Algorithms and Data Structures

Graph search algorithms

dr Szymon Murawski

Comarch SA

May 23, 2019

Table of contents I

Course plan

Graph search

- Many problems regarding graph correspond to finding a path in graph given starting node and end criteria
- Algorithms might differ in how fast they return an answer and if they return an optimal answer or not
- Applications:
  - Route planning
  - Solving a maze
  - Selecting a move in game
  - Proposing friends on facebook
  - Building index of websites by crawlers in search engines

Course plan

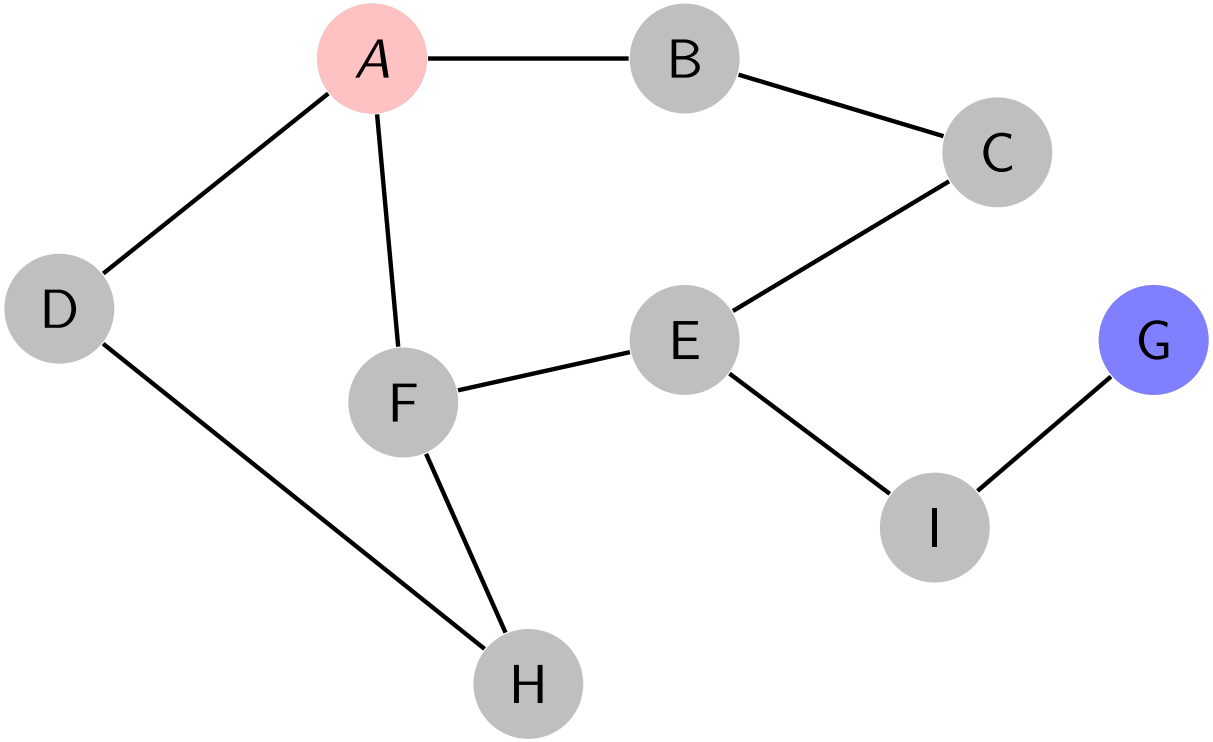
Depth-First Search

- In Depth-First Search (DFS) algorithm we follow one path as deep as possible, before exploring other options
- In different terms we first traverse children of node instead of its siblings
- DFS can produce an answer rather quickly, but it might be not the optimal solution
- For DFS all vertices must have an additional property `isVisited` initially set to 0, to avoid visiting the same node multiple times (in case of cycle in graph)
- We maintain a stack of vertices to visit
- As we travel to new vertex, we add all of its unvisited neighbours to the stack

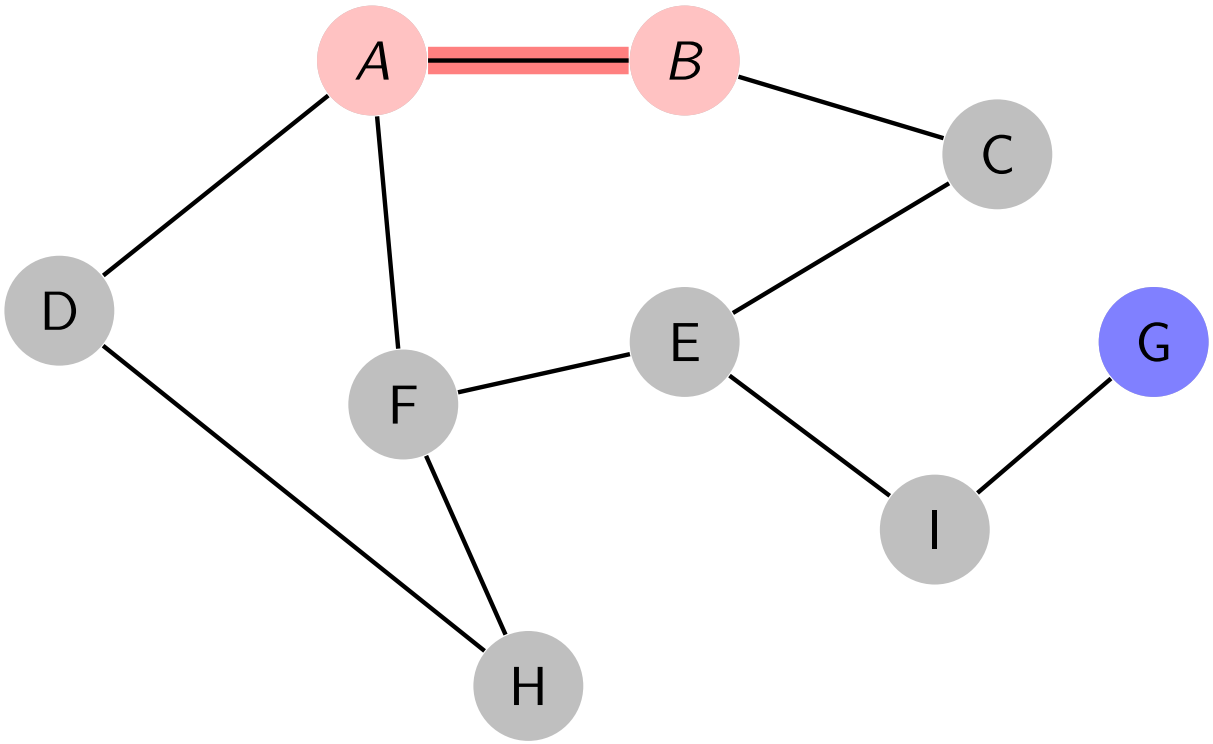
DFS pseudocode

```
1  DFS(Graph, startVertex, endVertex)
2    for each vertex in Graph
3      vertex.isVisited = 0;
4    new Stack
5    Stack.push(startVertex)
6    while(Stack is not empty)
7      vertex = Stack.pop()
8      if vertex == endVertex
9        return
10     vertex.isVisited = 1
11     foreach neighbourVertex in vertex.neighbours
12       Stack.push(neighbourVertex)
```

