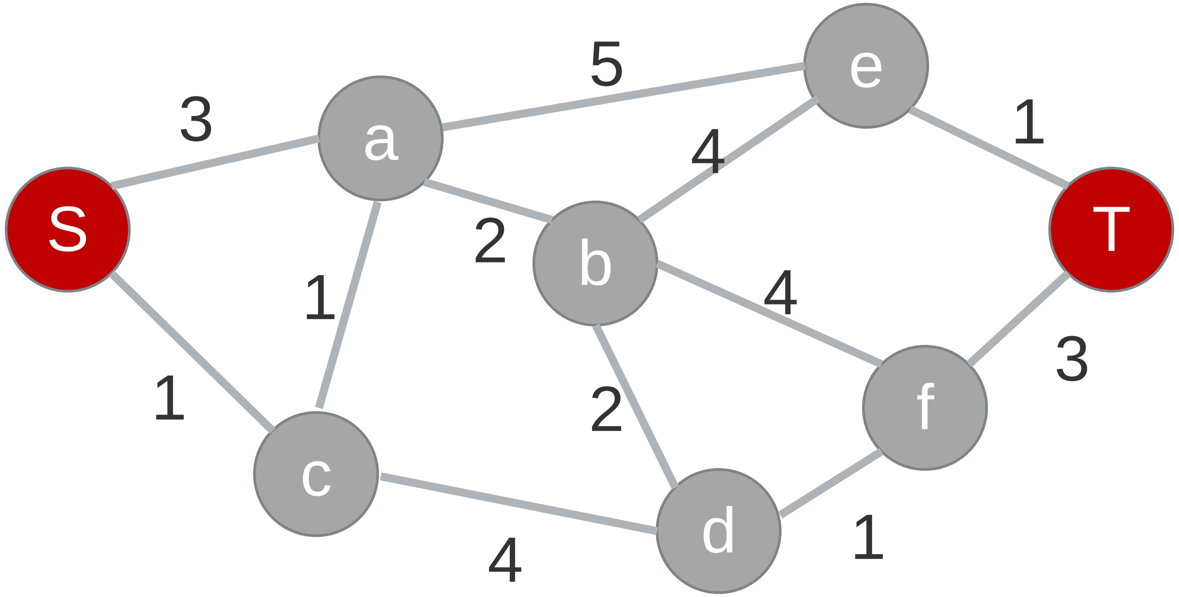
- **Extend the current shortest path** starting a source and ending at node `k` by all its neighbors `n(k)`.
- The path weight for the neighboring node is the cost to `k` plus the weight for the edge to the neighbor `e(k,n(k))`
- For each neighbor `n(k)`, test whether the new link is shorter than the previous one

- Once destination node is reached output the path

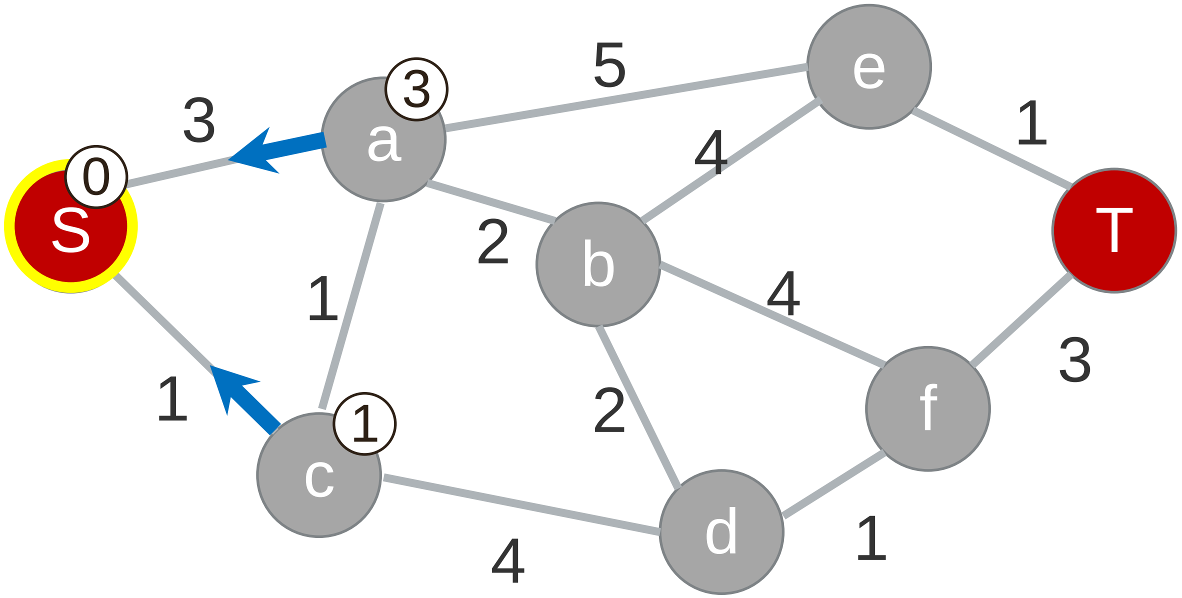
- Required data structures:
  - Remember the best predecessor to trace back the best solution from end to source
  - Priority queue to find the node with currently shortest path to source node

Dijkstra’s Algorithm

- Weighted Graph

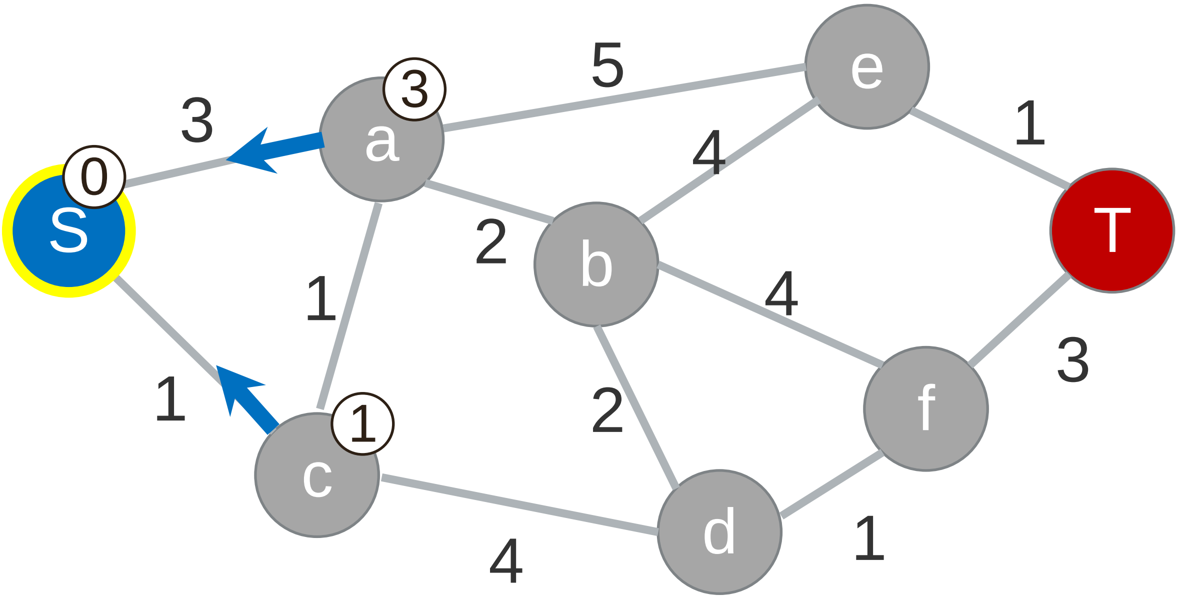


Dijkstra's Algorithm



- Begin with S
- Update: shortest path to neighbors c and a have distance < ∞ -> store new distances and remember the predecessor

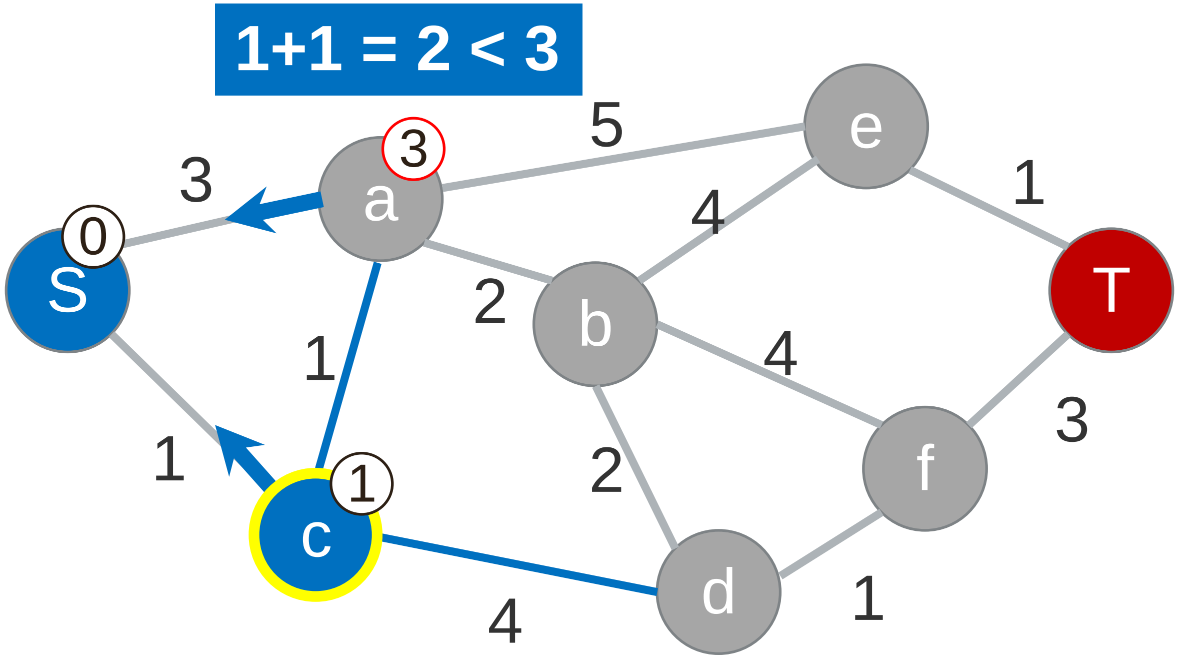
Dijkstra’s Algorithm



- S already processed
- Add children a and c to priority queue to determine **next best** node

```
Queue:   (c,1) (a,3)
```

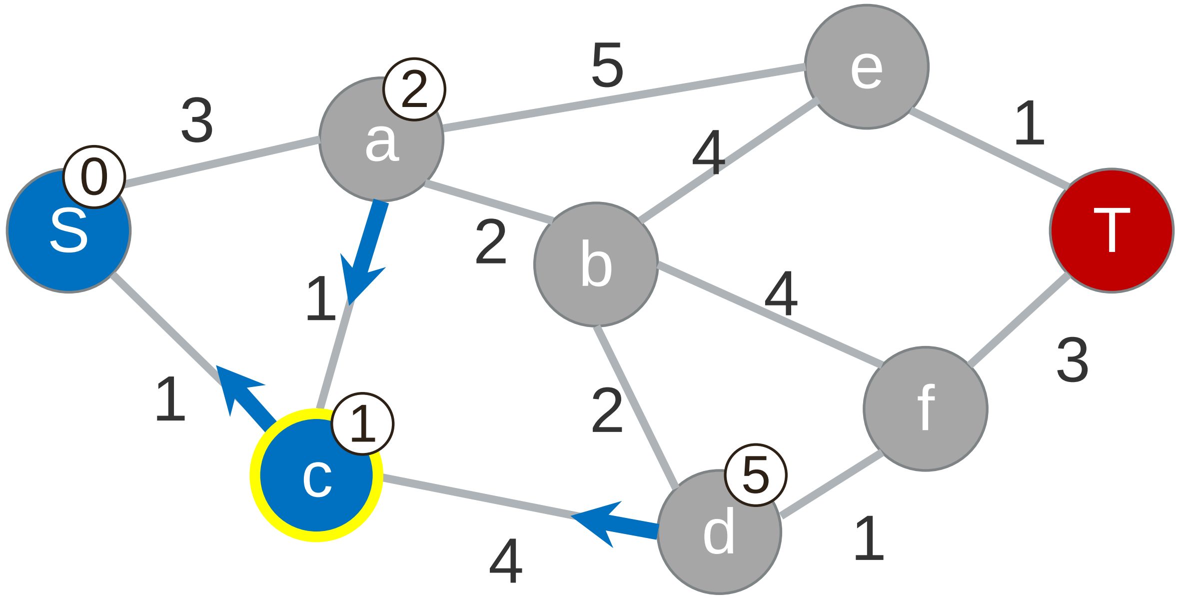
Dijkstra’s Algorithm



- Process c
- Extend path to all neighbor nodes

Queue: (a,3)

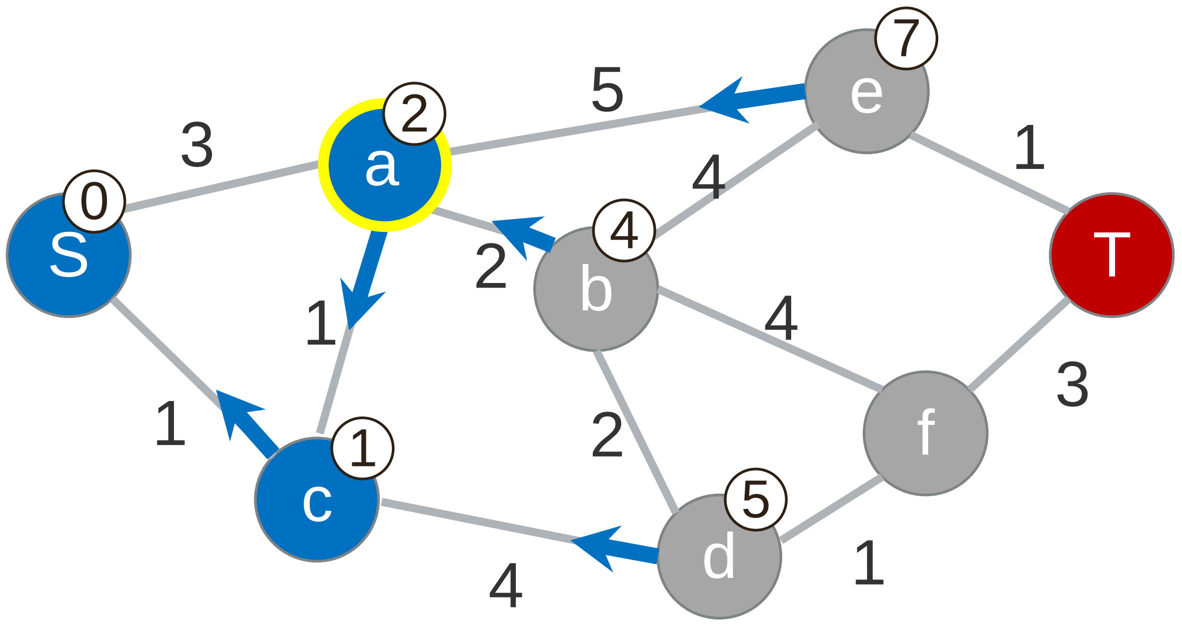
Dijkstra's Algorithm



- Process c
- Extend path to all neighbor nodes – store new distances
- **update predecessors**
- **update priority queue**

```
Queue: (a,2) (d,5)
```

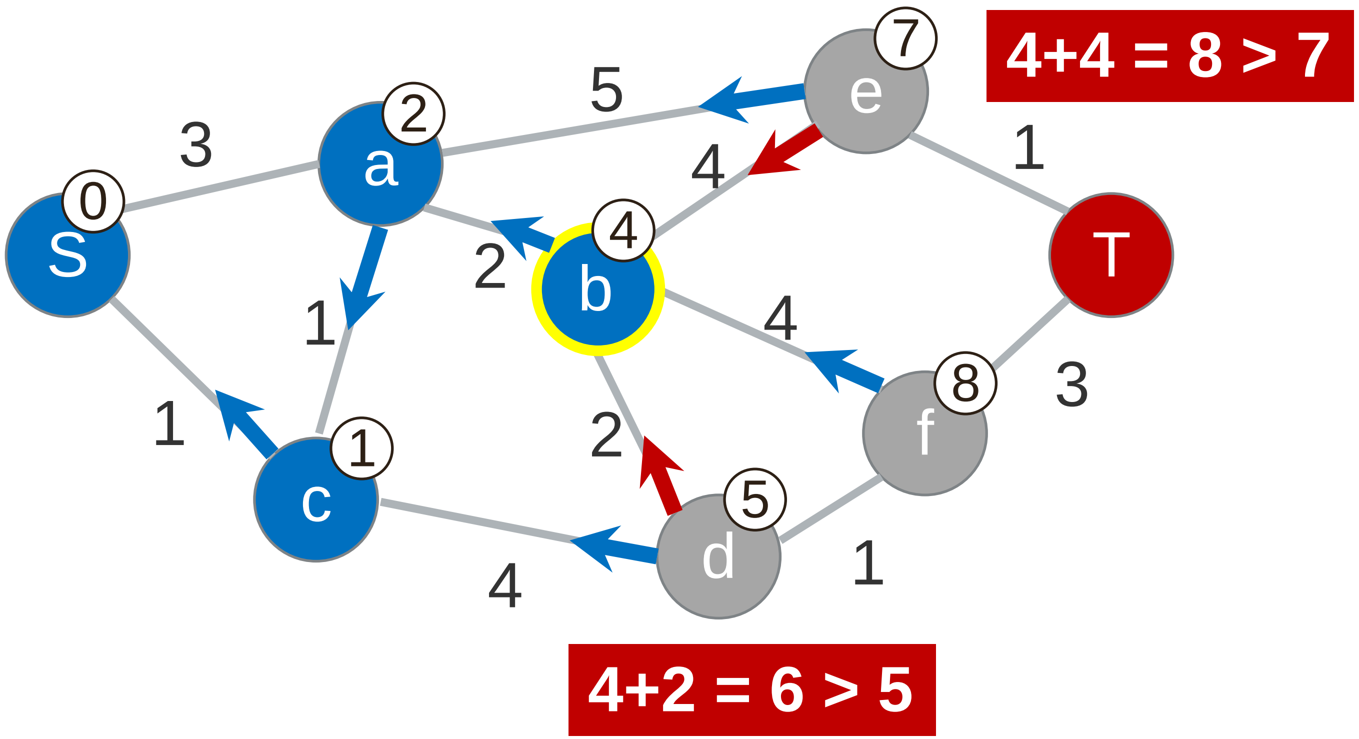
Dijkstra’s Algorithm



- Process a
- Extend path to all neighbor nodes – store new distances
- **update predecessors**
- **update priority queue**

`Queue: (b,4) (d,5) (e,7)`

Dijkstra’s Algorithm

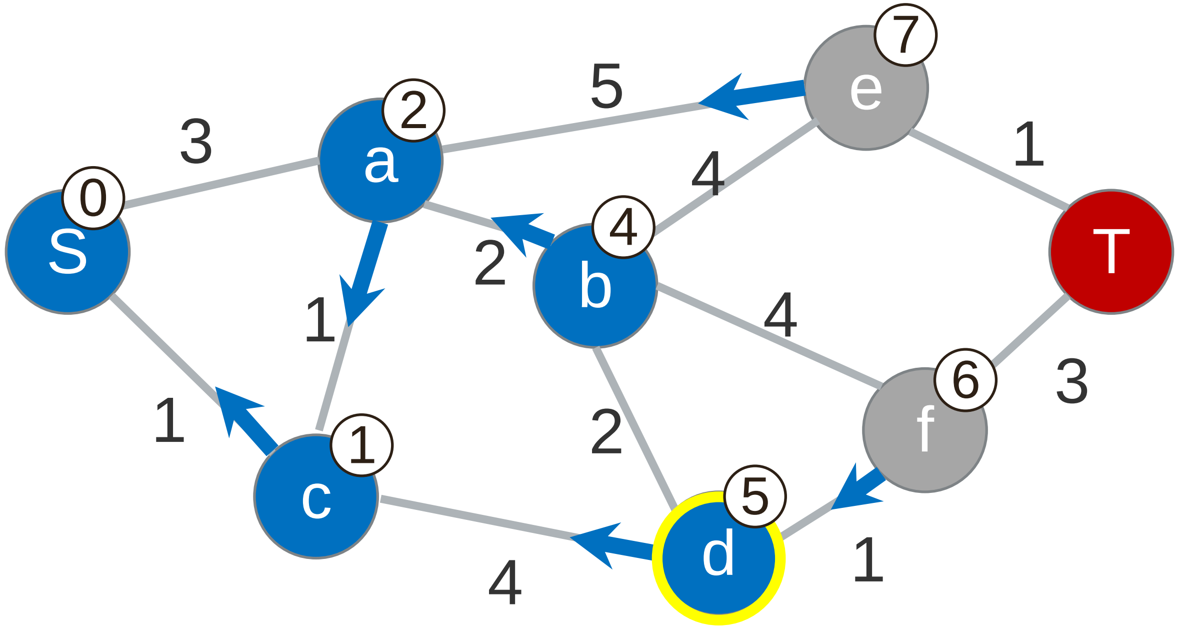


- Process b

- Extend path to all neighbor nodes – store new distances
- (No better path to e or d, no update)

`Queue: (d,5)(e,7)(f,8)`

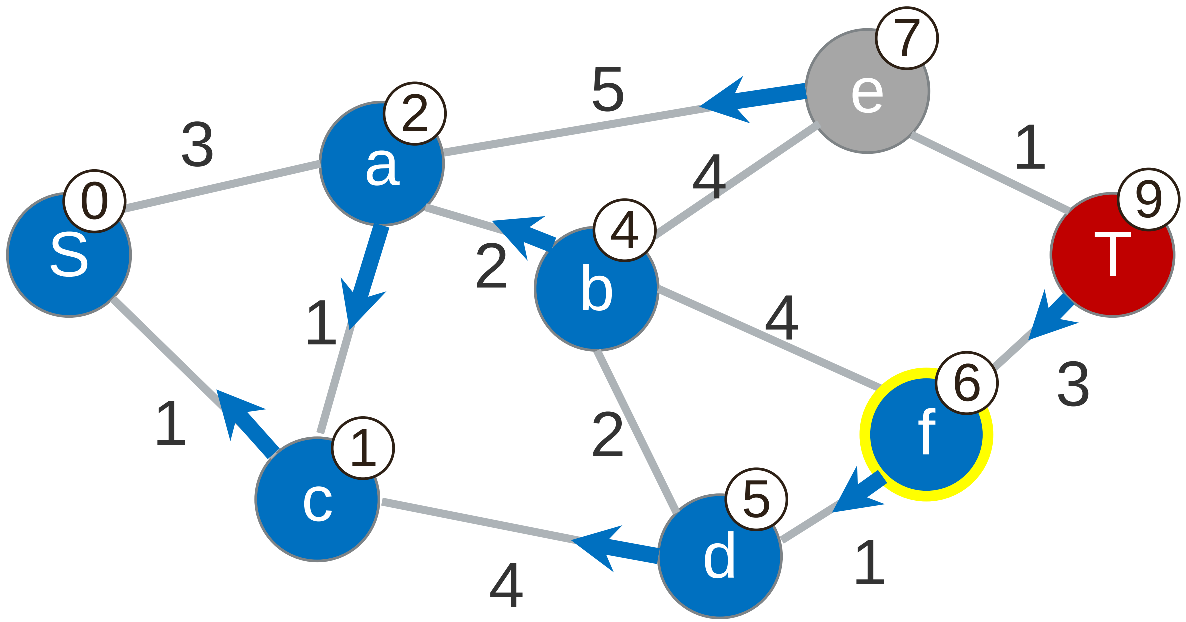
Dijkstra’s Algorithm



- Process d
- Extend path to all neighbor nodes – update if a cheaper path is found
- Update priority queue

```
Queue:(f,6)(e,7)
```

Dijkstra's Algorithm

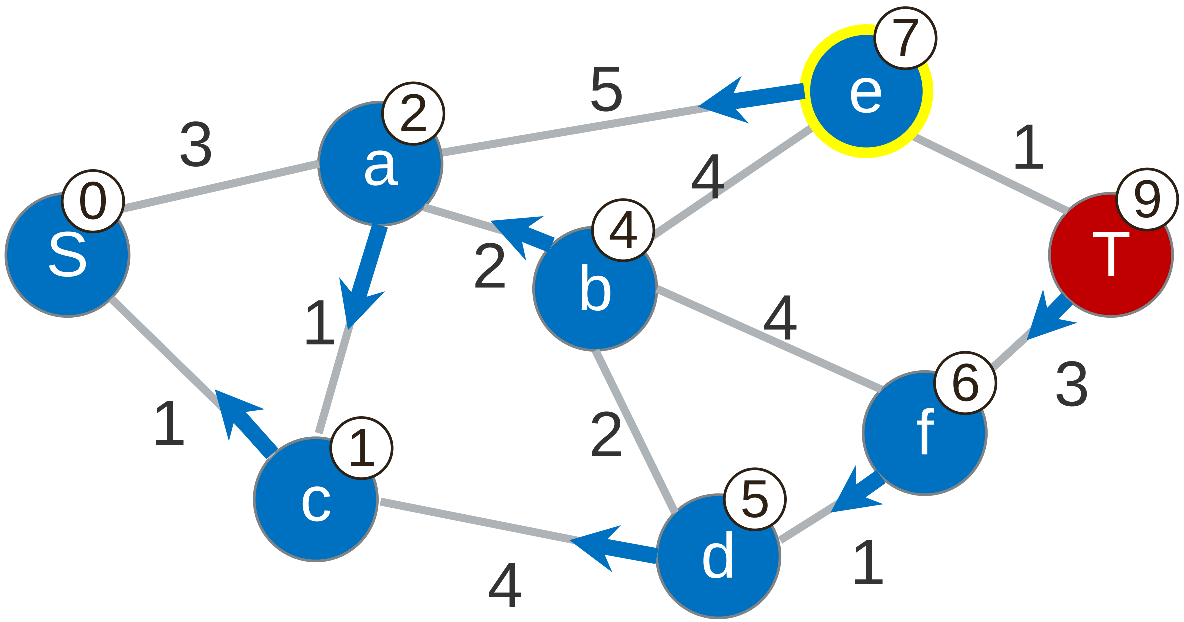


- Process d
- Extend path to all neighbor nodes – update if a cheaper path is found
- Update priority queue

`Queue: (e,7) (T,9)`

Dijkstra’s Algorithm

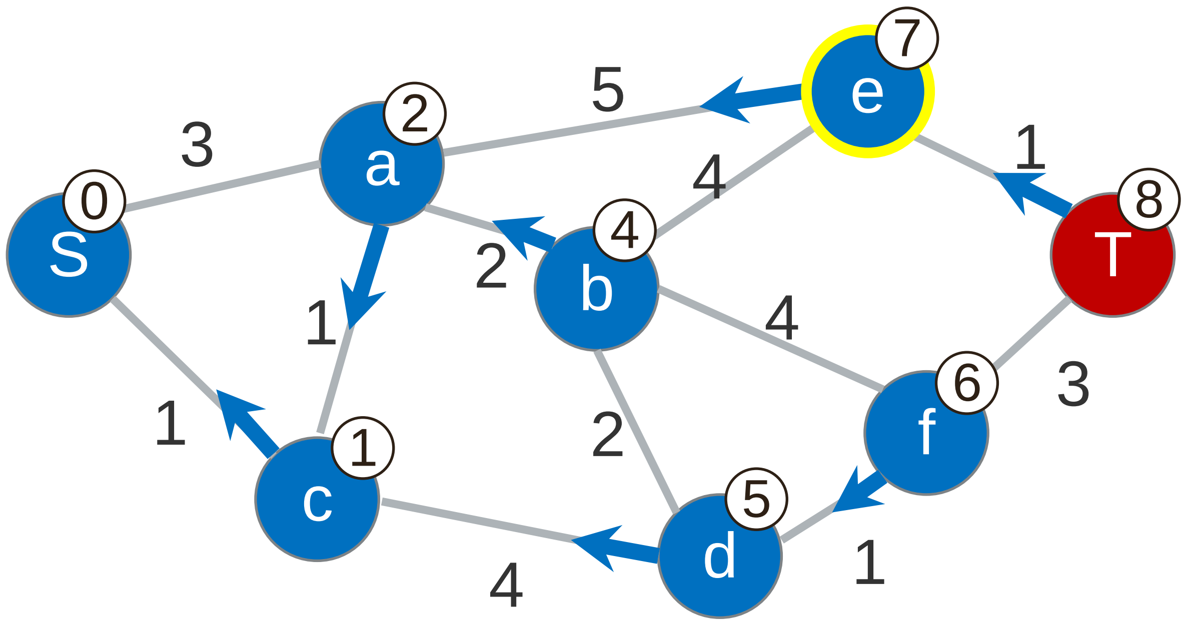
EBERHARD KARLS UNIVERSITÄT TÜBINGEN



- Process e
- Extend path to all neighbor nodes – update if a cheaper path is found
- Update priority queue

```
Queue: (T,9)
```

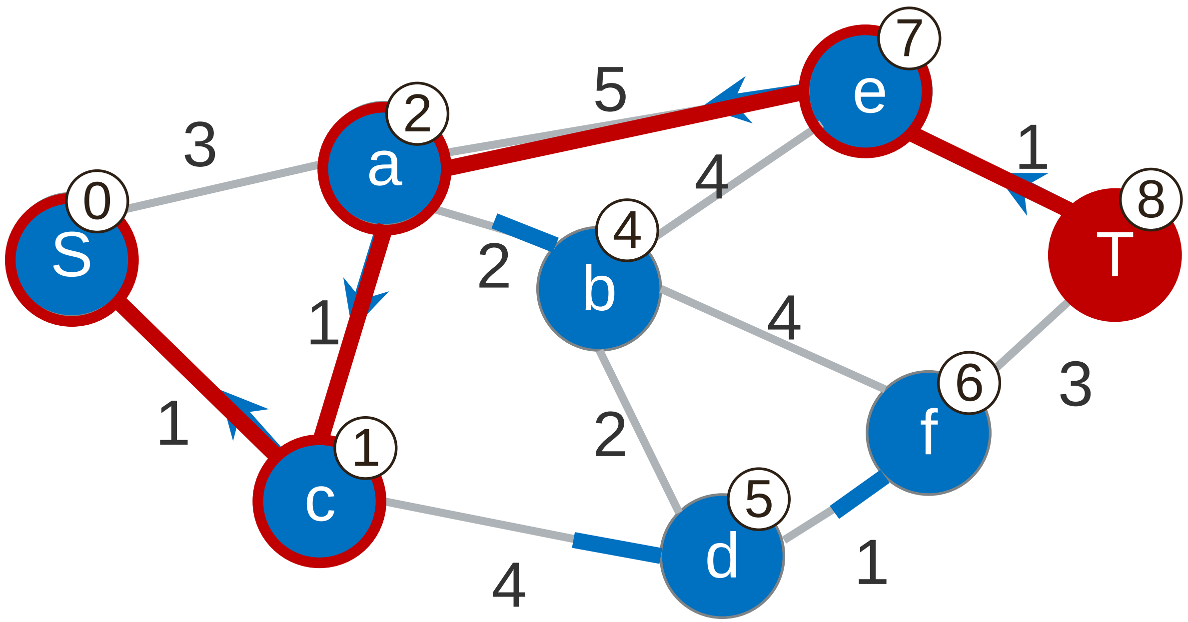
Dijkstra's Algorithm



- Process e
- Extend path to all neighbor nodes – update predecessor if a cheaper path is found
- Update priority queue

```
Queue: (T,8)
```

Dijkstra's Algorithm



- Process T: Reached goal!
- Shortest path: unroll from T
- Queue is empty

Queue:

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:                 // Initializations
        dist[v] := infinity ;                   // Unknown distance function from
                                                // source to v
        previous[v] := undefined ;              // Previous node in optimal path
    end for                                     // from source

    dist[source] := 0 ;                         // Distance from source to source
    Q := the set of all nodes in Graph ;        // All nodes in the graph are
                                                // unoptimized - thus are in Q
    while Q is not empty:                       // The main loop
    u := vertex in Q with smallest distance in dist[] ;
                        // Source node in first case
        remove u from Q ;
        if dist[u] = infinity:
            break ;                             // all remaining vertices are
        end if                                  // inaccessible from source

        for each neighbor v of u:               // where v has not yet been
                                                // removed from Q.
            alt := dist[u] + dist_between(u, v) ;
            if alt < dist[v]:                   // Relax (u,v,a)
                dist[v] := alt ;
                previous[v] := u ;
                decrease-key v in Q;            // Reorder v in the Queue
            end if
        end for
    end while
    return dist;
endfunction
```

[source: wikipedia]

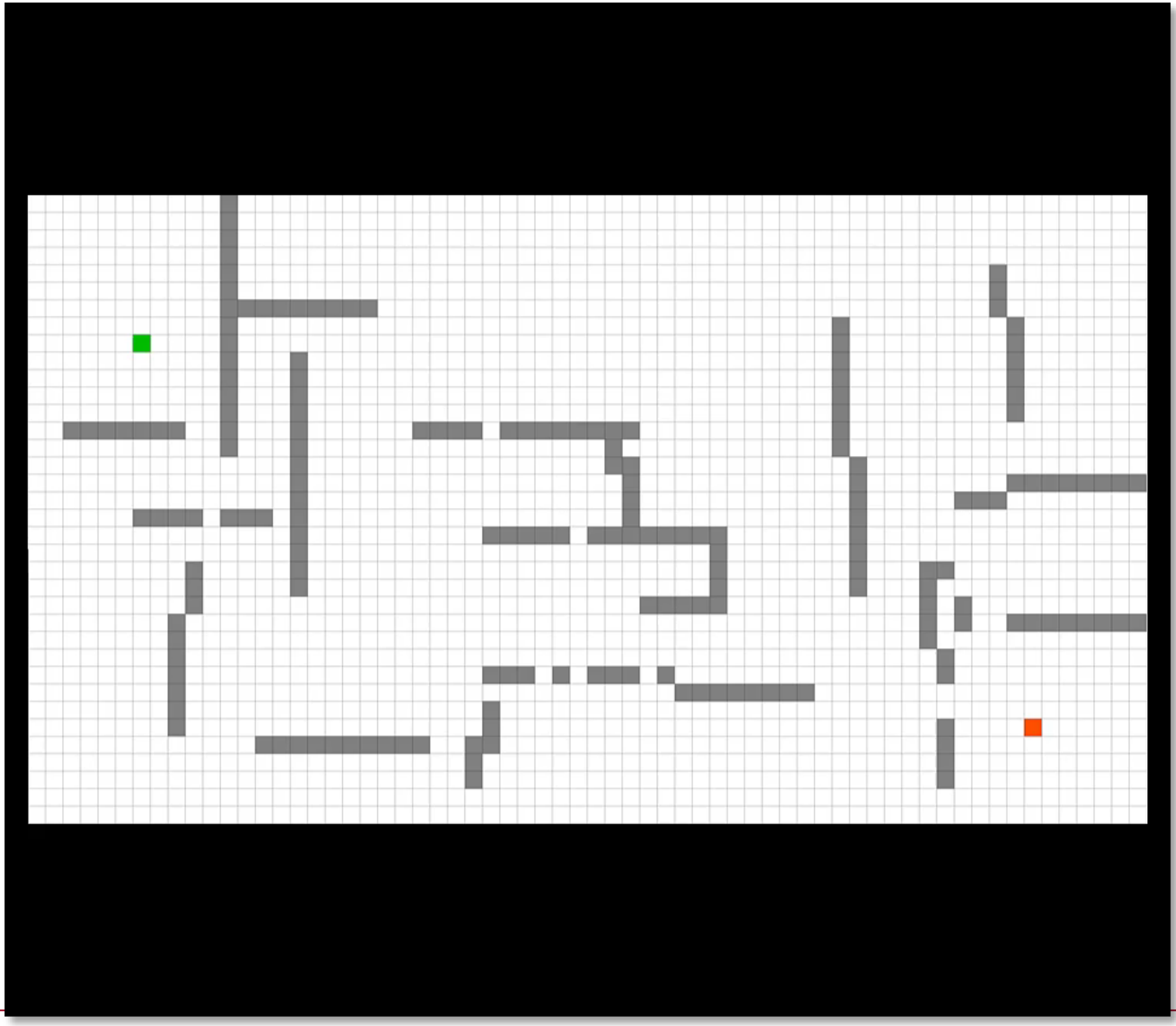
Observation

The currently determined shortest subpath never becomes shorter by later processing of additional nodes

- Already visited nodes do not need to be looked at again

Priority queue - different implementations possible:

- Insertion of all nodes vs. insertion of just reachable nodes
- Heap vs. set (elements must also be changed)



Dijkstra using the BGL

- Boost provides free peer-reviewed portable C++ source libraries
- Many of the libraries later became part of the STL
- The BGL is a (not so modern anymore) library for Graph algorithms
- Most of the code is used to configure vertices and edges

```
// define vertex property to be a name and a color
using VertexPropertyType = property<vertex_name_t, std::string, property<vertex_color_t, default_color_type>>;

// define edge property to contain an int weight and a color
using EdgePropertyType = property<edge_weight_t, int, property<edge_color_t, default_color_type>>;

// graph will be an adjacency list with directed edges and our vertex and edge properties
using DirectedGraphType = adjacency_list<vecS, vecS, directedS, VertexPropertyType, EdgePropertyType>;

// vertex manipulation happens through handler
using VertexDescriptor = graph_traits<DirectedGraphType>::vertex_descriptor;

DirectedGraphType g;
VertexDescriptor a = boost::add_vertex(VertexPropertyType("a", white_color), g);
VertexDescriptor b = boost::add_vertex(VertexPropertyType("b", white_color), g);
VertexDescriptor c = boost::add_vertex(VertexPropertyType("c", white_color), g);
VertexDescriptor d = boost::add_vertex(VertexPropertyType("d", white_color), g);
VertexDescriptor e = boost::add_vertex(VertexPropertyType("e", white_color), g);
boost::add_edge(a, c, EdgePropertyType(1, black_color), g); add_edge(b, d, EdgePropertyType(1, black_color), g);
boost::add_edge(b, e, EdgePropertyType(2, black_color), g); add_edge(c, b, EdgePropertyType(5, black_color), g);
boost::add_edge(c, d, EdgePropertyType(10, black_color), g); add_edge(d, e, EdgePropertyType(4, black_color), g);
boost::add_edge(e, a, EdgePropertyType(3, black_color), g); add_edge(e, b, EdgePropertyType(7, black_color), g);
```

Dijkstra using the BGL

```
std::vector<int> distances(boost::num_vertices(g));      // Output for distances to each node
std::vector<VertexDescriptor> predMap(boost::num_vertices(g)); // Output for predecessors of each node in the shortest path tree result

// create predecessor/distance map
auto distanceMap = boost::predecessor_map(
 boost::make_iterator_property_map(predMap.begin(), boost::get(boost::vertex_index, g))).distance_map(
 boost::make_iterator_property_map(distances.begin(), boost::get(boost::vertex_index, g)));

// set start node
VertexDescriptor sourceV = a;

// run dijkstra
boost::dijkstra_shortest_paths(g, sourceV, distanceMap);

// set end node
VertexDescriptor destinationV = e;

int totalCost = distances[destinationV]; // read total path length

// trace path from end vertex to source vertex
std::vector<VertexDescriptor> path;
VertexDescriptor current = destinationV;
while (sourceV != current)
{
 path.push_back(current);
 current = predMap[current];
}
path.push_back(sourceV); // add source as last element to path

for (auto rit = path.rbegin(); rit != path.rend(); ++rit) // print path from source to destination (iterate using rbegin())
 std::cout << *rit << " -> ";
```

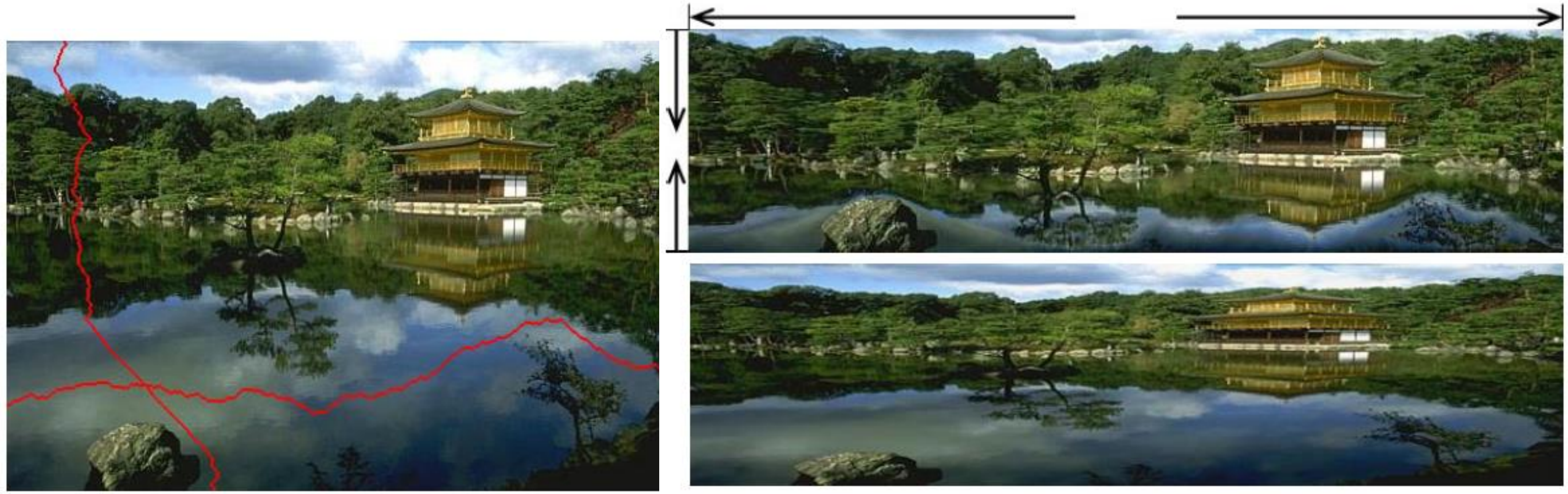
Dynamic Programming

Shortest paths on images

Seam Carving for Content-Aware Image Resizing

- [Shai Avidan, Ariel Shamir, ACM SIGGRAPH 2007]

input

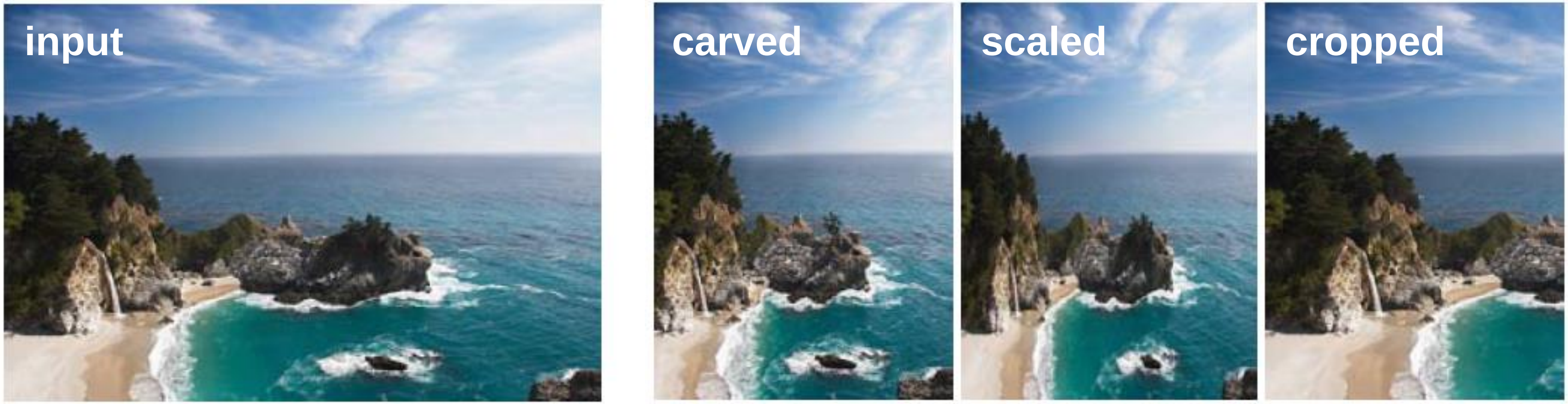


carved

scaled

- Goal: Change the size or aspect ratio of the image without distorting the content
- Repeatedly find the shortest path through the image

Seam Carving for Content-Aware Image Resizing



[Shai Avidan, Ariel Shamir, ACM SIGGRAPH 2007]

Seam Carving

- Find a path, from top to bottom (left/right), that destroys as little image content as possible

- Give importance of image content determined by changes to neighboring pixels

$$e = \left|\frac{\partial I}{\partial x}\right| + \left|\frac{\partial I}{\partial y}\right|$$

$$\frac{\partial I}{\partial x} \approx \frac{I(x+1,y) - I(x-1,y)}{2}$$

central differences

$$\frac{\partial I}{\partial y} \approx \frac{I(x,y+1) - I(x,y-1)}{2}$$

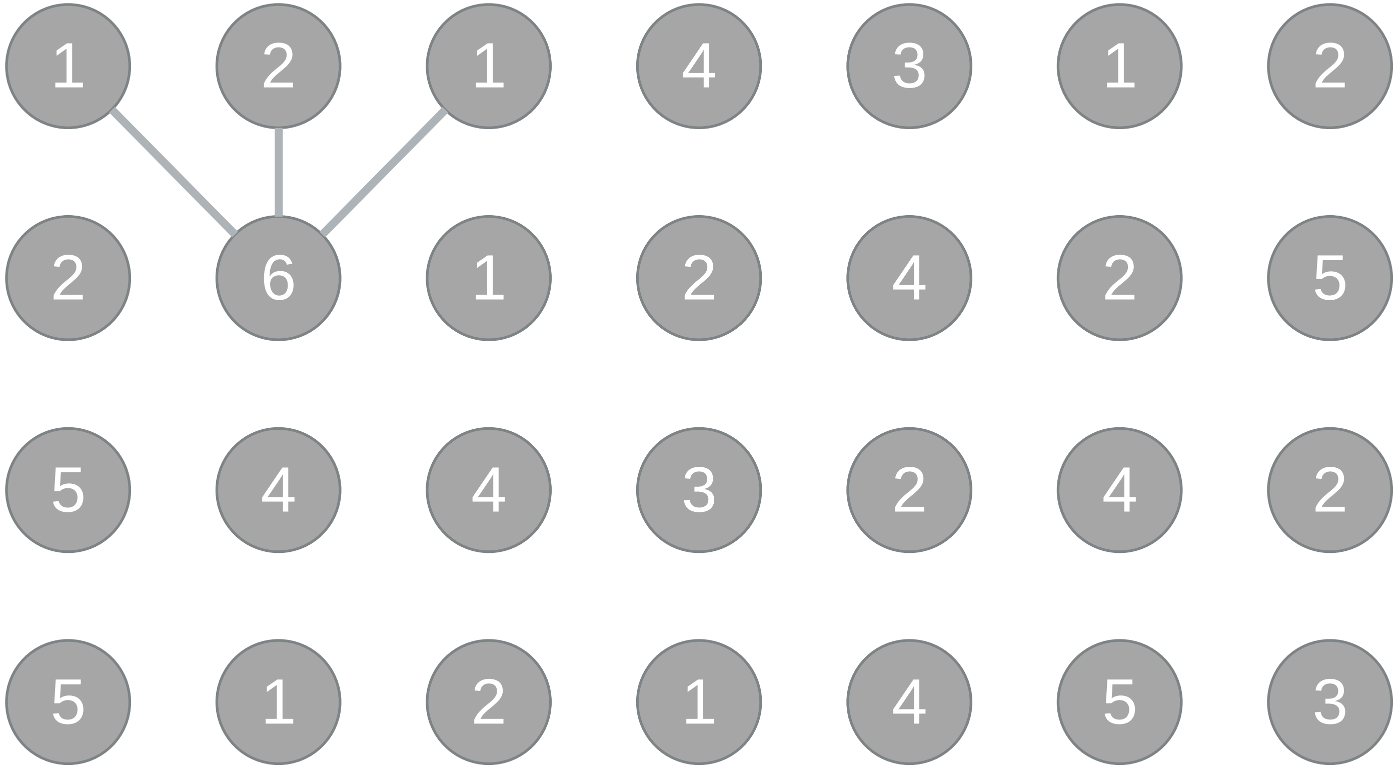
- source pixel (in top row) and destination pixel (in bottom row) are unknown

- Dijkstra for all point pairs would be too inefficient

Example: Shortest Path through an Image

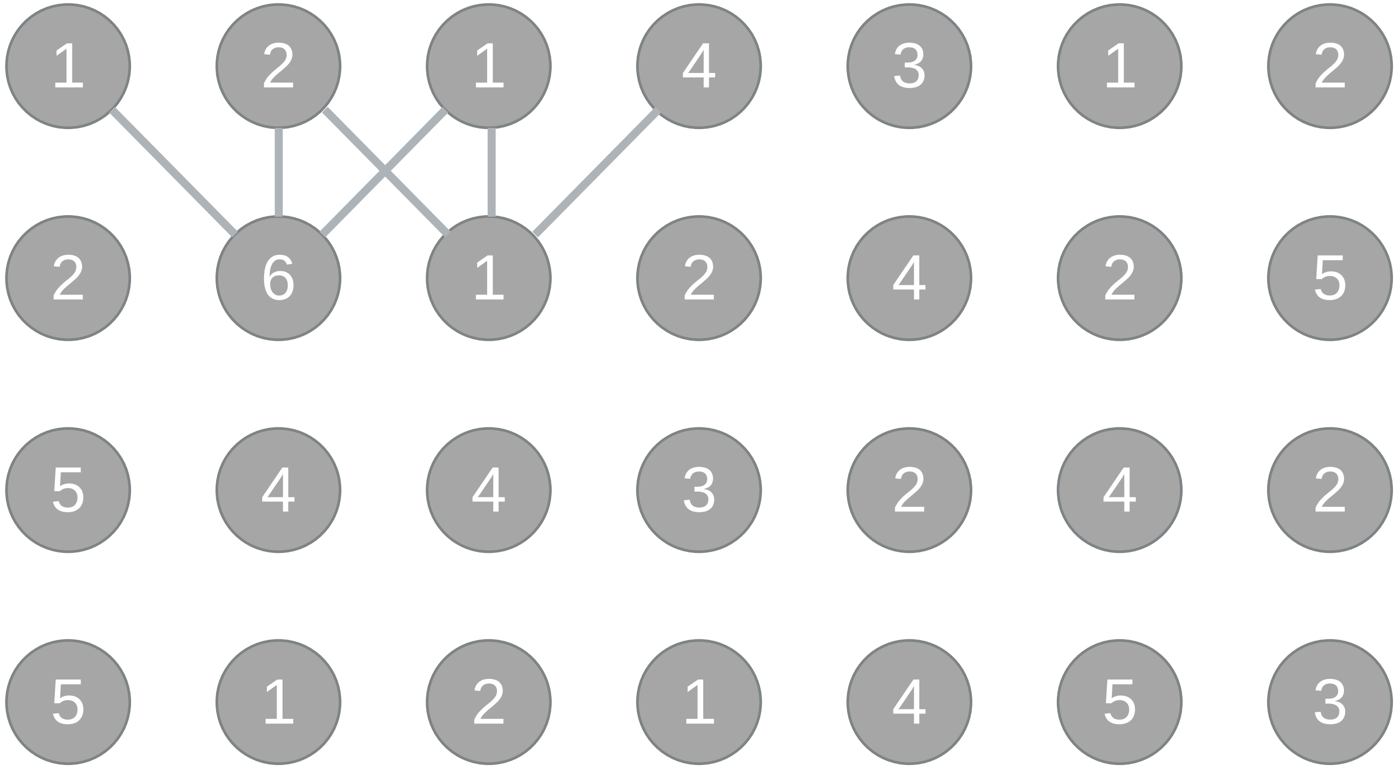
1	2	1	4	3	1	2
2	6	1	2	4	2	5
5	4	4	3	2	4	2
5	1	2	1	4	5	3

Example: Shortest Path through an Image



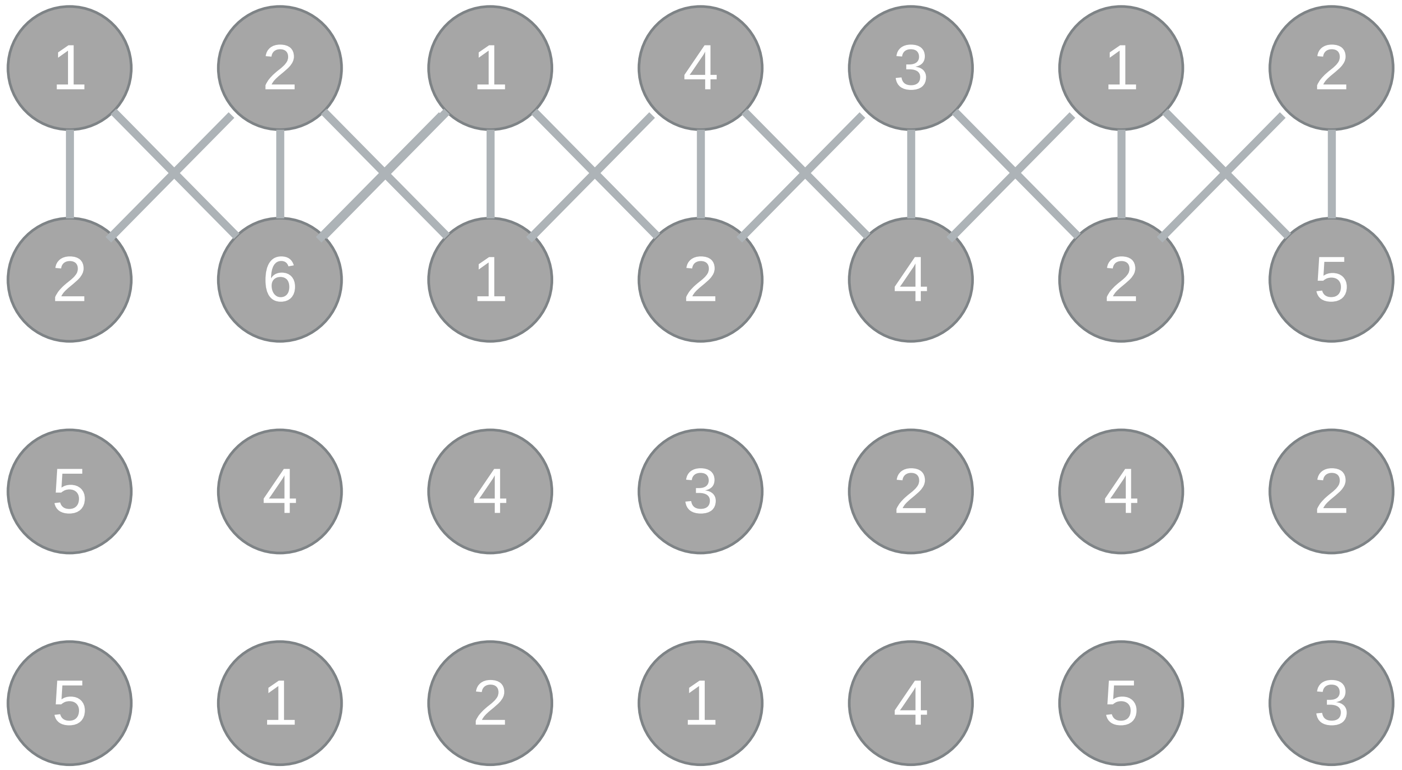
- Graph implicitly given on image: e.g. virtual edges to three predecessors

Example: Shortest Path through an Image



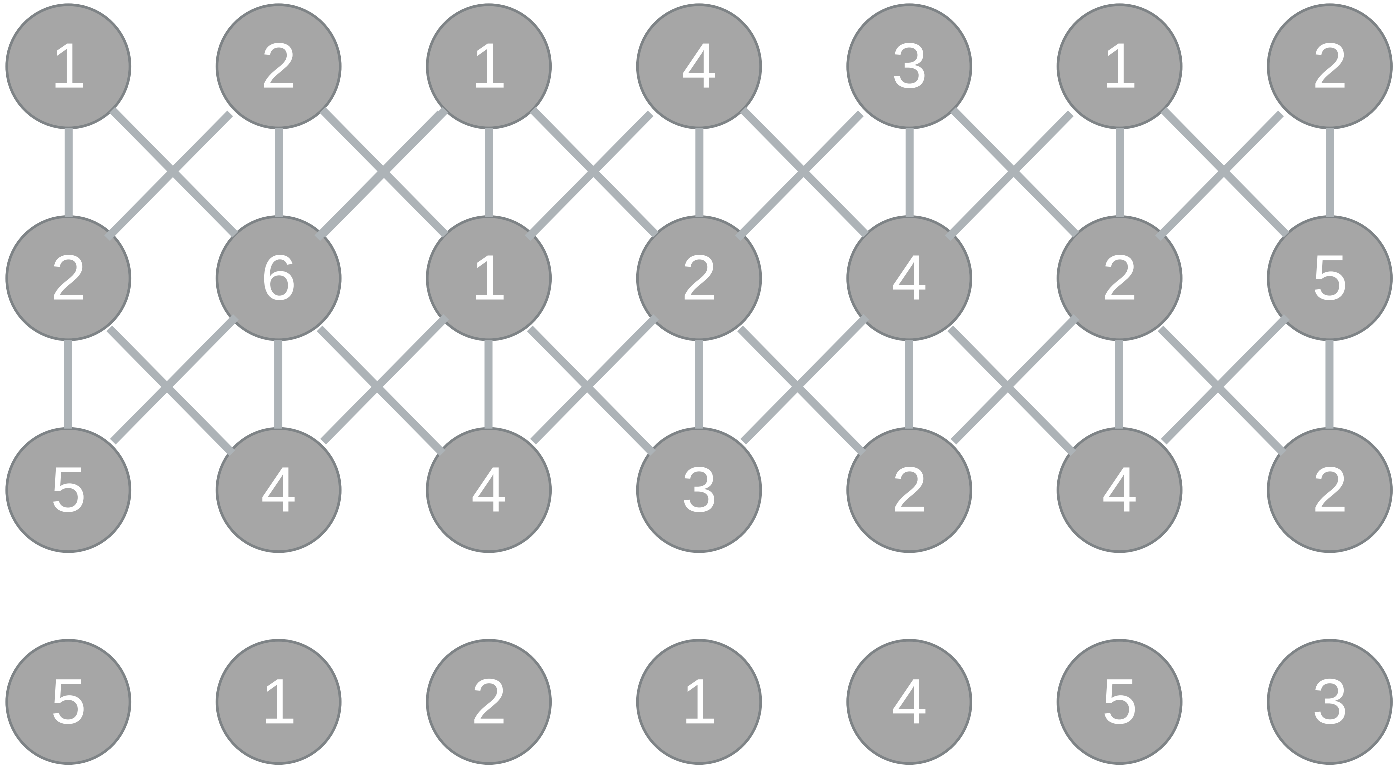
- Graph implicitly given on image: e.g. virtual edges to three predecessors

Example: Shortest Path through an Image



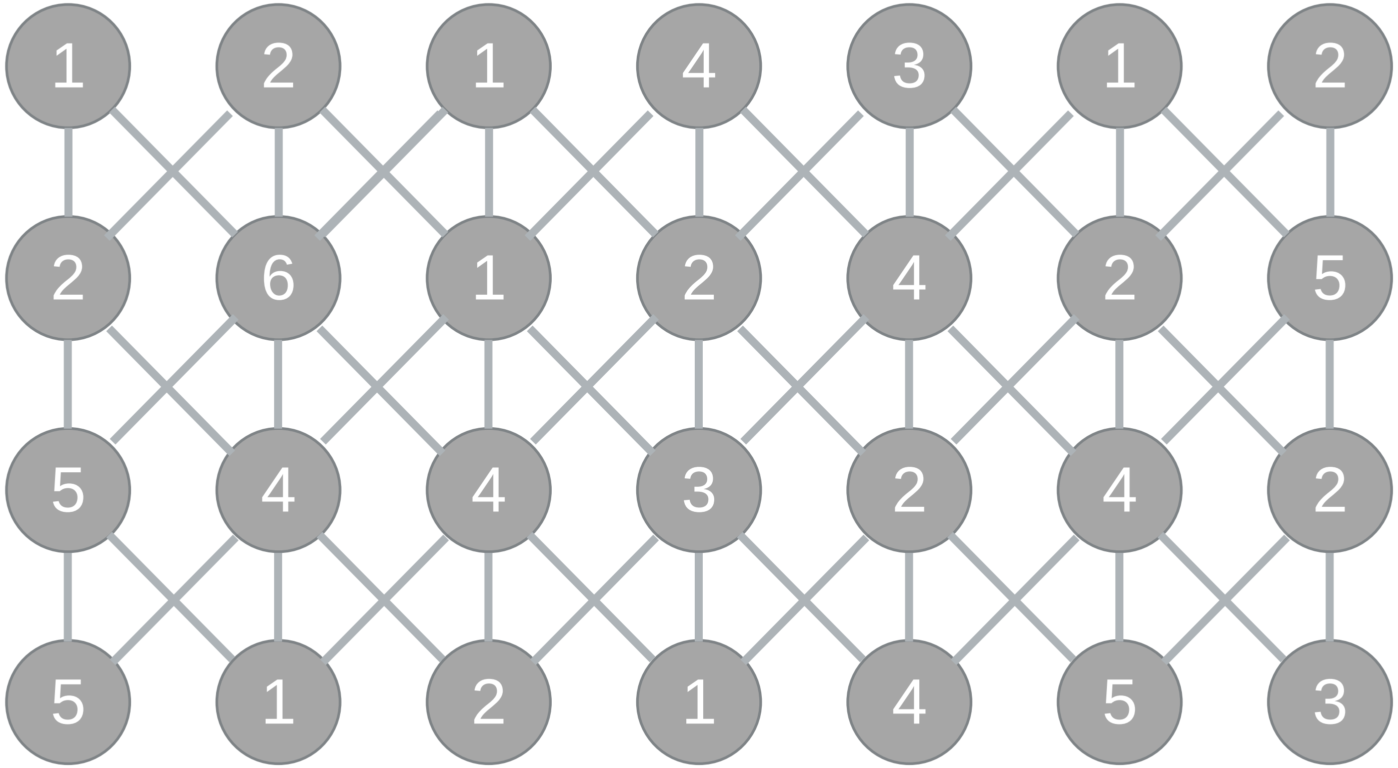
- Graph implicitly given on image: e.g. virtual edges to three predecessors

Example: Shortest Path through an Image



- Graph implicitly given on image: e.g. virtual edges to three predecessors

Example: Shortest Path through an Image



- Graph implicitly given on image: e.g. virtual edges to three predecessors

Shortest Path with Dynamic Programming

- Path weight: sum of the node weights along the path

In each line

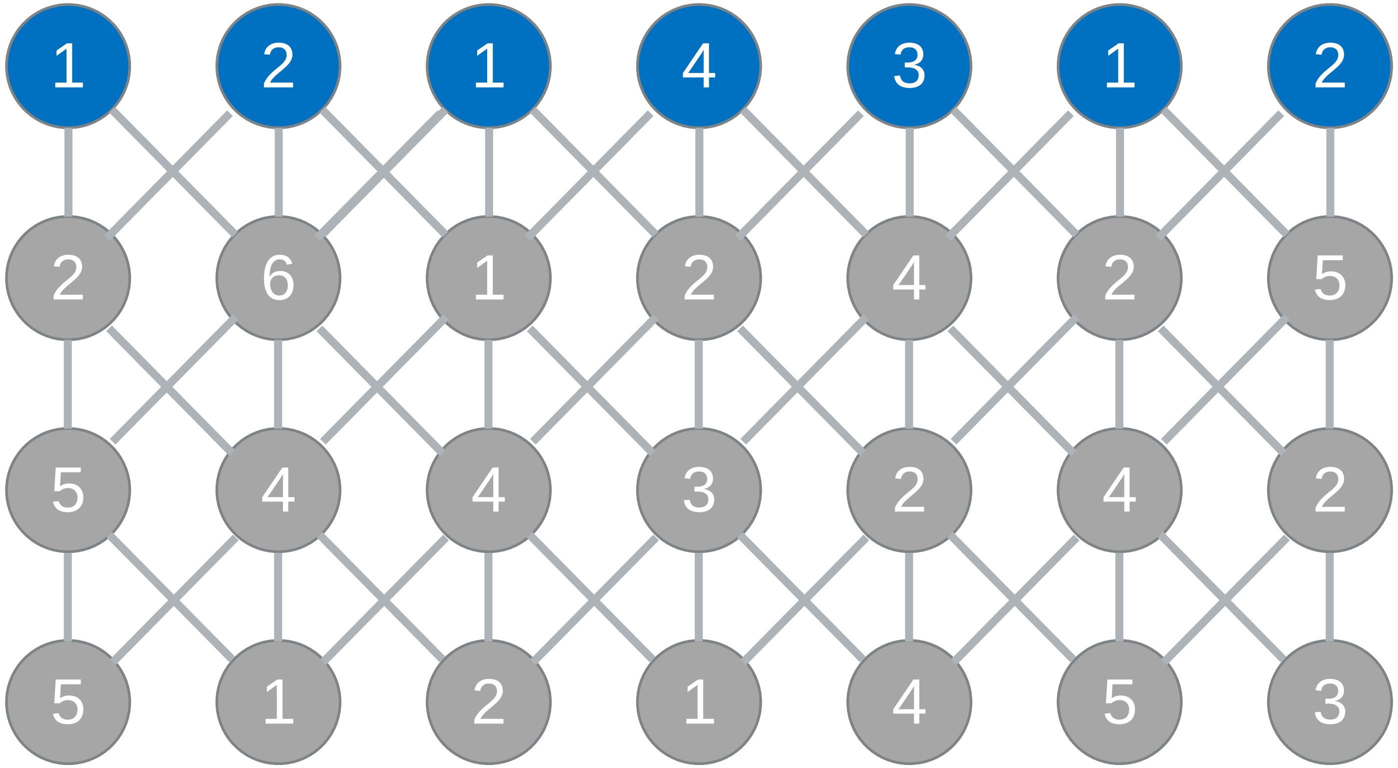
- For each point, calculate the shortest path to the row above it
  - From the **three** predecessors choose the one with the smallest path weight
  - Add node weight

- source(s) in the top row

- Remember predecessors

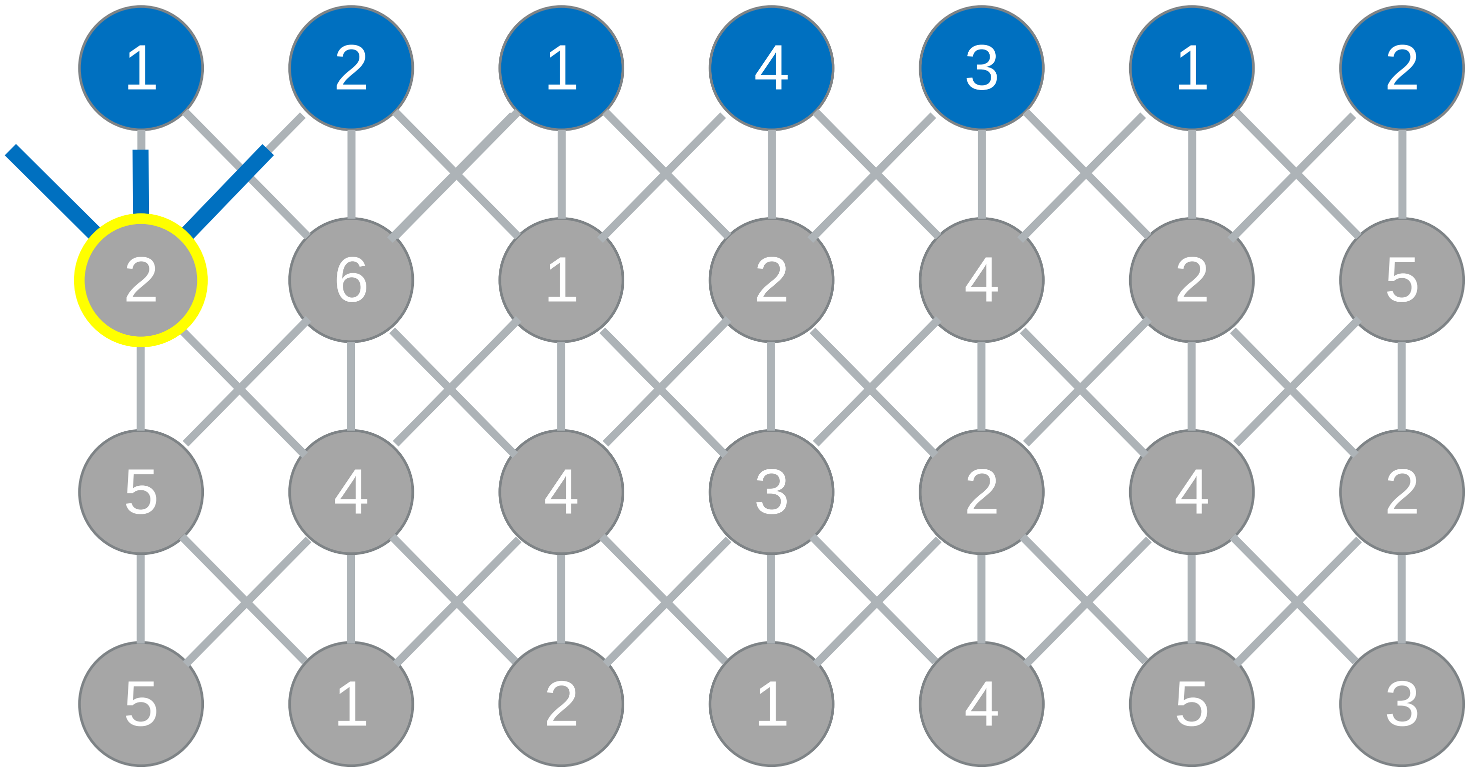
- In the last row, determine pixel with smallest path weight

Example: Shortest Path through an Image



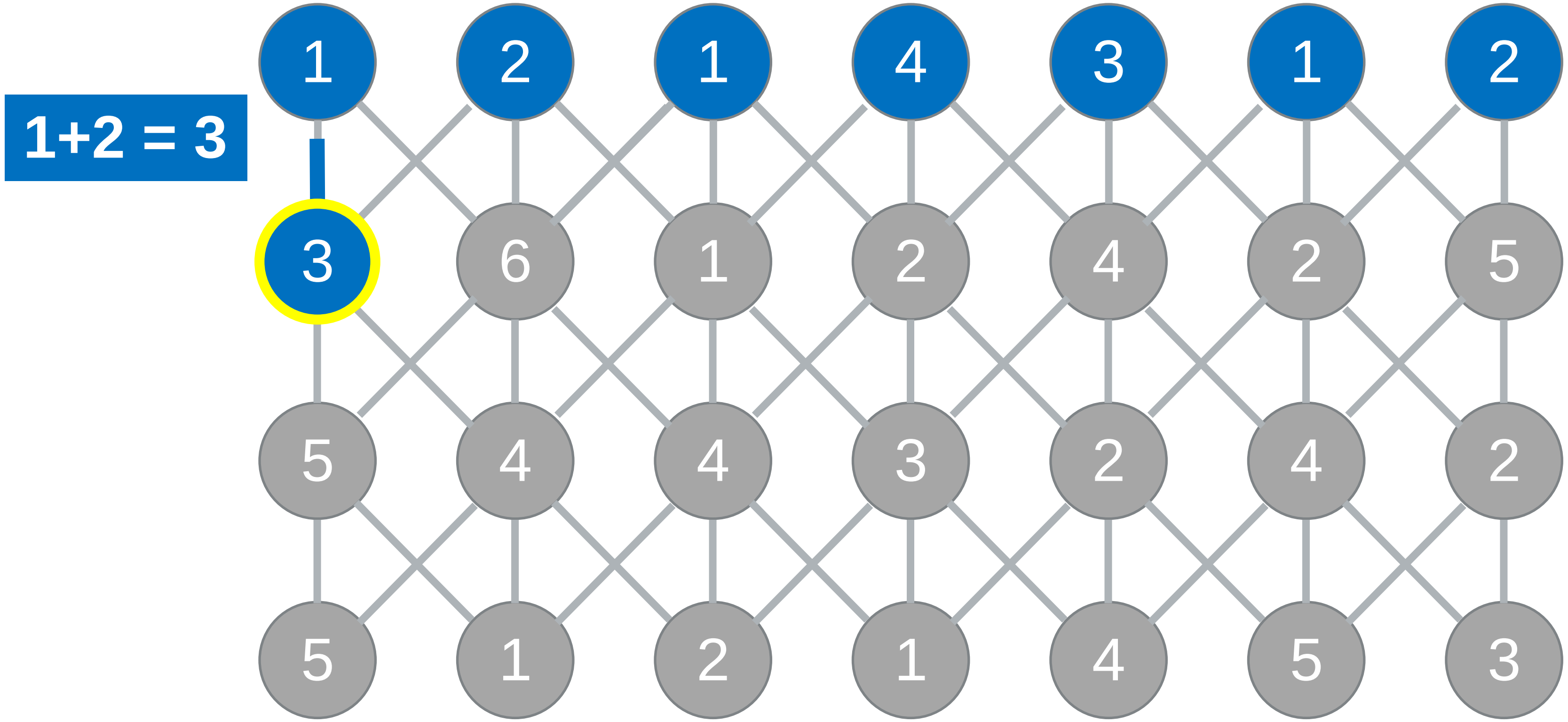
- Trivial for first row: Shortest path of the pixel → itself

Example: Shortest Path through an Image



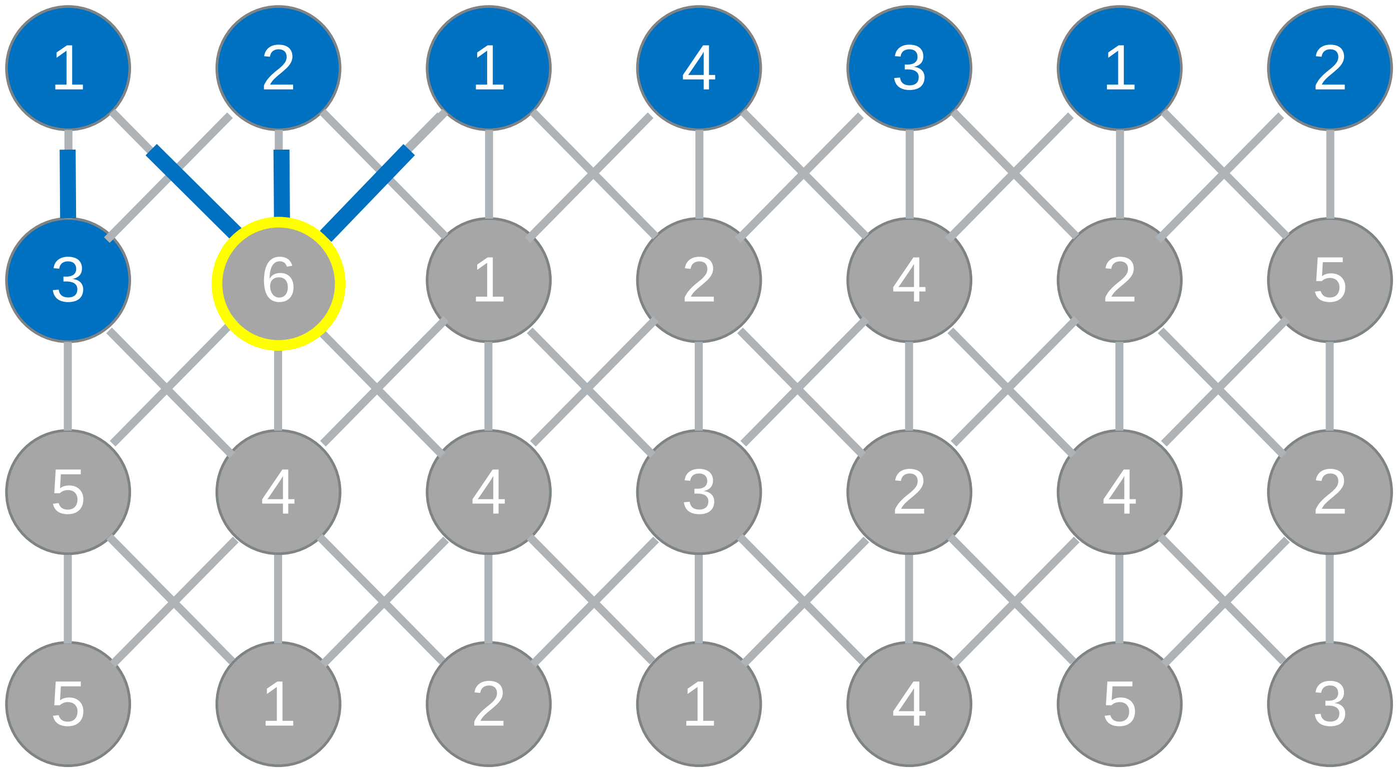
- Second row:
  - Find the most favorable path from the three predecessors
  - Add node weight

Example: Shortest Path through an Image



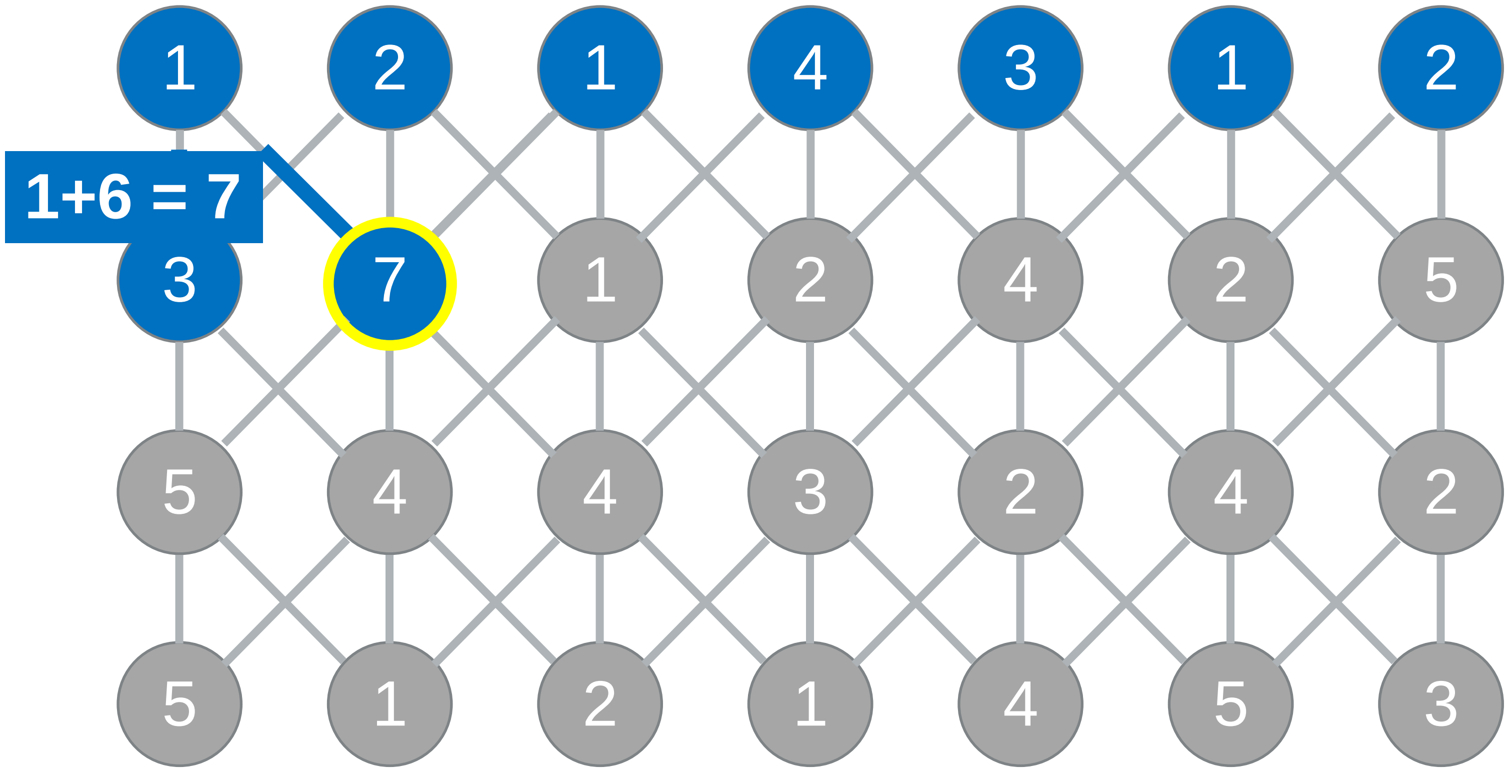
- Second row:
  - Find the most favorable path from the three predecessors
  - Add node weight

Example: Shortest Path through an Image



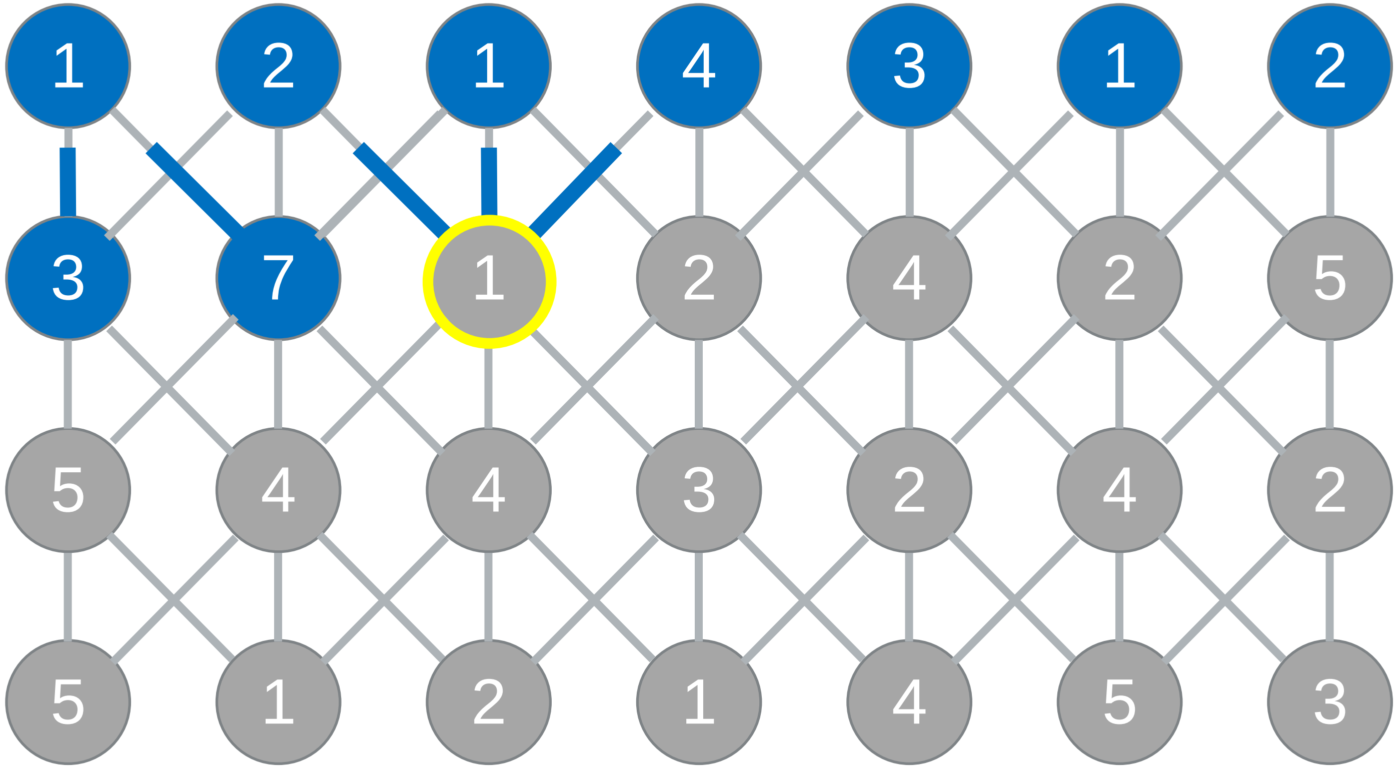
- Second row:
  - Find the most favorable path from the three predecessors
  - Add node weight

Example: Shortest Path through an Image



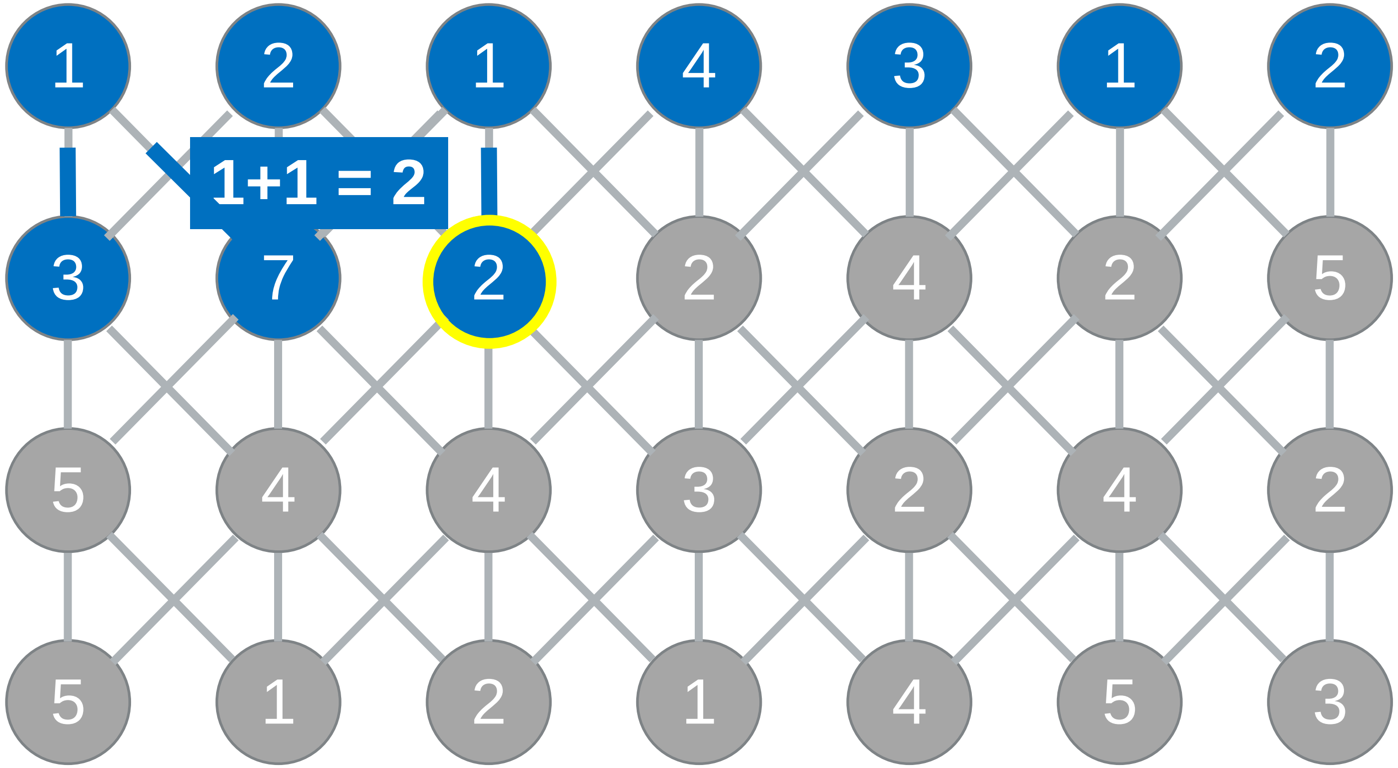
- Second row:
  - Find the most favorable path from the three predecessors
  - Add node weight

Example: Shortest Path through an Image



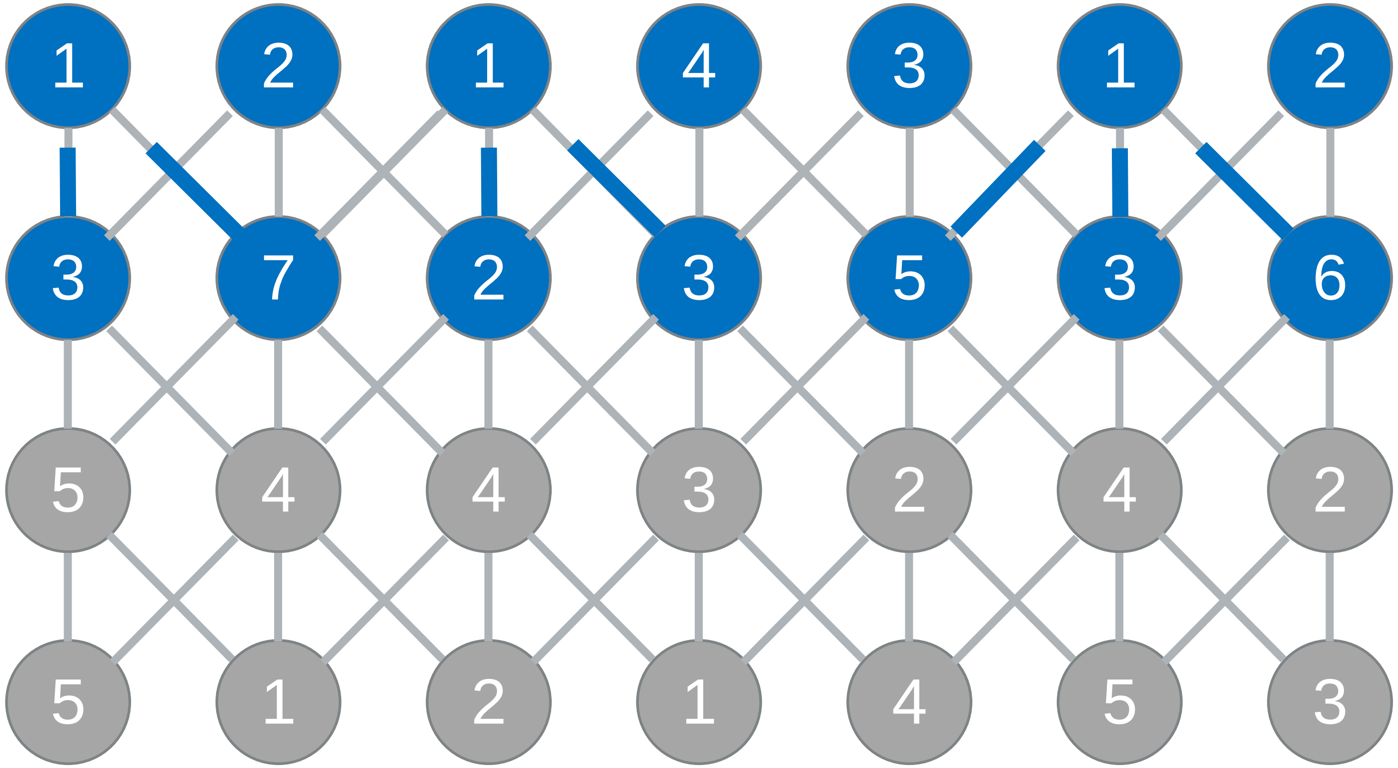
- Second row:
  - Find the most favorable path from the three predecessors
  - Add node weight

Example: Shortest Path through an Image



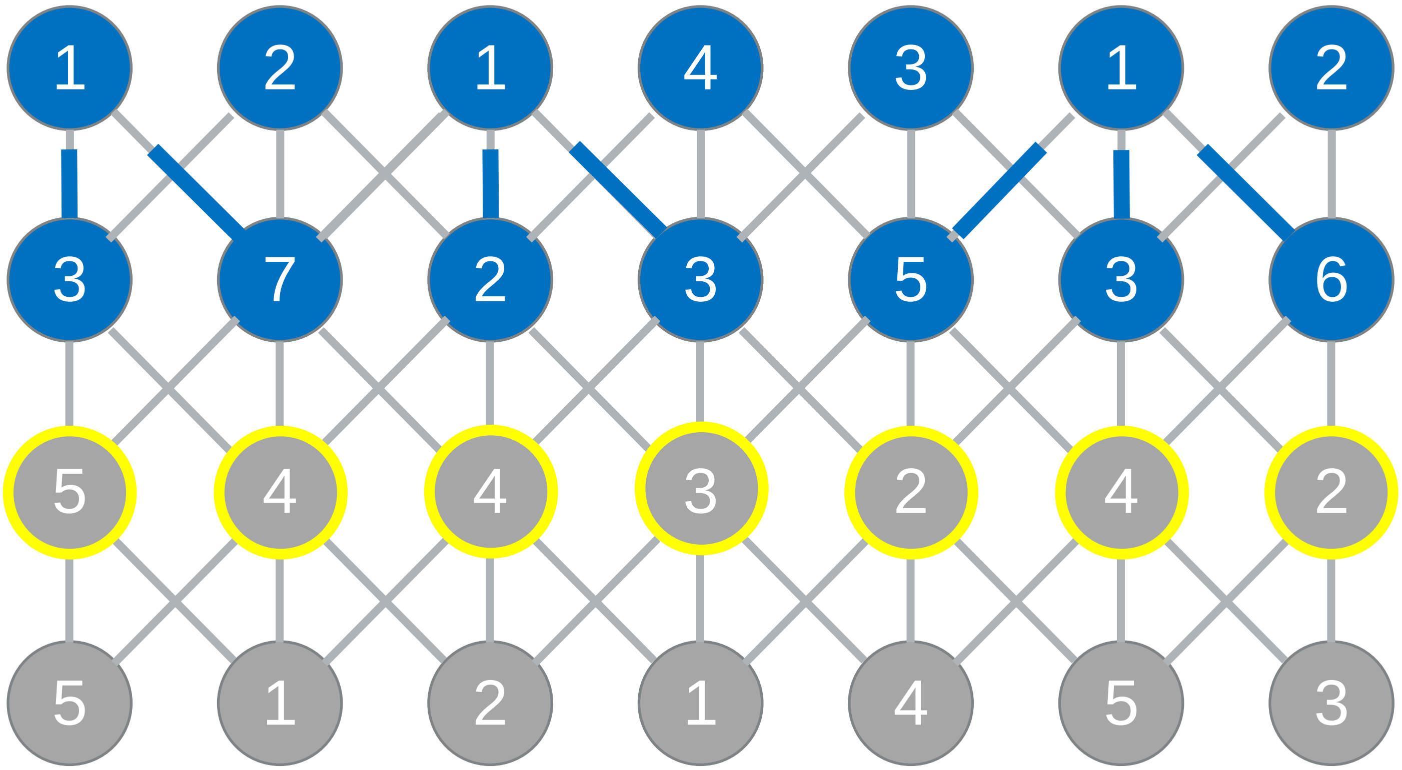
- Second row:
  - Find the most favorable path from the three predecessors
  - Add node weight

Example: Shortest Path through an Image



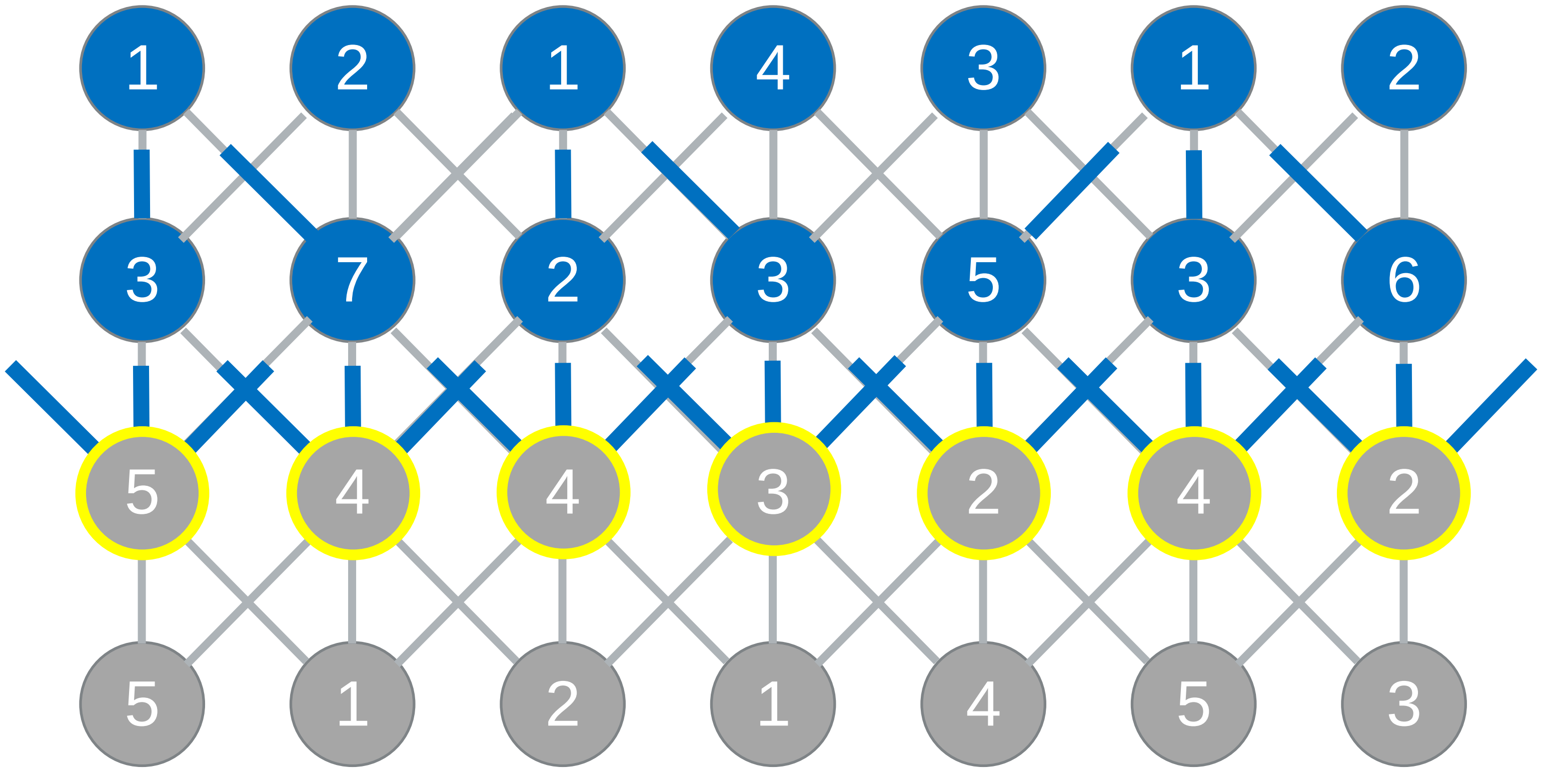
- Second row:
  - Find the most favorable path from the three predecessors
  - Add node weight

Example: Shortest Path through an Image



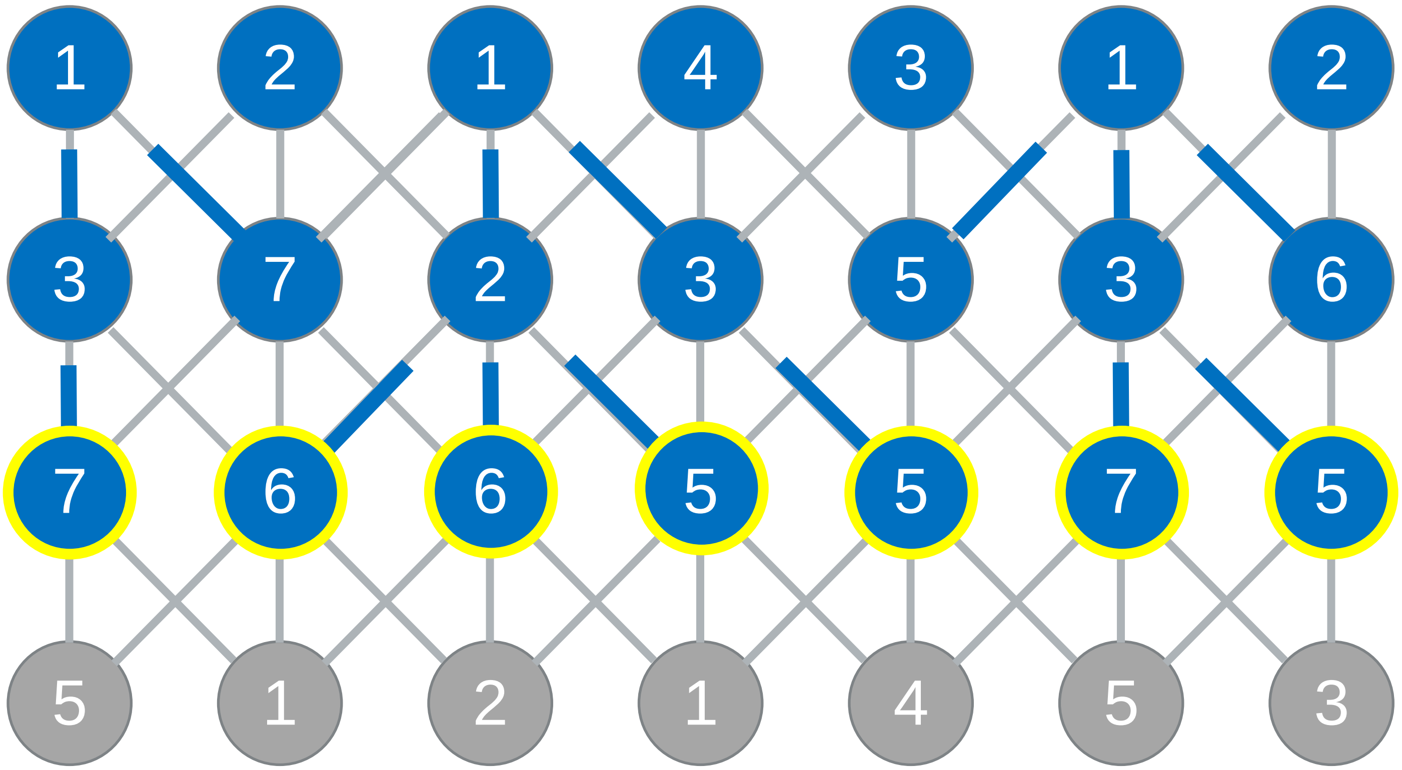
- Third row:
  - Find the most favorable path from the three predecessors
  - Add node weight

Example: Shortest Path through an Image



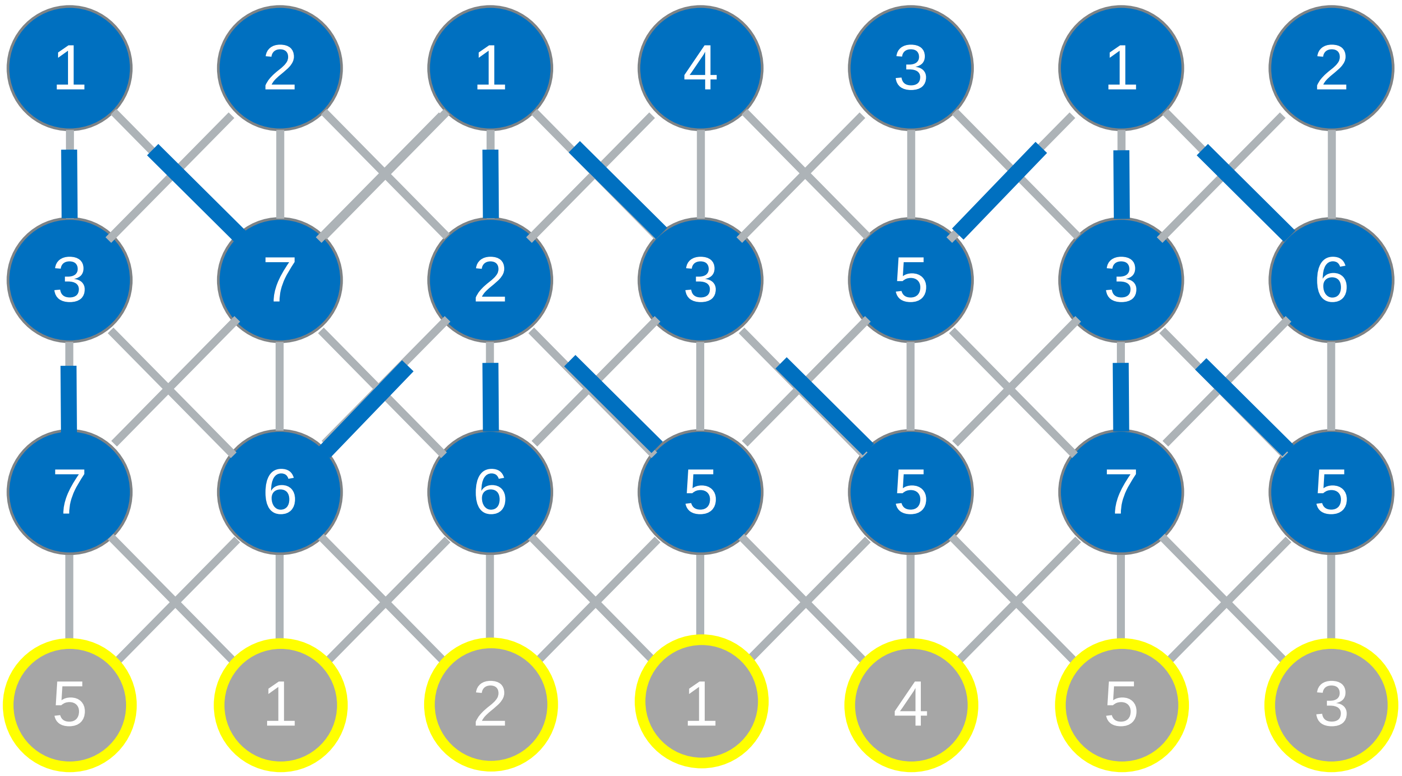
- Third row:
  - Find the most favorable path from the three predecessors
  - Add node weight

Example: Shortest Path through an Image



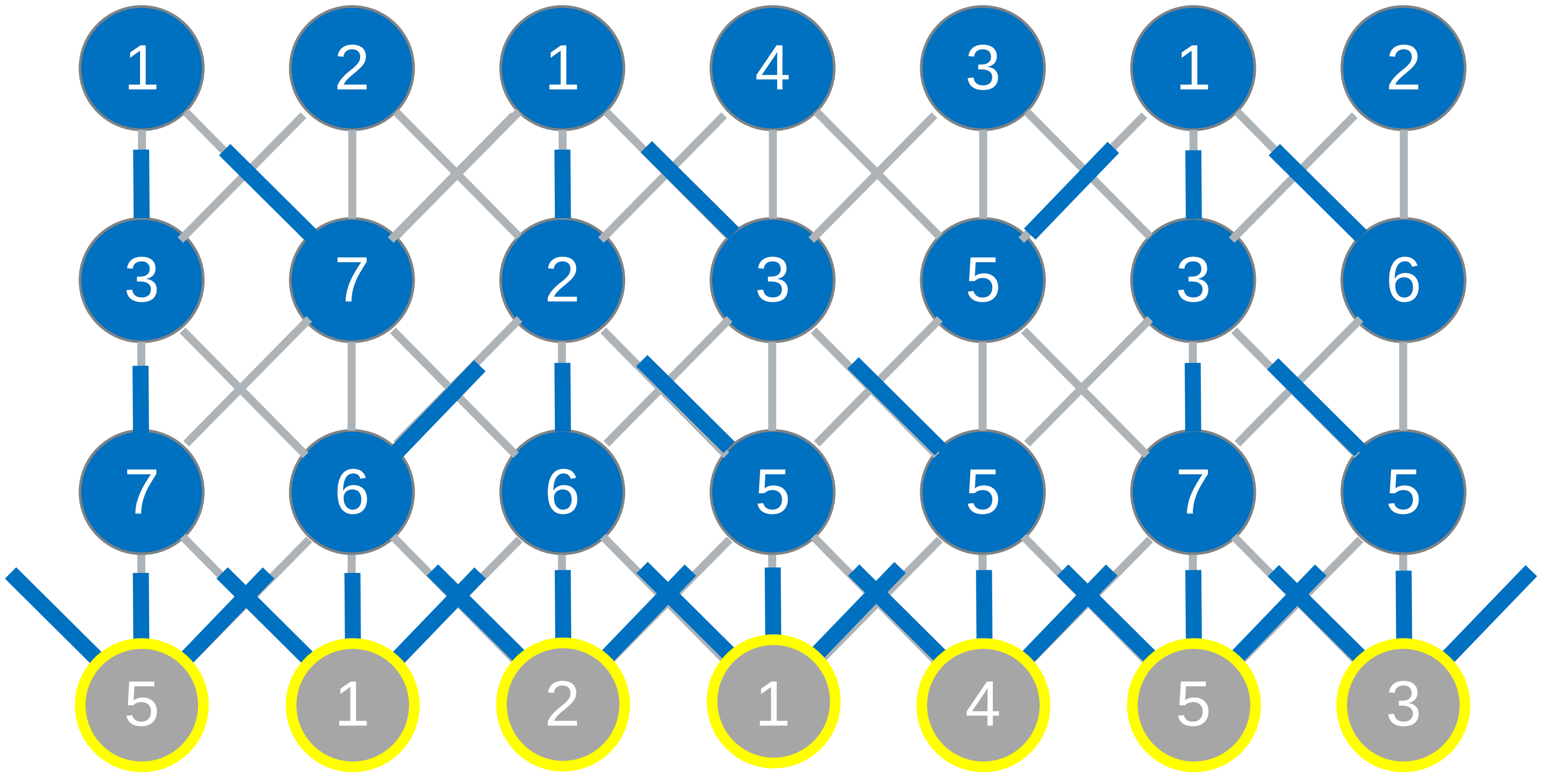
- Third row:
  - Find the most favorable path from the three predecessors
  - Add node weight

Example: Shortest Path through an Image



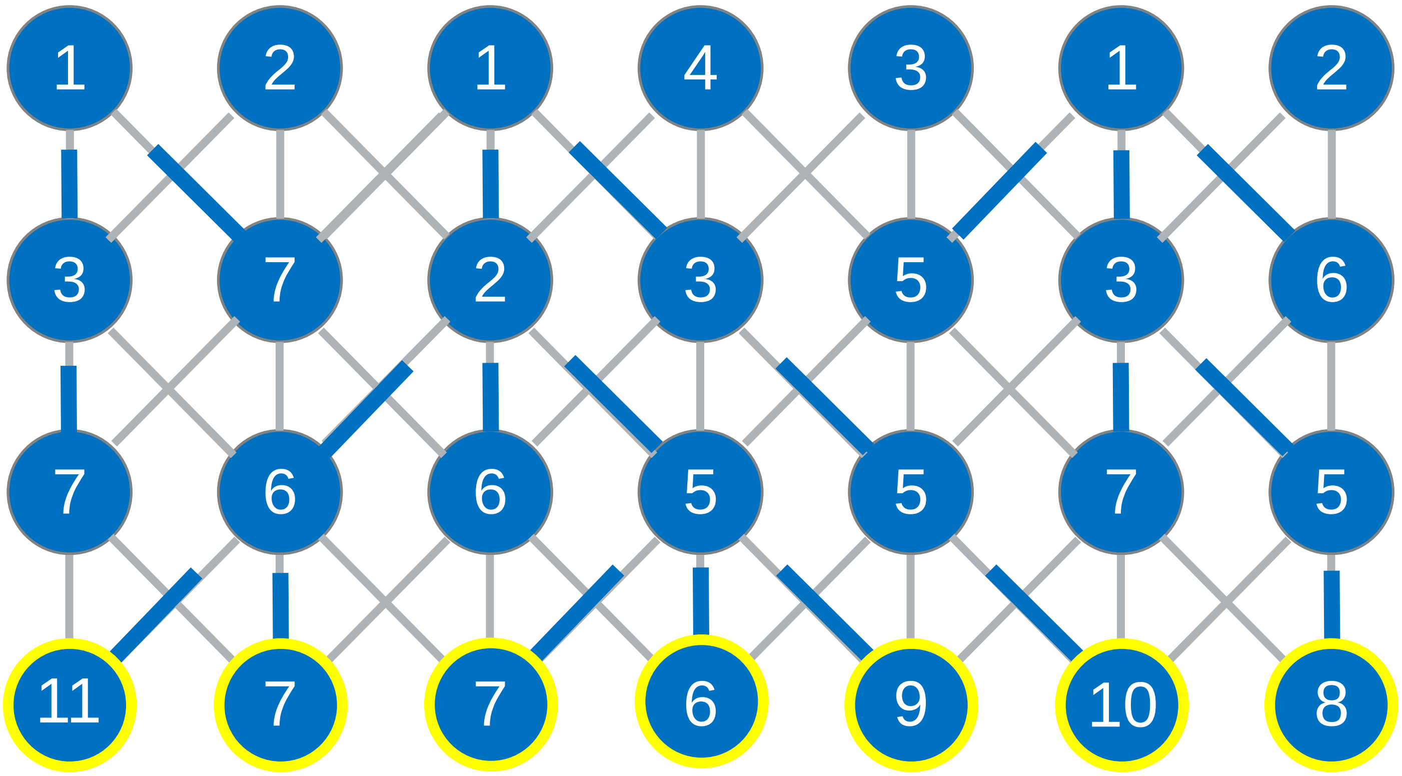
- Fourth row:
  - Find the most favorable path from the three predecessors
  - Add node weight

Example: Shortest Path through an Image



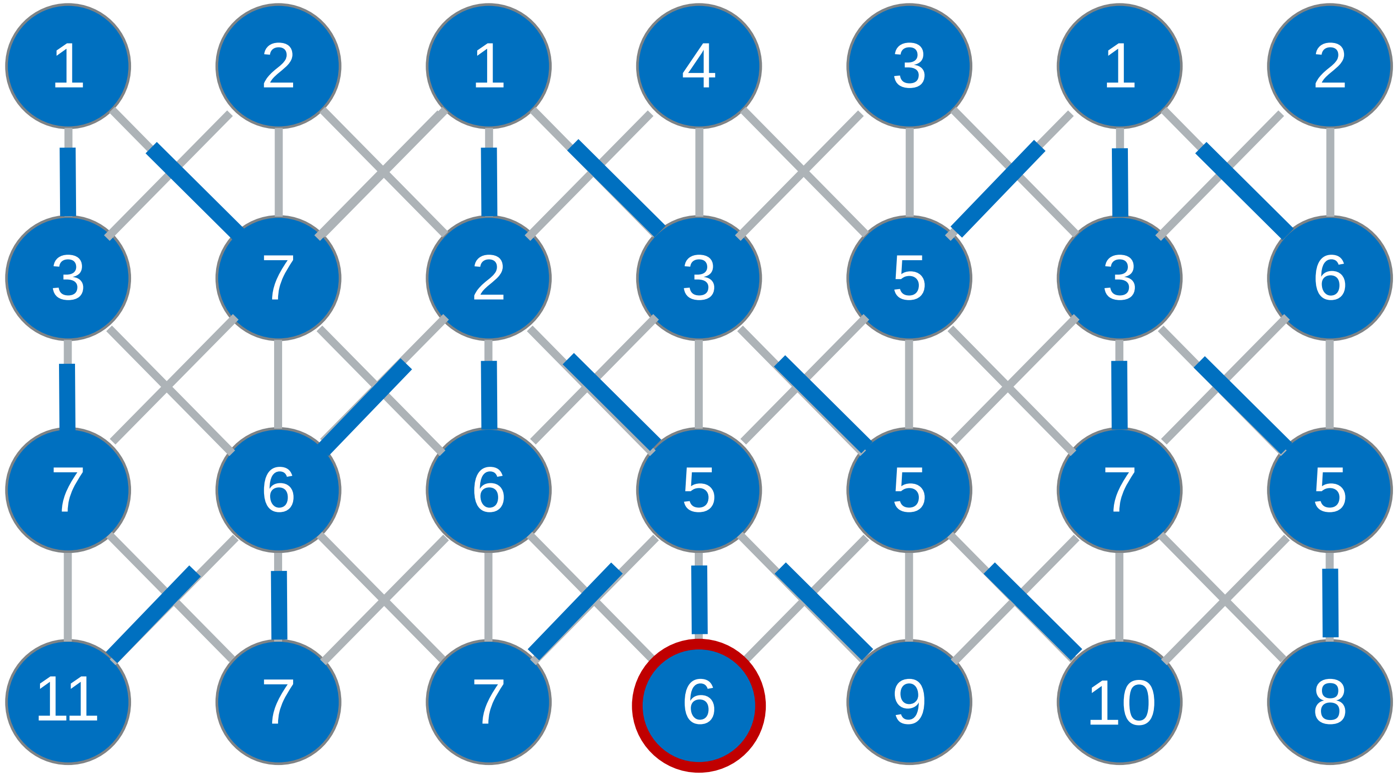
- Fourth row:
  - Find the most favorable path from the three predecessors
  - Add node weight

Example: Shortest Path through an Image



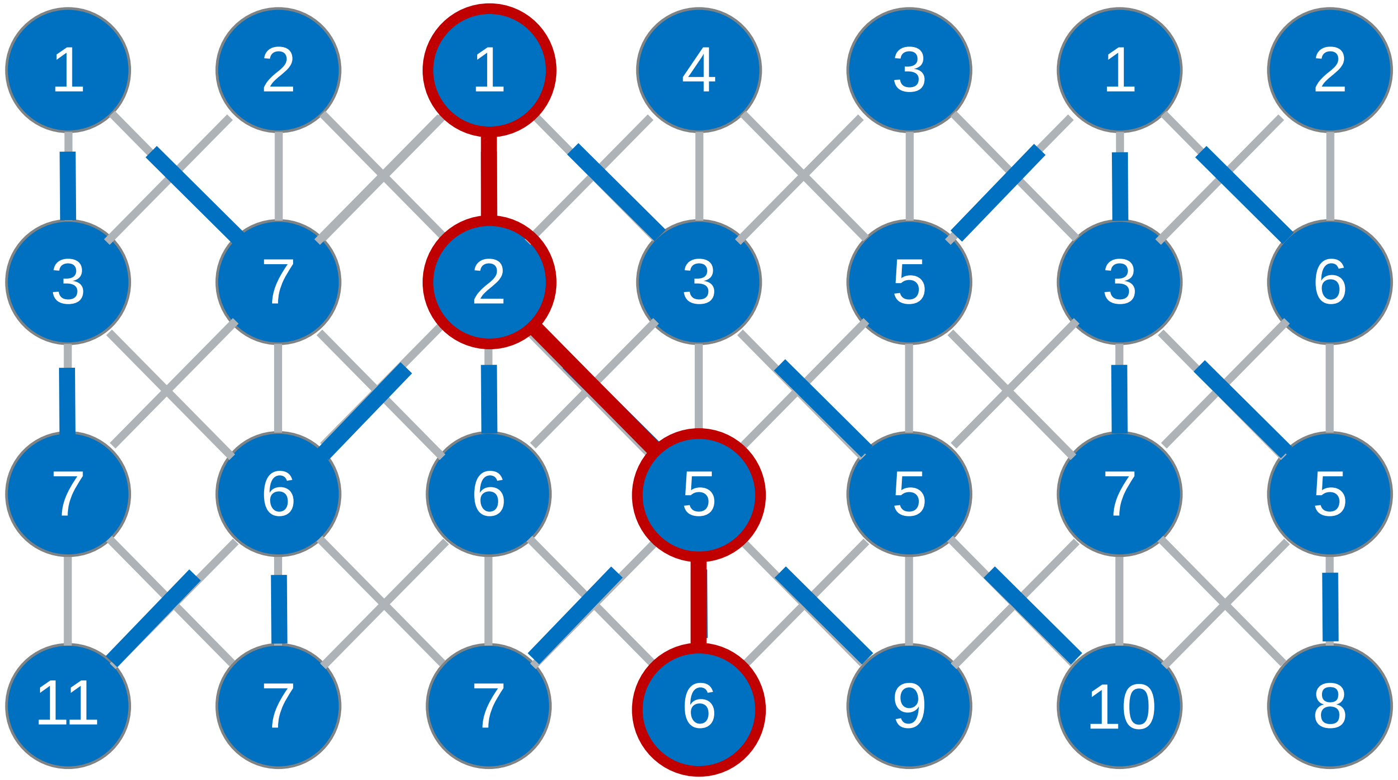
- Fourth row:
  - Find the most favorable path from the three predecessors
  - Add node weight

Example: Shortest Path through an Image



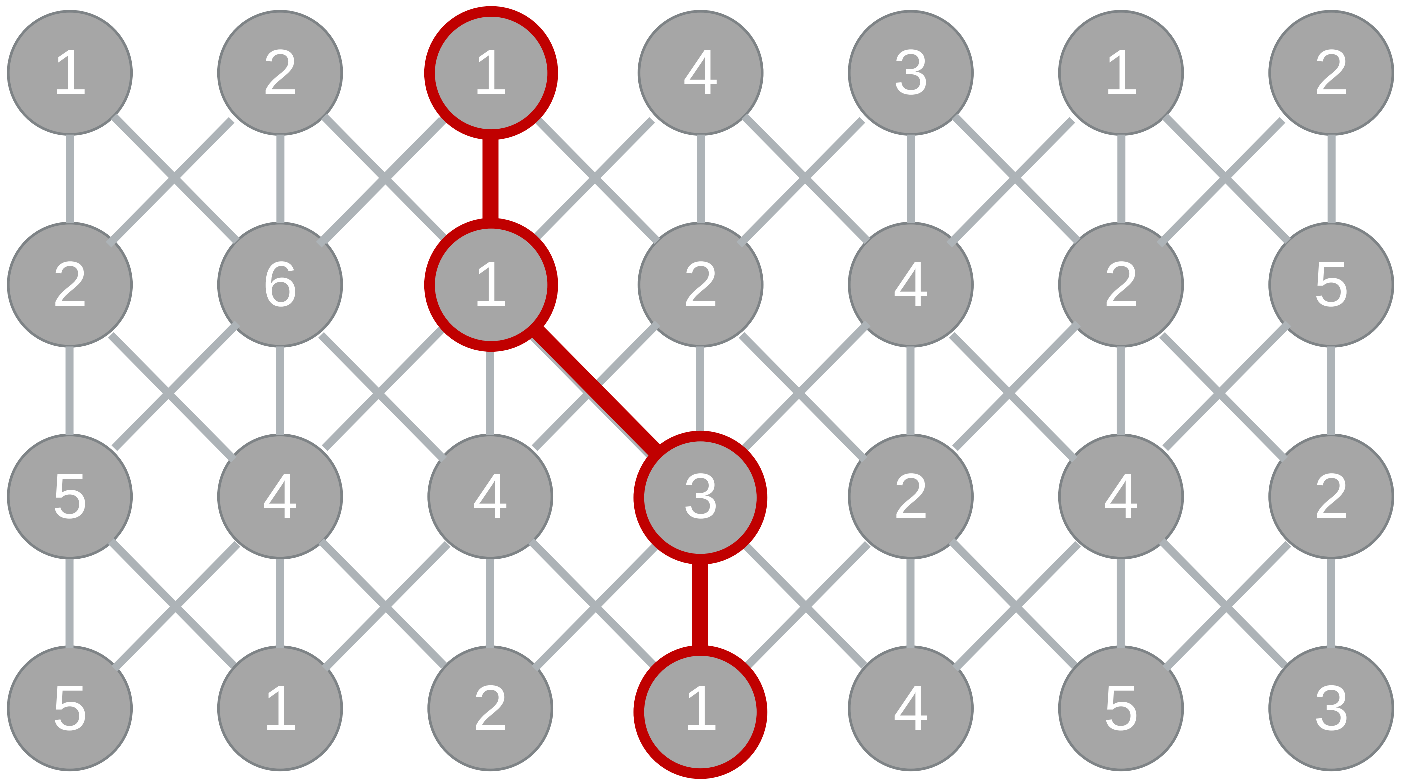
- Last row:
  - Smallest accumulated path weight →
  - source of the shortest path

Example: Shortest Path through an Image



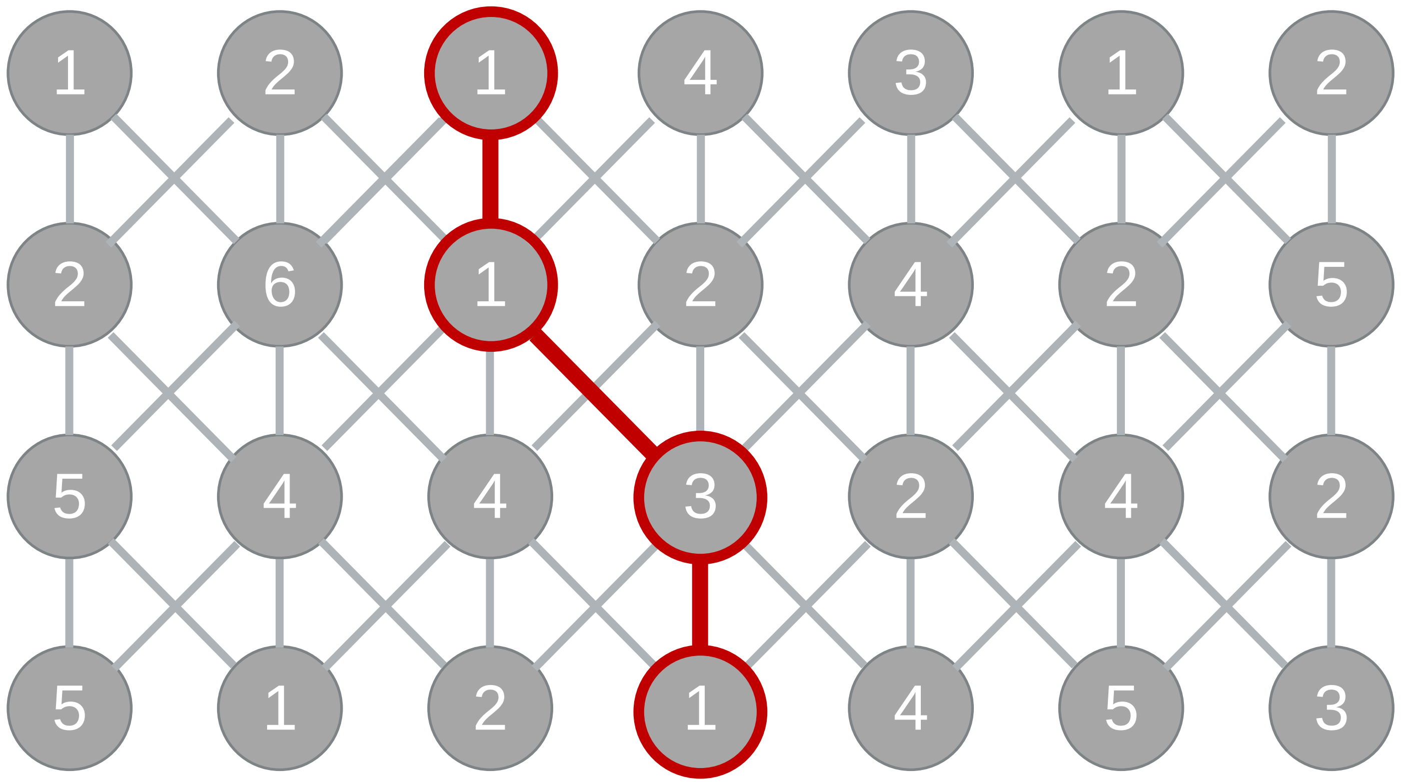
- Last row:
  - Smallest accumulated path weight →
  - source of the shortest path

Example: Seam Carving



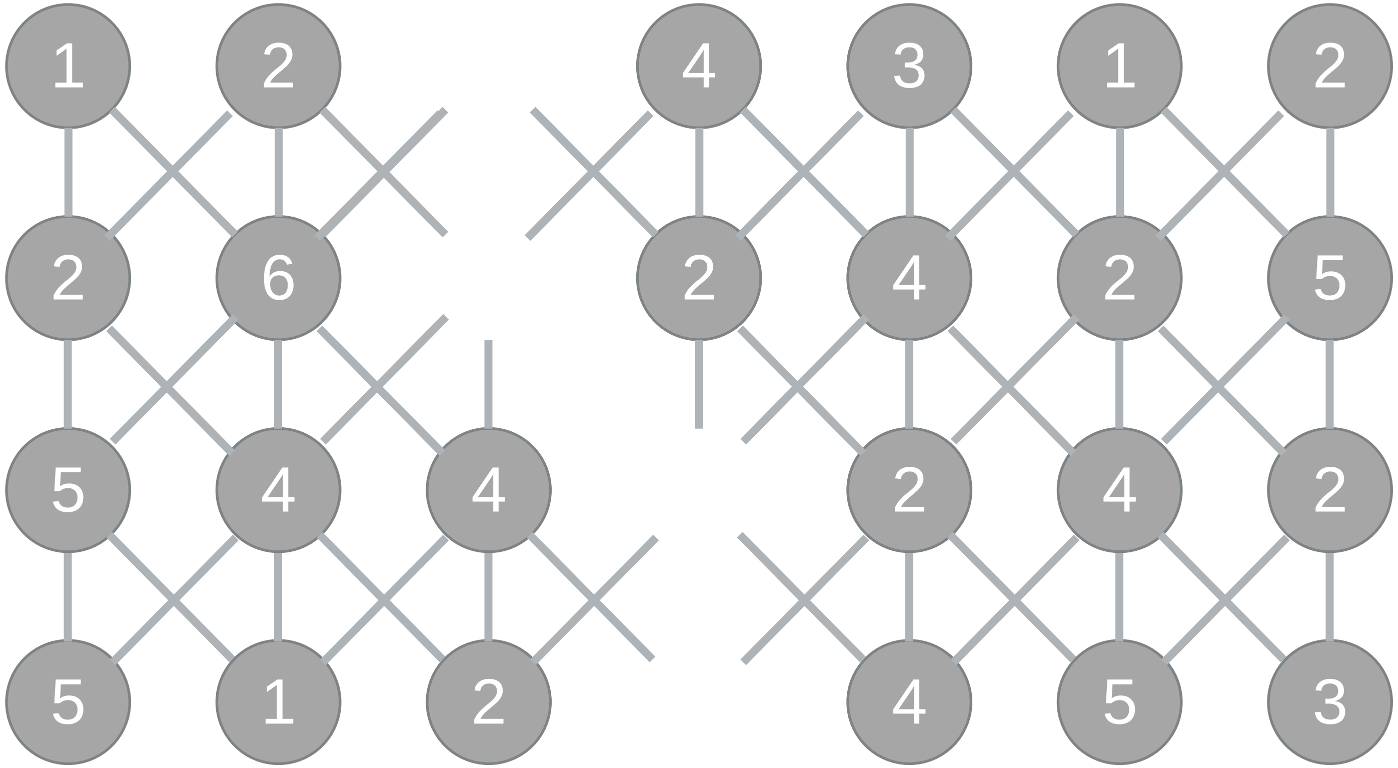
- Remove all pixels along the shortest path

Seam Carving



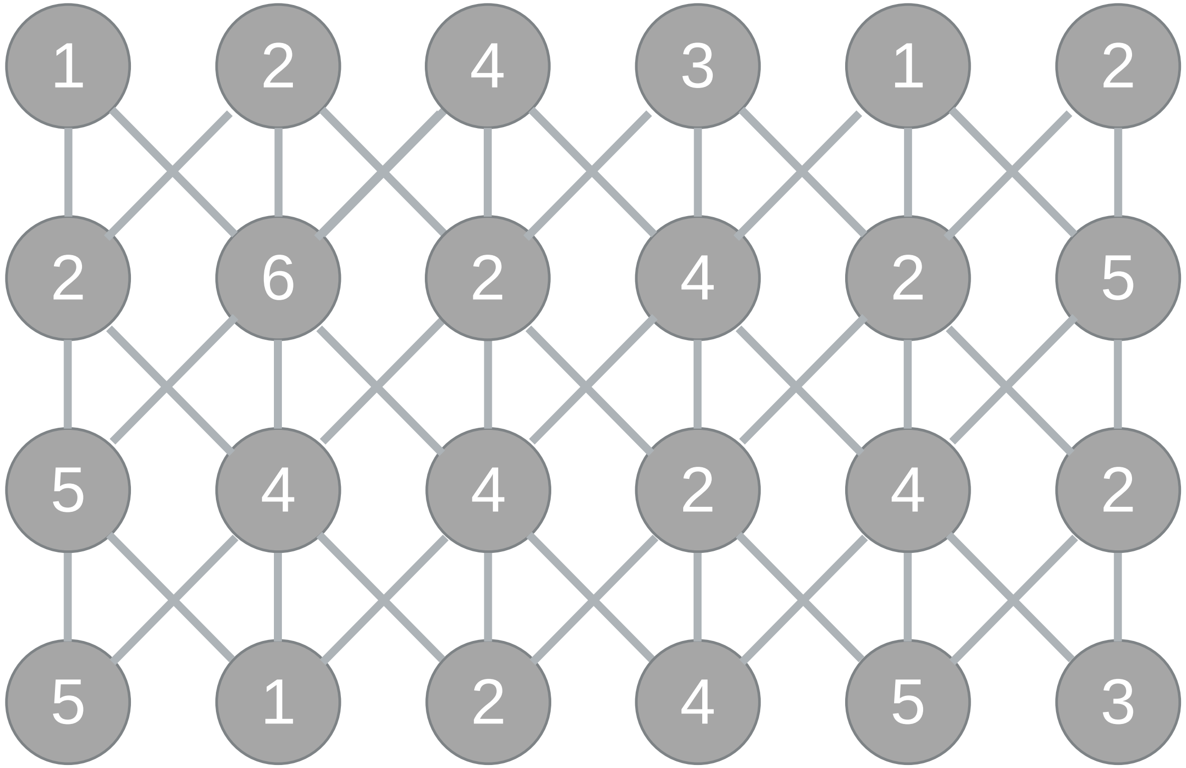
- Remove all pixels along the shortest path

Seam Carving



- Remove all pixels along the shortest path

Seam Carving



- Do … repeat

Summary

- For large structured data, explicitly building a Graph is too much overhead.
- For structured data the graph is not explicitly built but e.g., implicitly given by image structure

A*-Algorithm

Recap Dijkstra’s Algorithm – Shortest Path

The currently determined shortest subpath never becomes shorter by later processing of additional nodes

- Already visited nodes do not need to be looked at again

Priority queue:

- Insertion of all nodes vs. insertion of just reachable nodes
- Heap vs. set (elements must also be changed)

Properties of Dijkstra's algorithm:

- Similar to breadth-first search
- No targeted search
- Guarantees the shortest path
- Growth isotropically in all directions ("Sphere" from source)

A*-Algorithm

Basic idea:

- Mix depth and breadth-first search (if necessary)
- Like Dijkstra: expand the currently most favorable path
- Prioritization is based on an estimate of the **total cost** of a path **to the destination**:

$$w(p) = w(s,p) + w(p,t)$$

- Cost from source point to p known / propagated. $w(s,p)$
- Remaining cost from p to destination estimated with **heuristics** $w(p,t) \approx h(p)$

Condition on the Heuristic

The heuristic is monotonic if:

1. Actual costs are **never overestimated** by the heuristic
2. For each node p and successor p' must hold:

$$h(p) \leq w(p, p') + h(p')$$

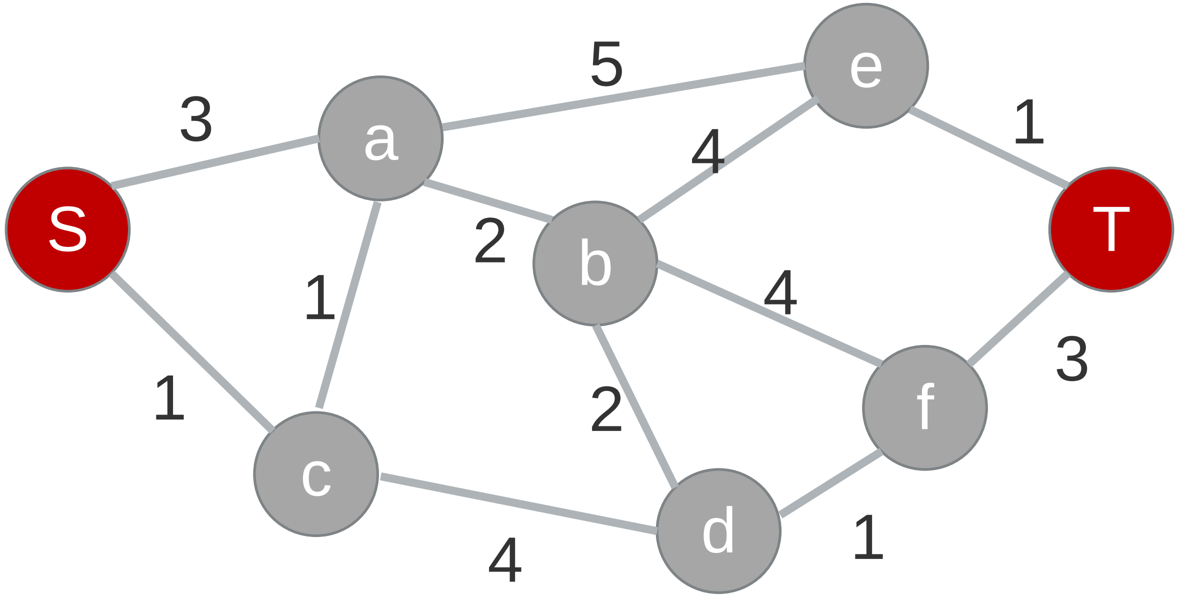
   (**triangle inequality**): estimated cost of a node p is smaller than the actual cost of the edge + estimated cost of that node

- If condition 2 is violated, the heuristic might still be valid

- **Example:** Euclidean distance (line of sight) in 2D maps is a monotonic heuristic for the actual travel distance between cities

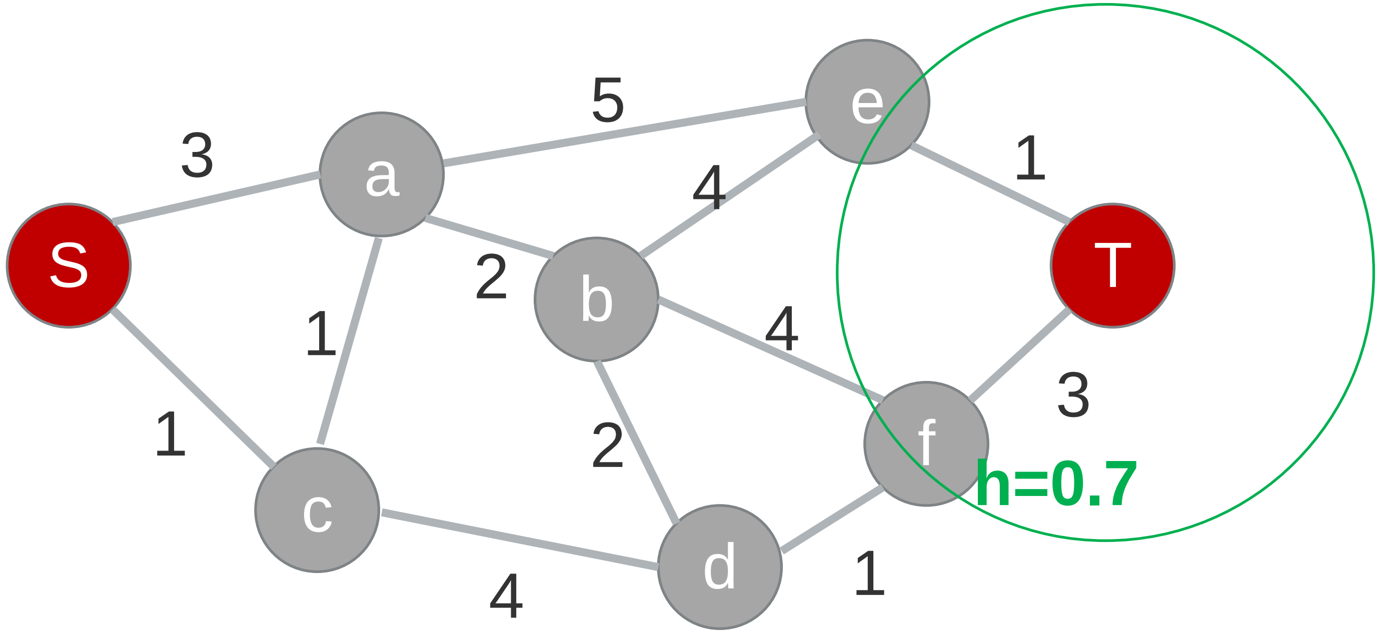
A*-Algorithm – Heuristic

- Weighted graph



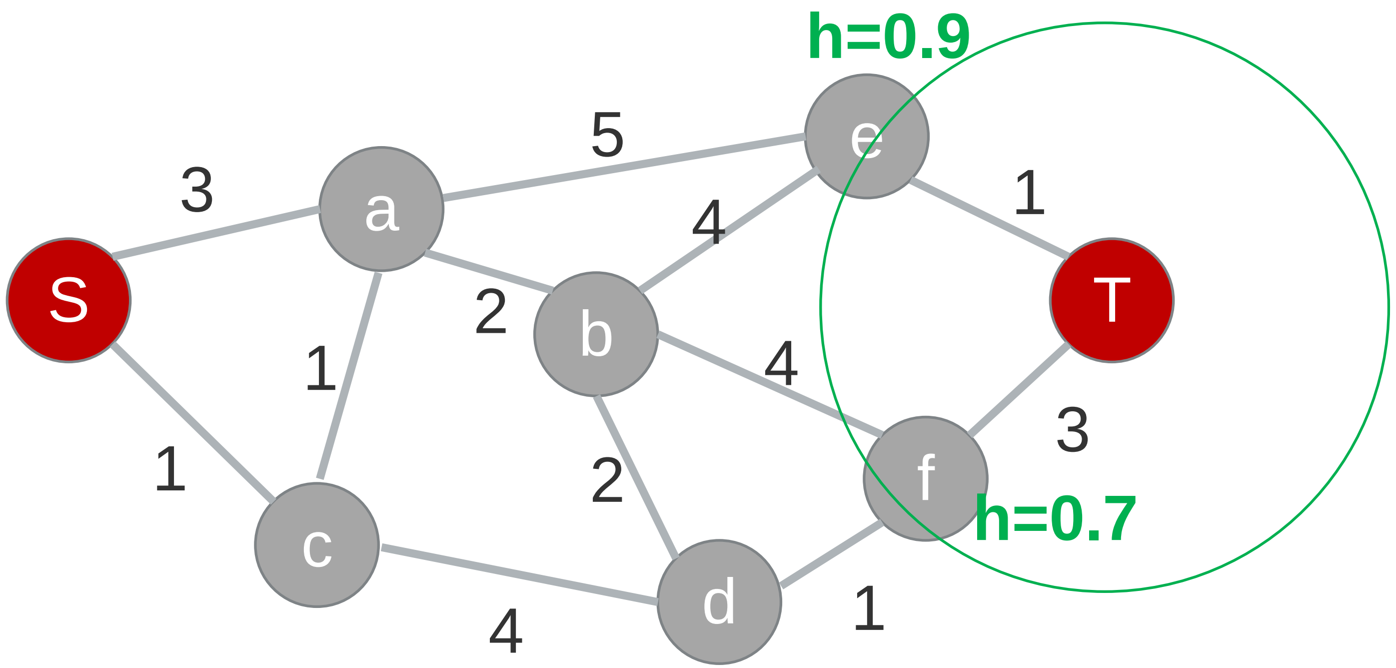
A*-Algorithm – Heuristic

- Weighted graph



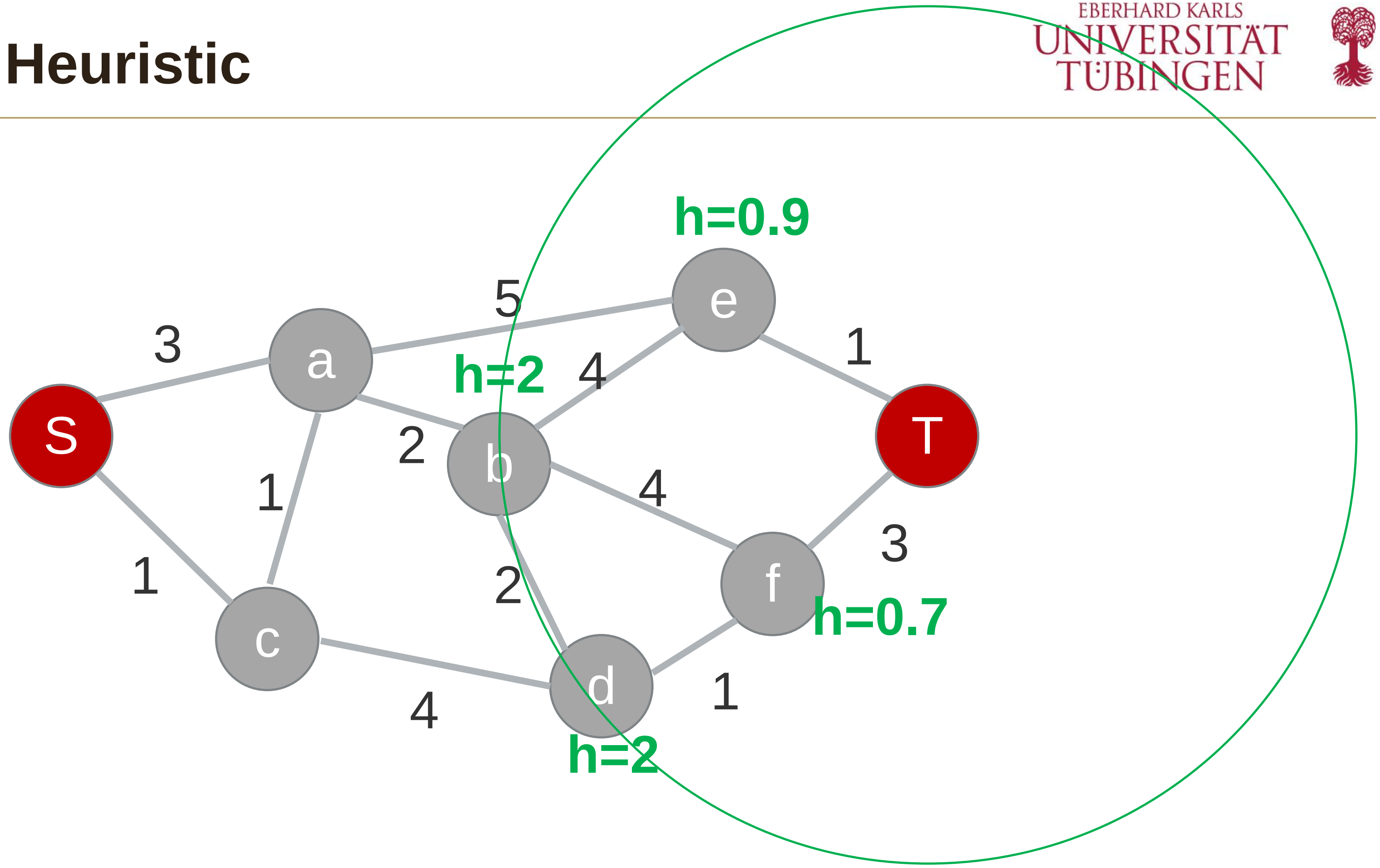
A*-Algorithm – Heuristic

- Weighted graph



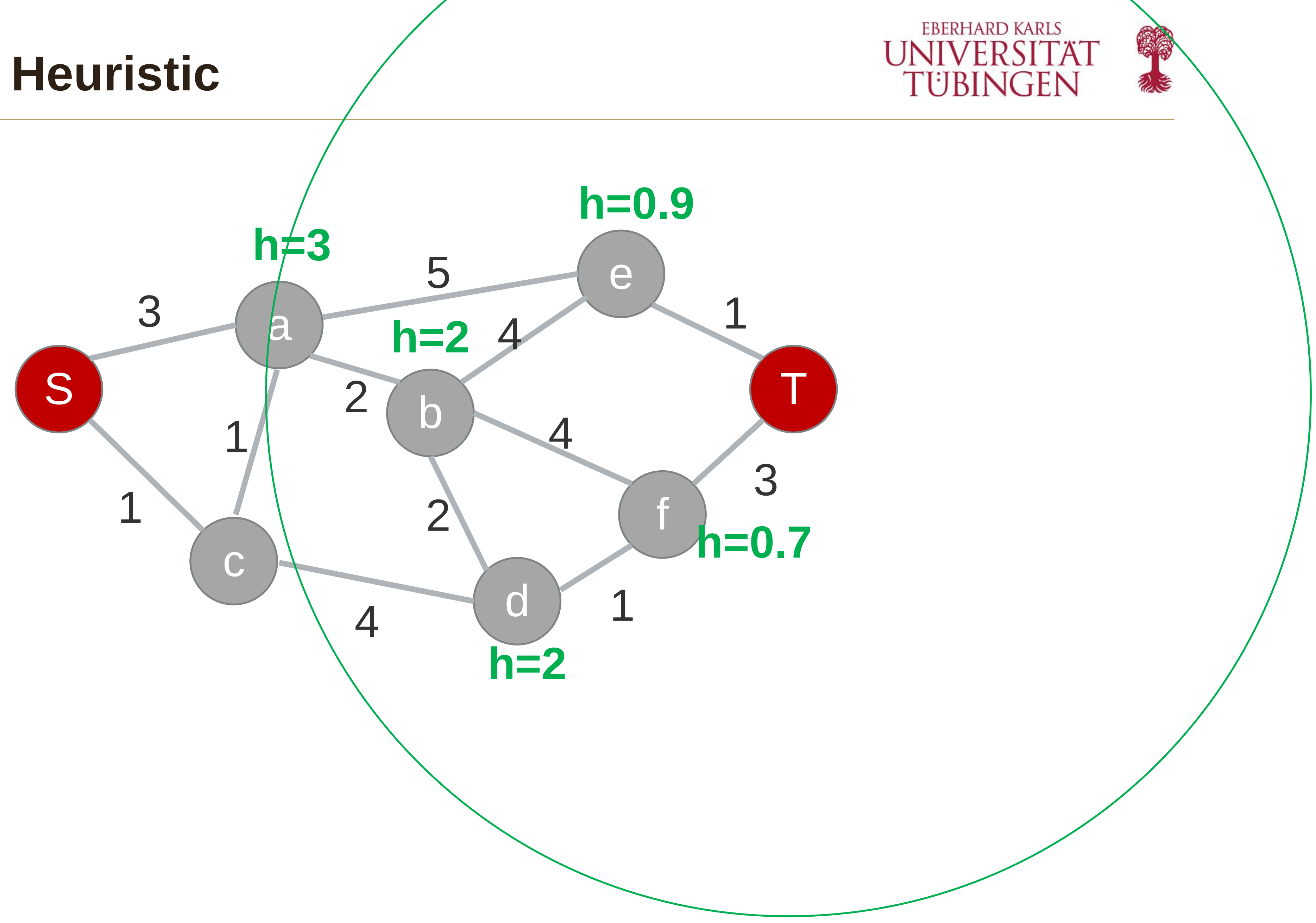
A*-Algorithm – Heuristic

- Weighted graph



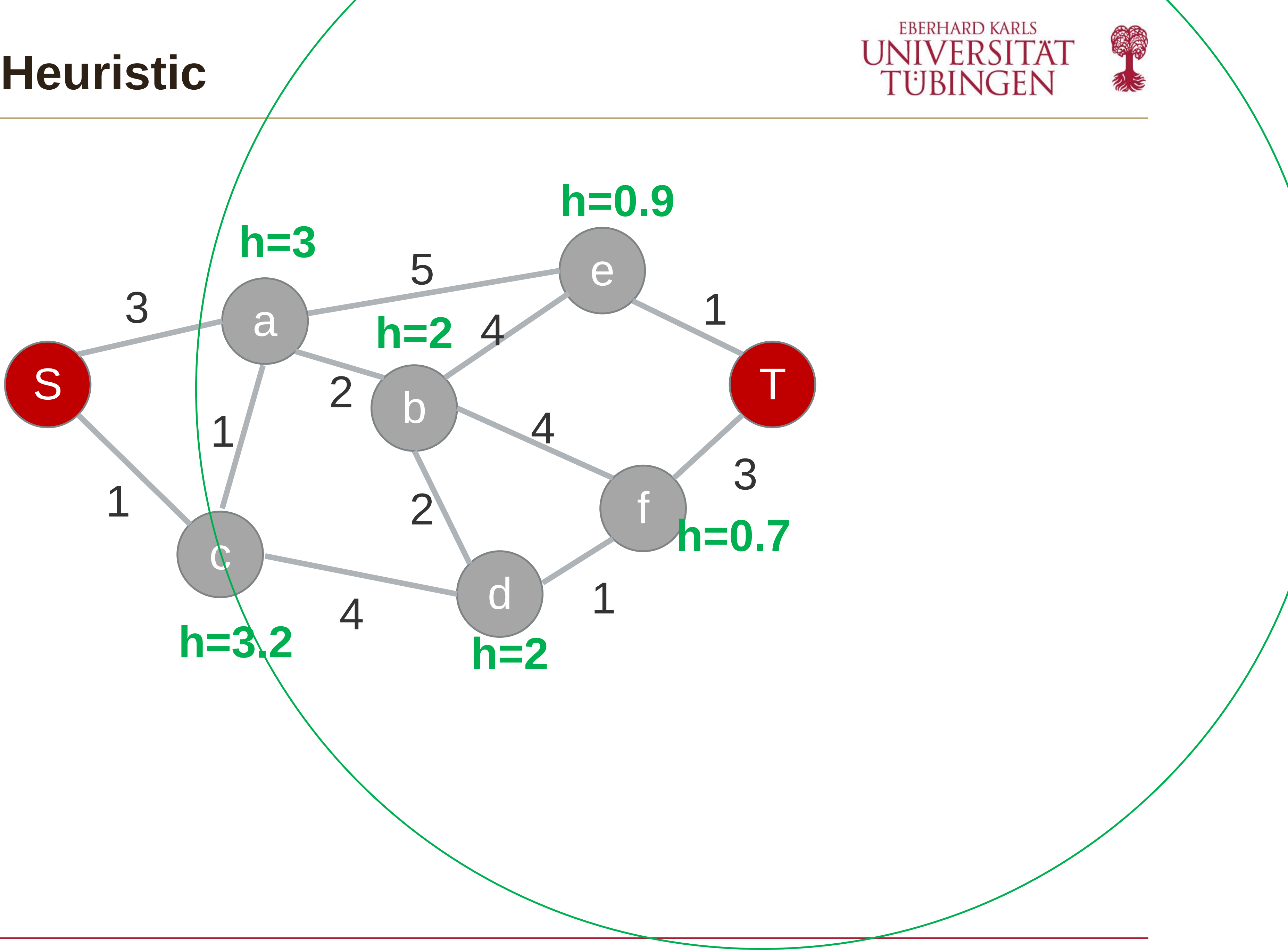
A*-Algorithm – Heuristic

- Weighted graph



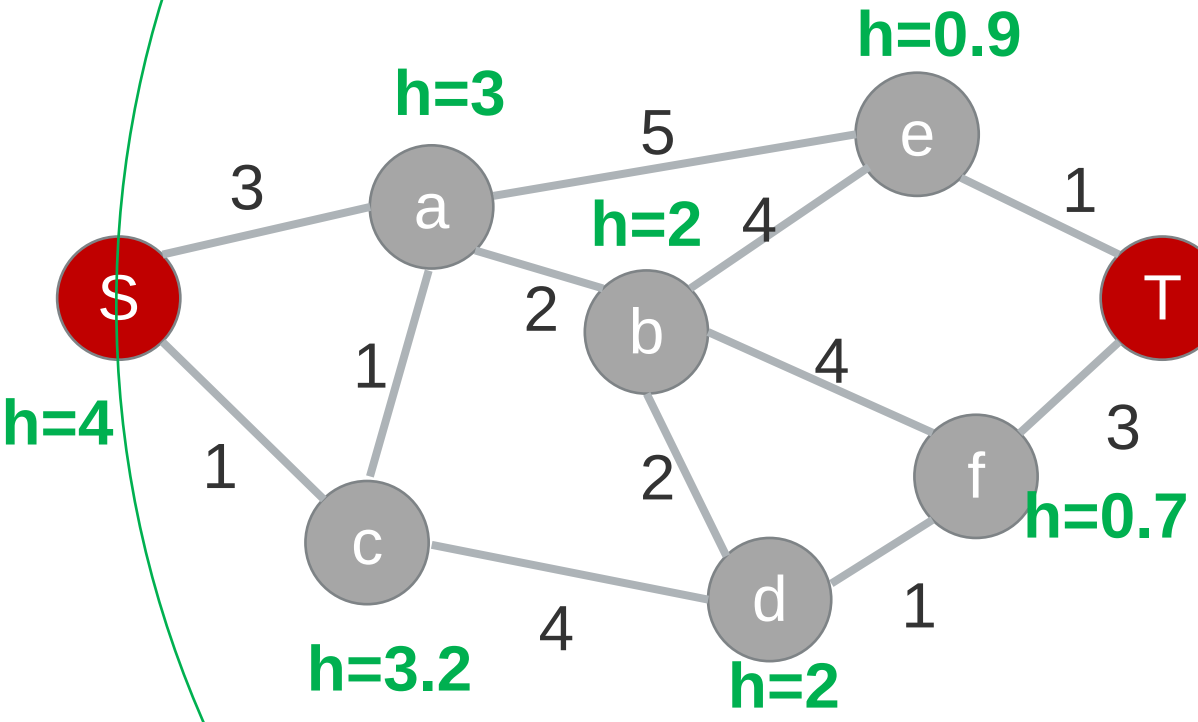
A*-Algorithm – Heuristic

- Weighted graph



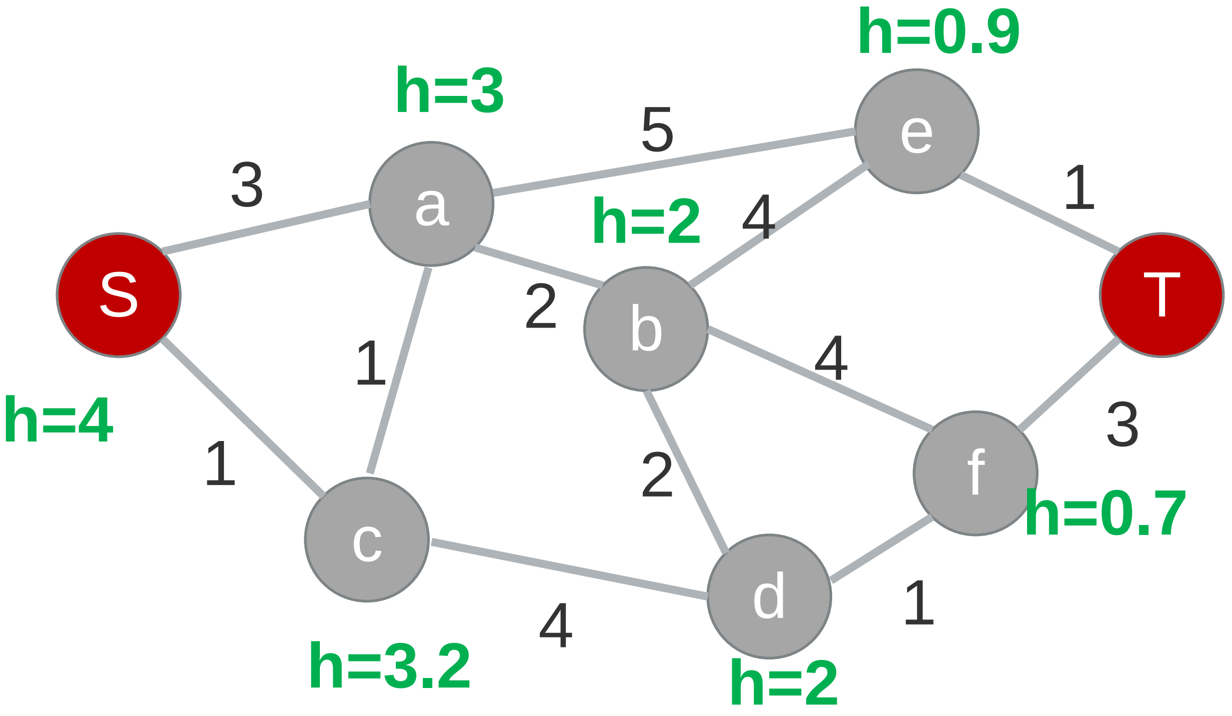
A*-Algorithm – Heuristic

- Weighted graph

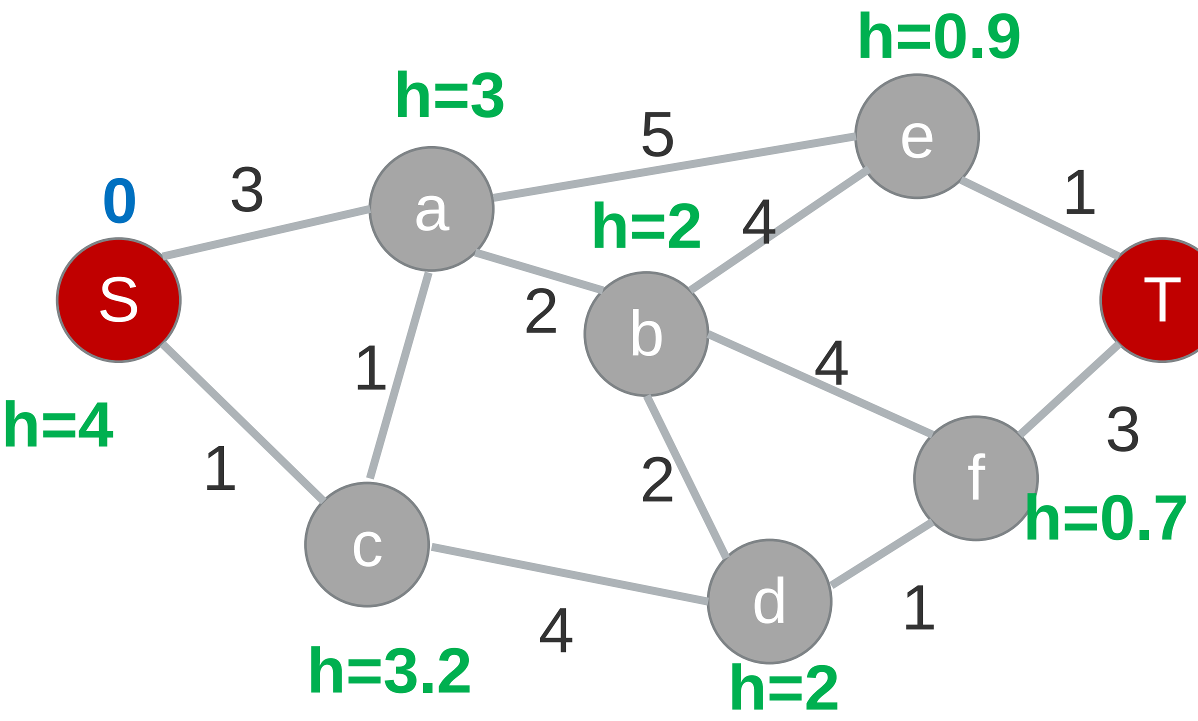


A*-Algorithm – Heuristic

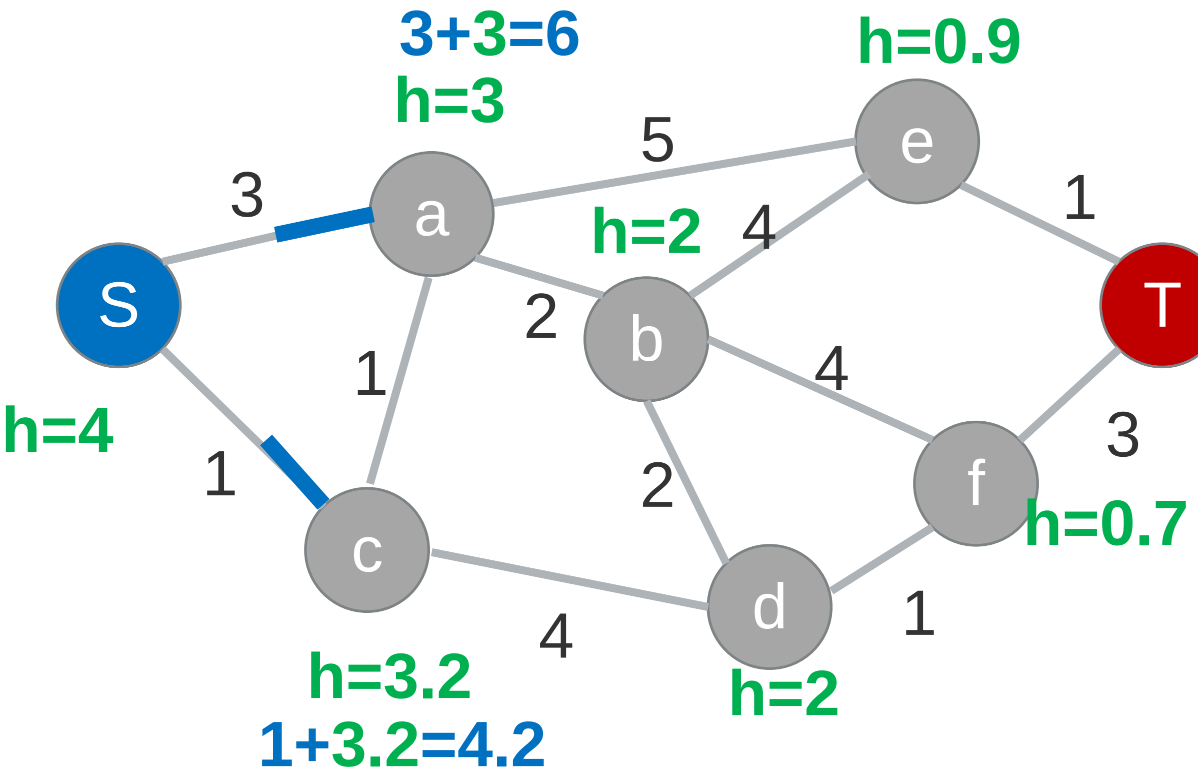
- Weighted graph



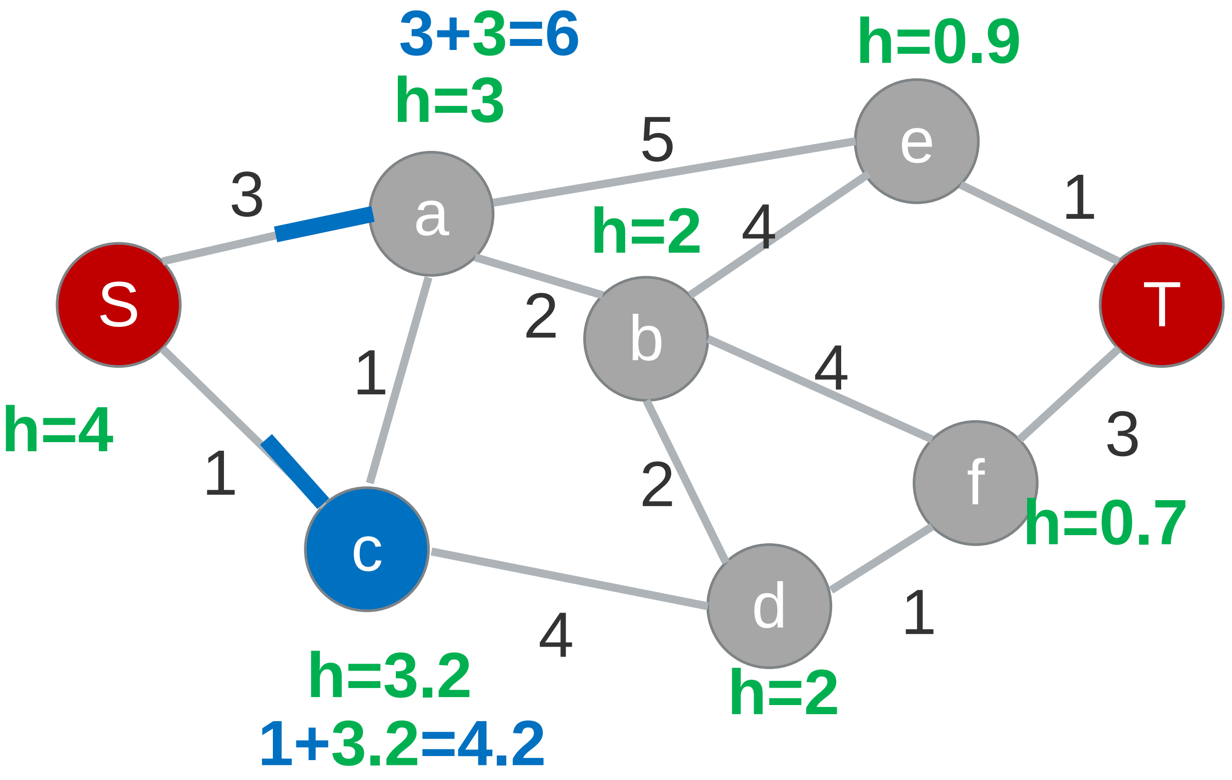
A*-Algorithm



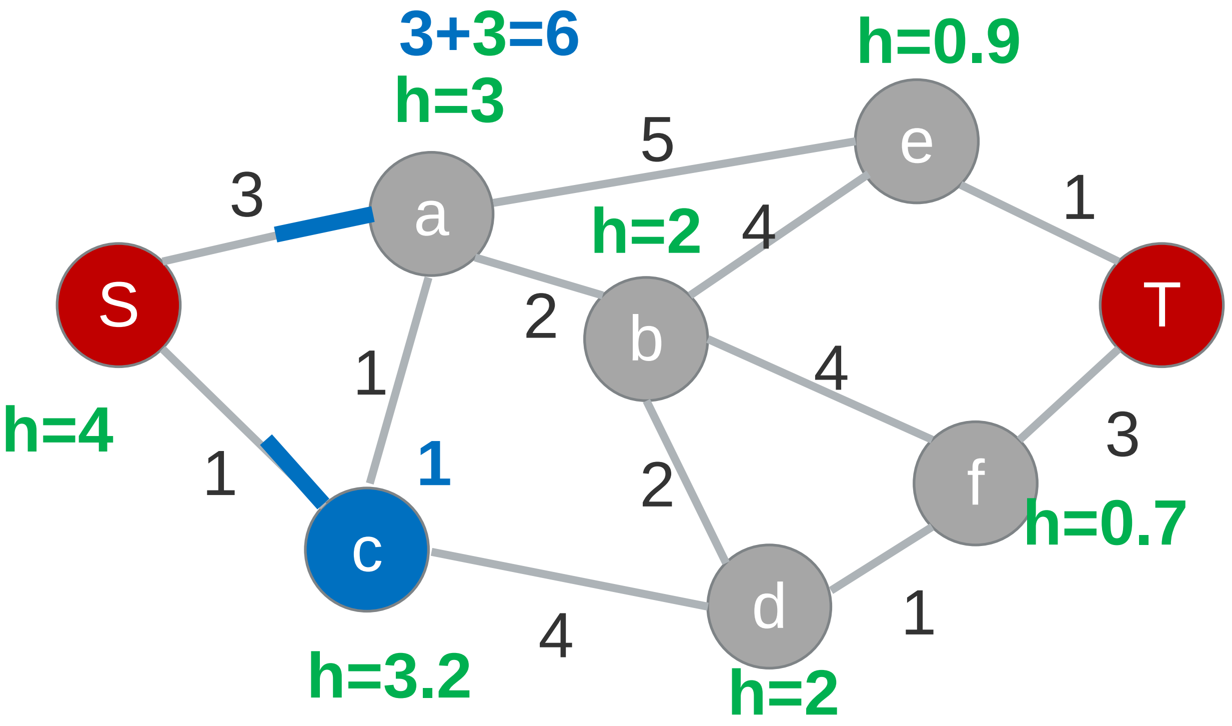
A*-Algorithm



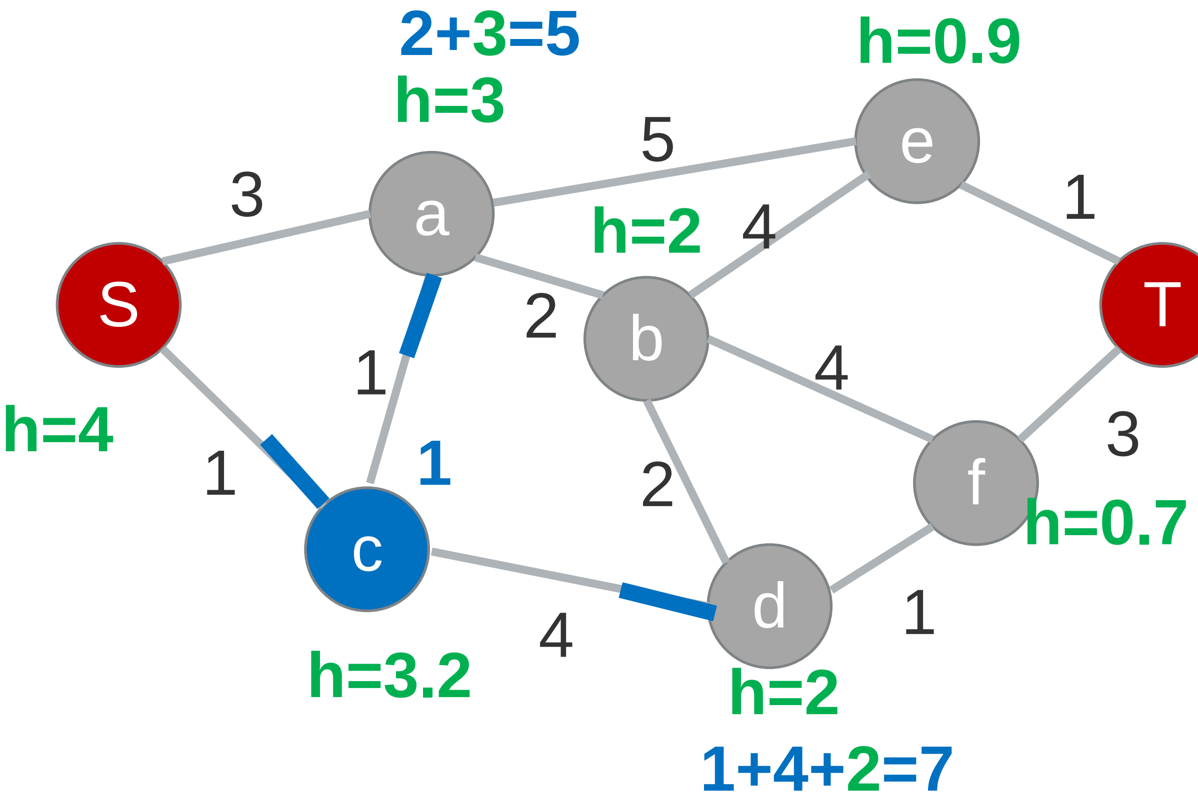
A*-Algorithm



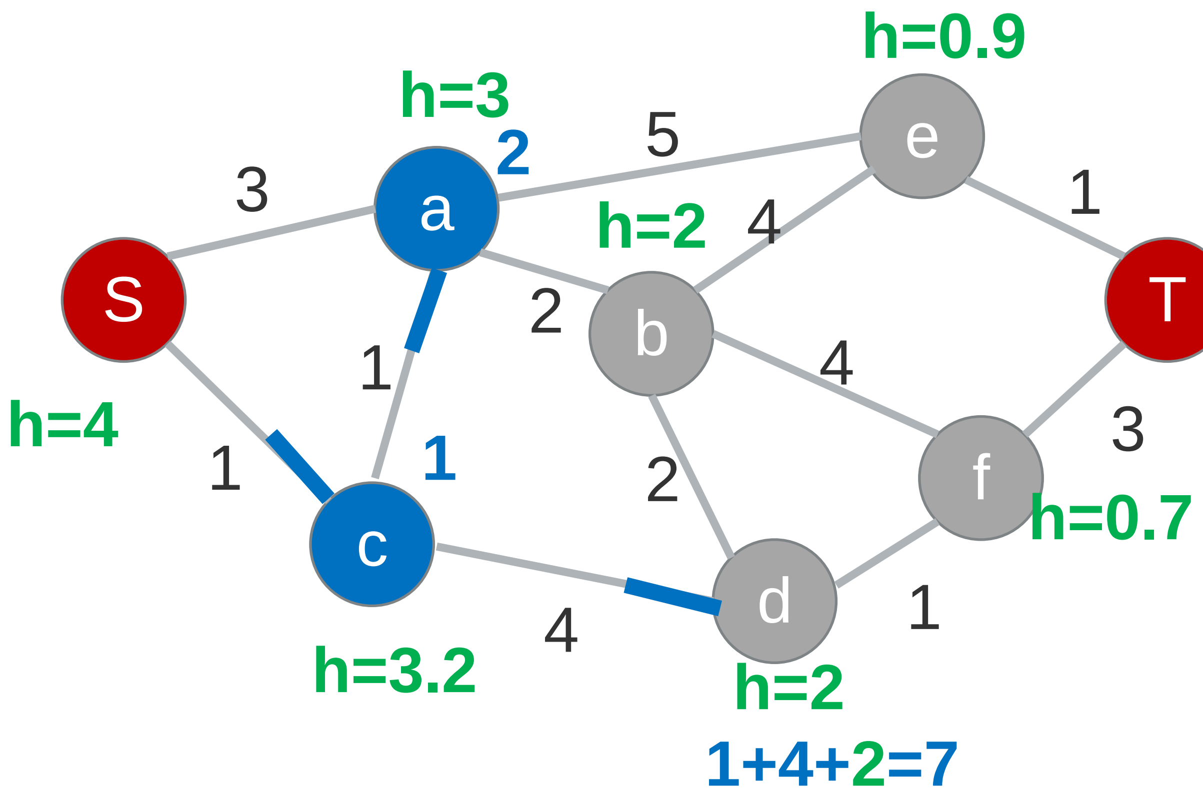
A*-Algorithm



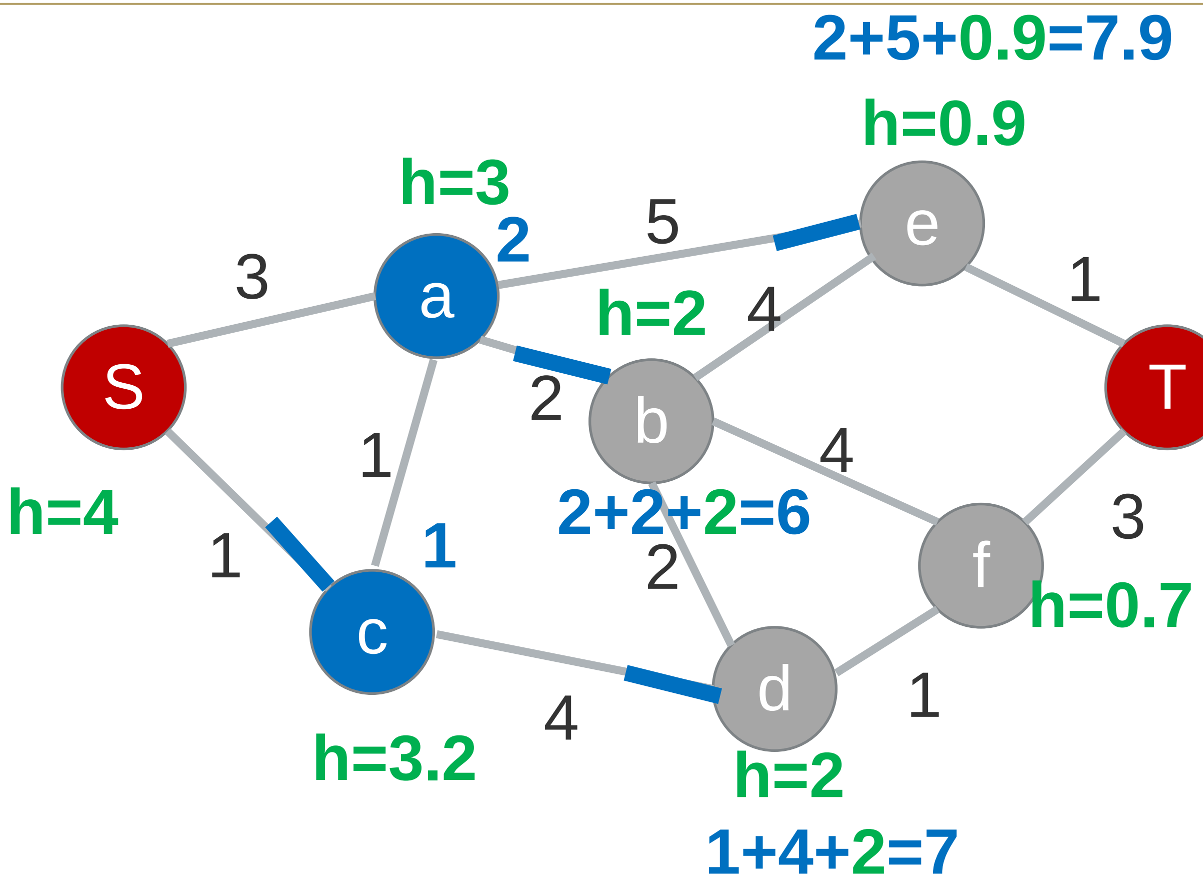
A*-Algorithm



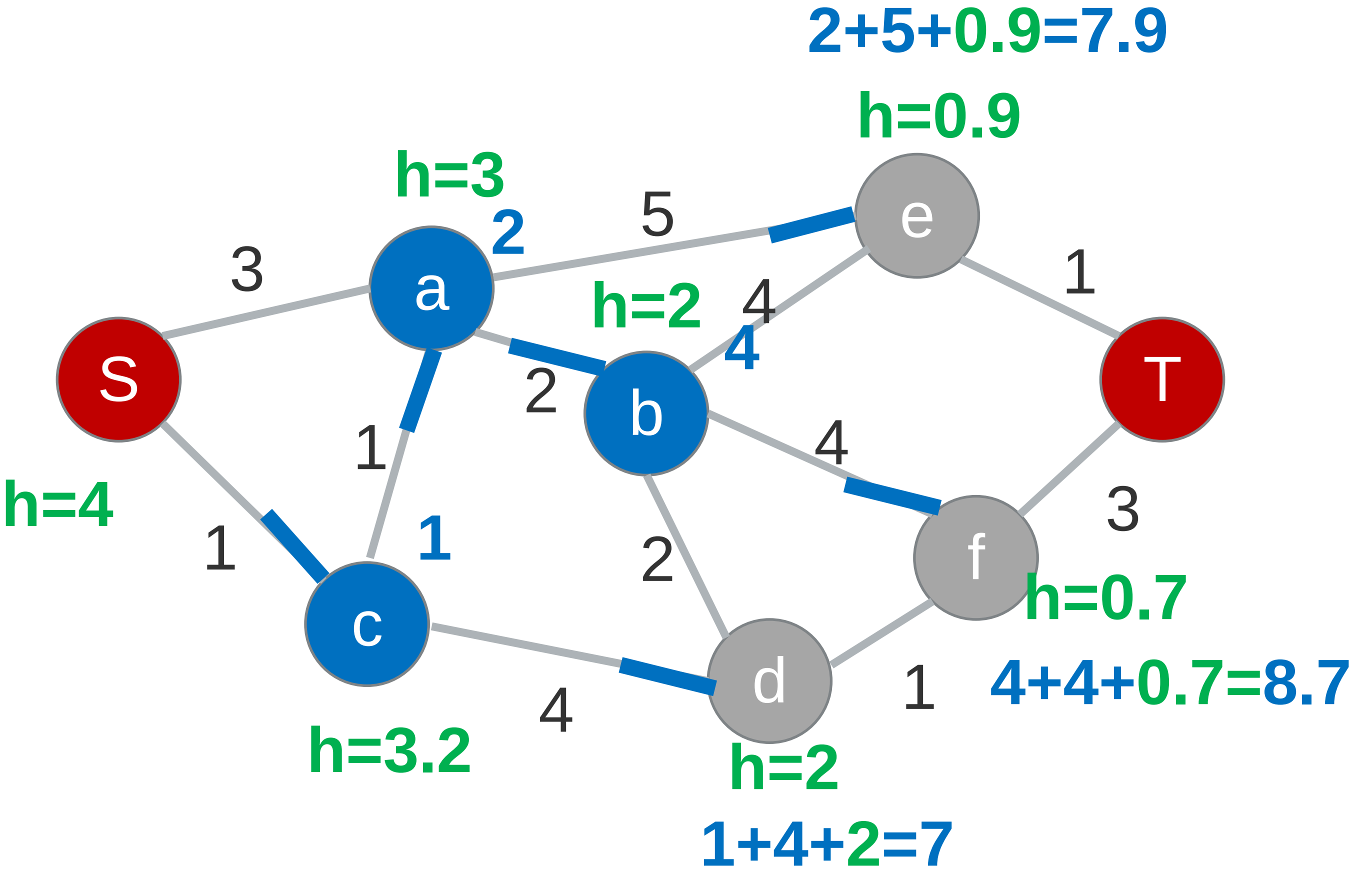
A*-Algorithm



A*-Algorithm

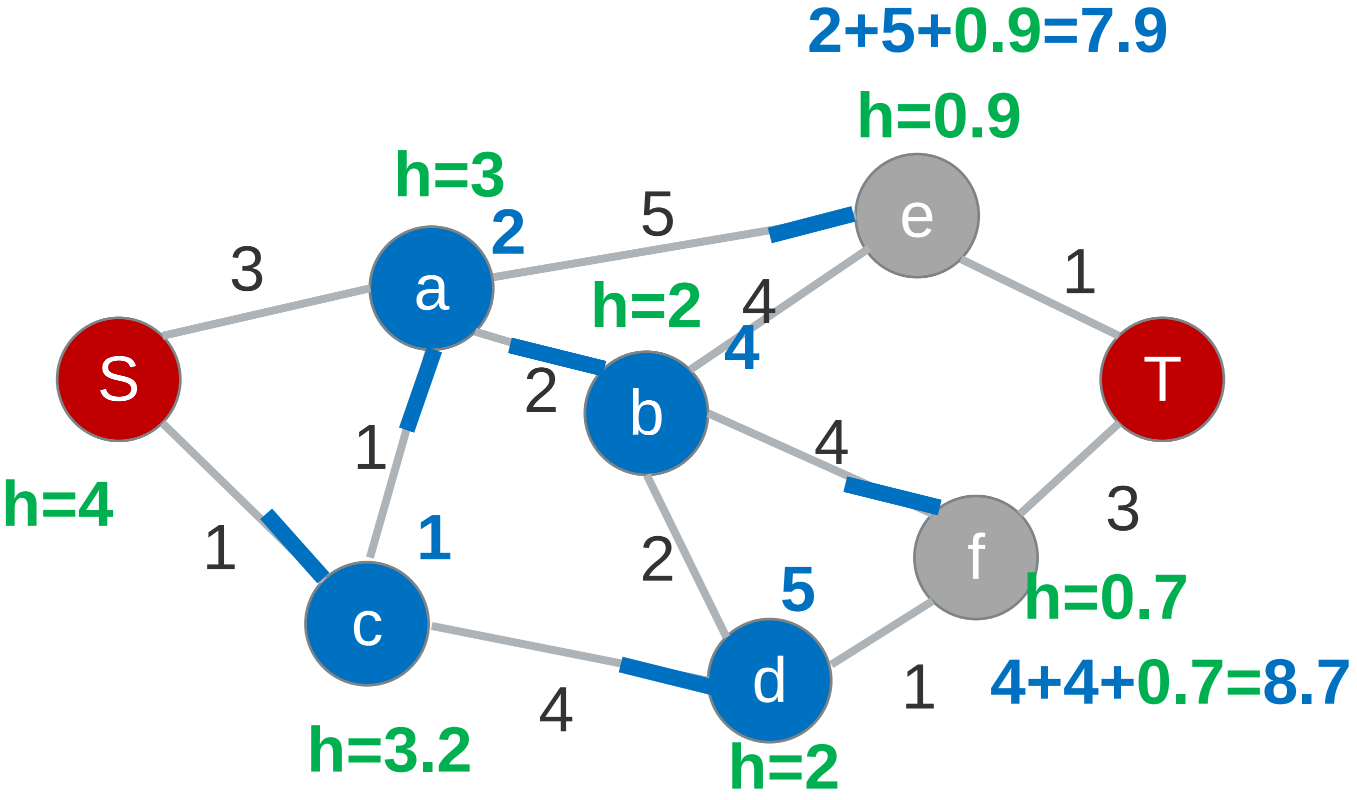

2+5+0.9=7.9
h=0.9
h=3
2
5
e
3
a
1
h=2
4
S
T
2
b
1
4
h=4
2+2+2=6
1
1
2
3
f
h=0.7
c
d
4
1
h=3.2
h=2
1+4+2=7

A*-Algorithm



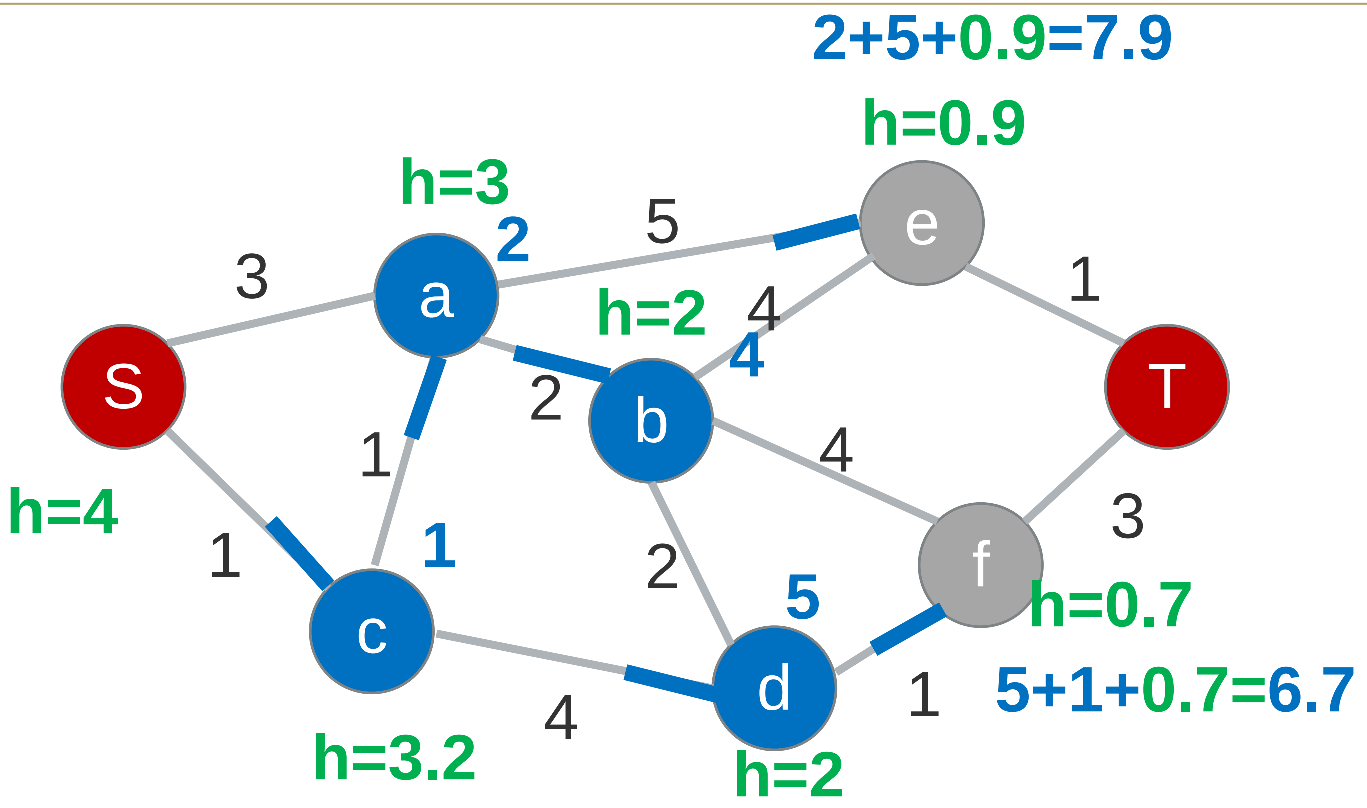
2+5+0.9=7.9
h=0.9
h=3
2
5
e
3
a
h=2
4
1
4
S
2
b
T
1
4
h=4
1
1
2
3
f
c
h=0.7
4
d
1
4+4+0.7=8.7
h=3.2
h=2
1+4+2=7

A*-Algorithm



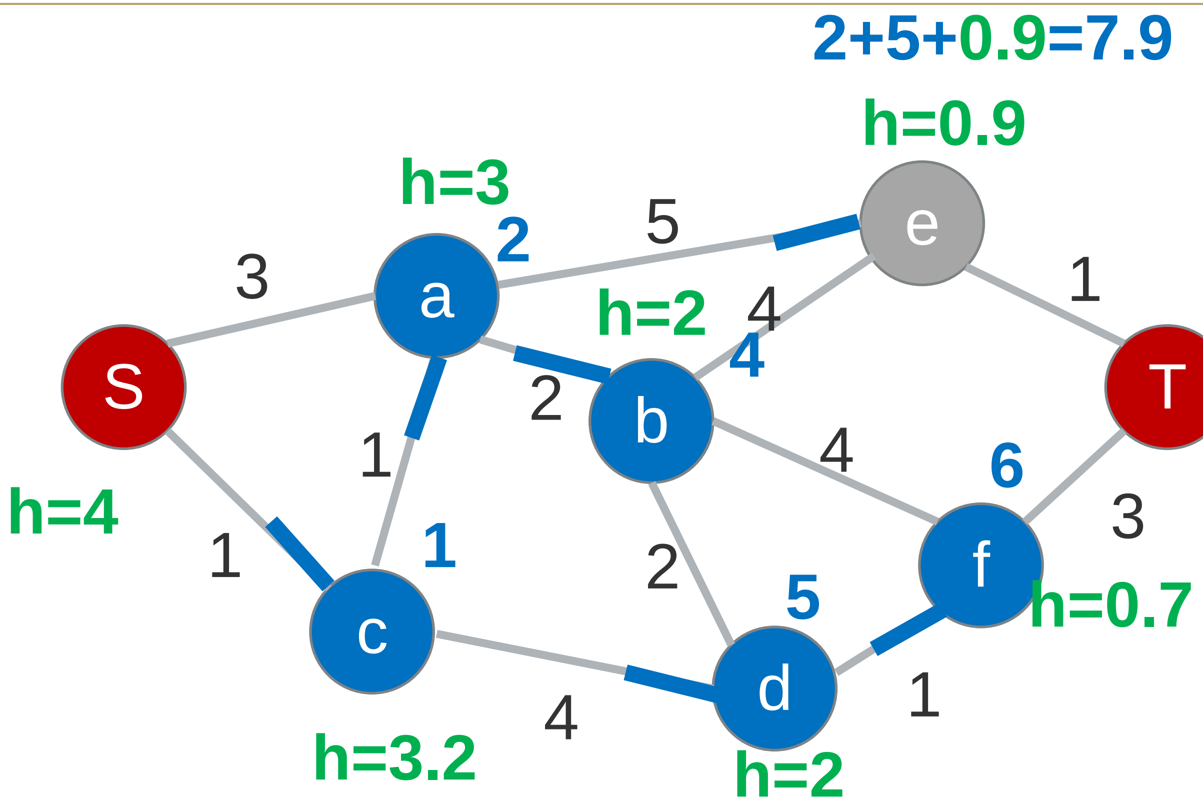
2+5+0.9=7.9
h=0.9
h=3
2
5
e
3
a
1
h=2
4
4
S
T
2
b
1
4
h=4
3
1
1
f
2
5
h=0.7
c
d
1
4+4+0.7=8.7
4
h=3.2
h=2

A*-Algorithm

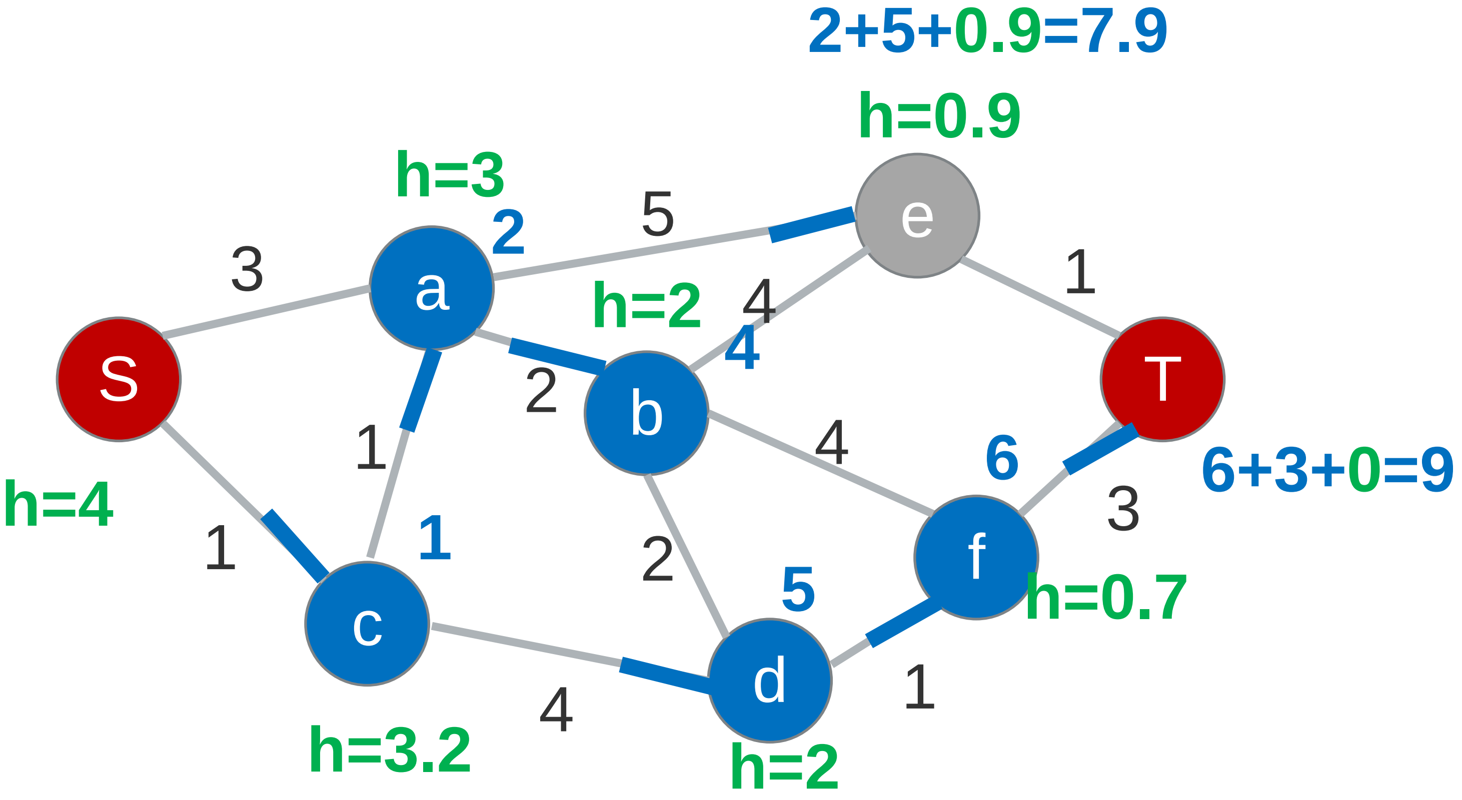


2+5+0.9=7.9
h=0.9
h=3
2
5
e
3
a
1
h=2
4
4
S
2
b
T
1
4
h=4
1
1
2
3
f
5
h=0.7
c
d
4
1
5+1+0.7=6.7
h=3.2
h=2

A*-Algorithm

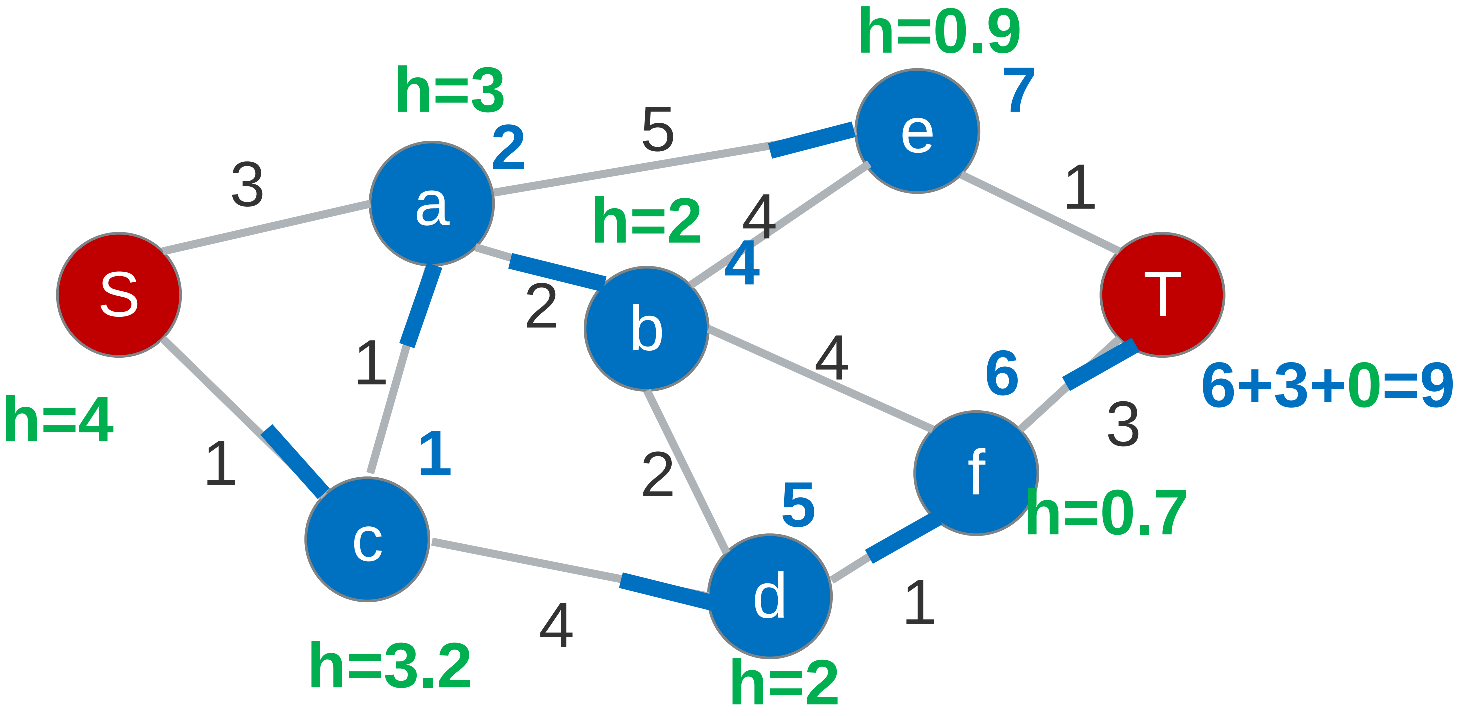


A*-Algorithm

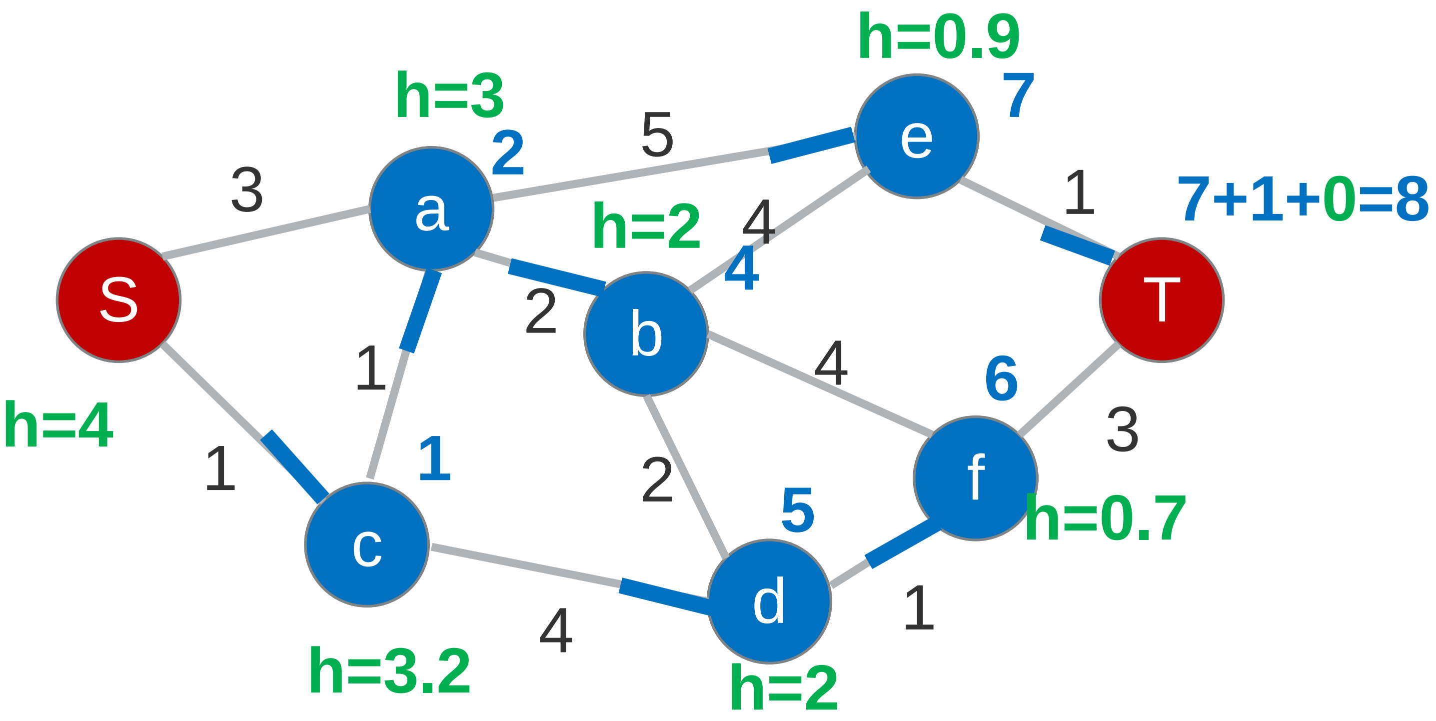


2+5+0.9=7.9
h=0.9
h=3
2
5
e
3
a
1
h=2
4
4
S
2
b
T
1
4
6
6+3+0=9
h=4
1
1
2
3
f
5
h=0.7
c
d
1
4
h=3.2
h=2

A*-Algorithm



A*-Algorithm



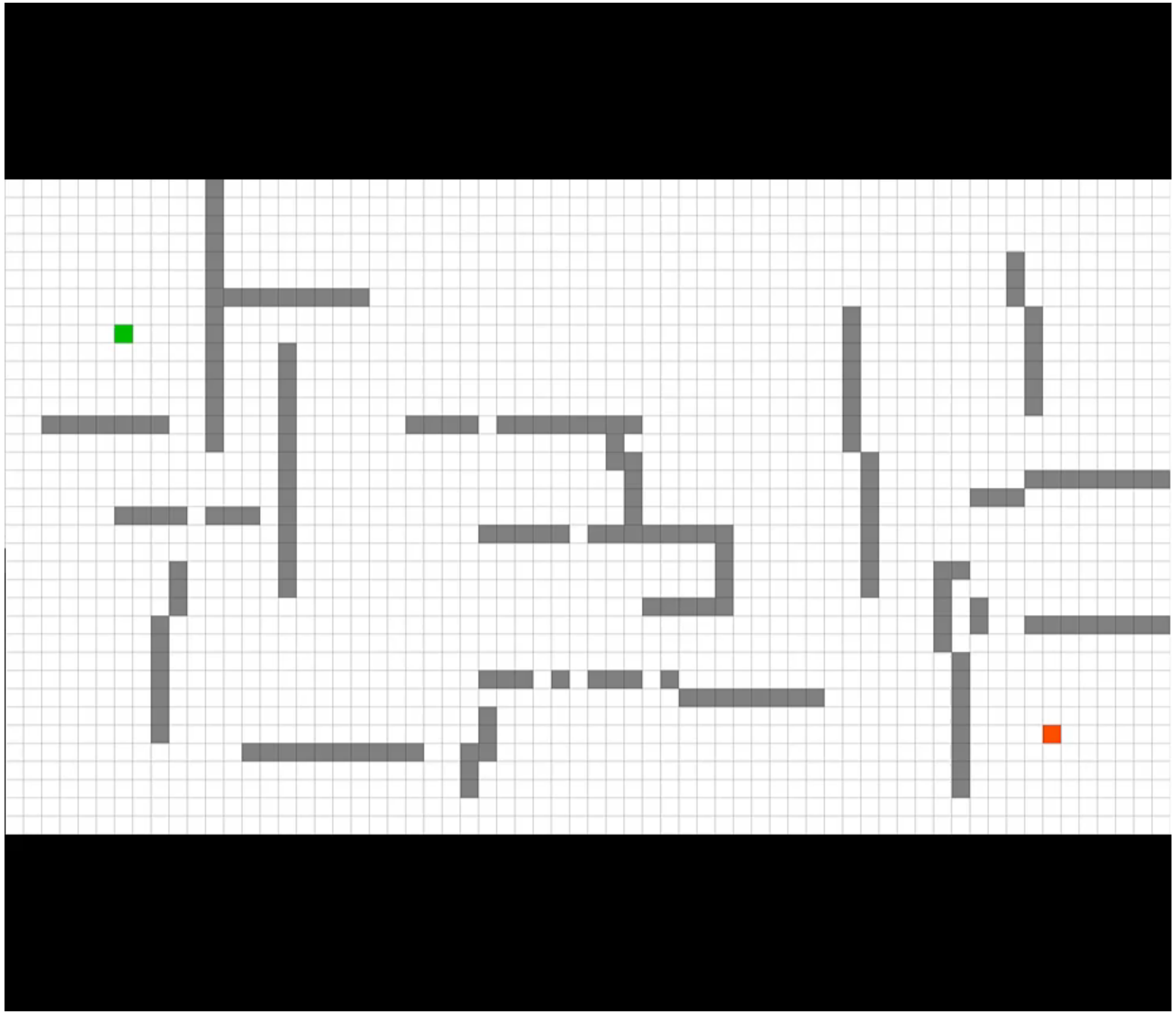
A*-Algorithm - Discussion

The A* algorithm is

- **complete**: it finds a solution if a path exists
- **optimal**: it finds a shortest path, if there are several shortest paths of the same length, it finds one of them
- **optimally efficient**: for the given heuristic there is no other algorithm that finds the solution faster

- https://www.boost.org/doc/libs/1_80_0/libs/graph/example/astar-cities.cpp

A* visits significantly fewer sites



Summary

- Shortest path algorithm
  - Dijkstra
  - Dynamic programming

- A*-Algorithm
  - Shortest path with heuristics

Concluding the Lecture

We covered a lot of ground in this intensive block course.

What didn't we cover?
A lot! … and there is more coming in new C++ standards (theory) or becomes useable with better compiler support (practice).

We hope that the course prepared you to dive deeper into C++.

That you, if needed, are now able to read into the technical C++ documentation on topics we couldn't cover and apply what you learned in your projects.

Thanks

to the TAs for their help

to you for listening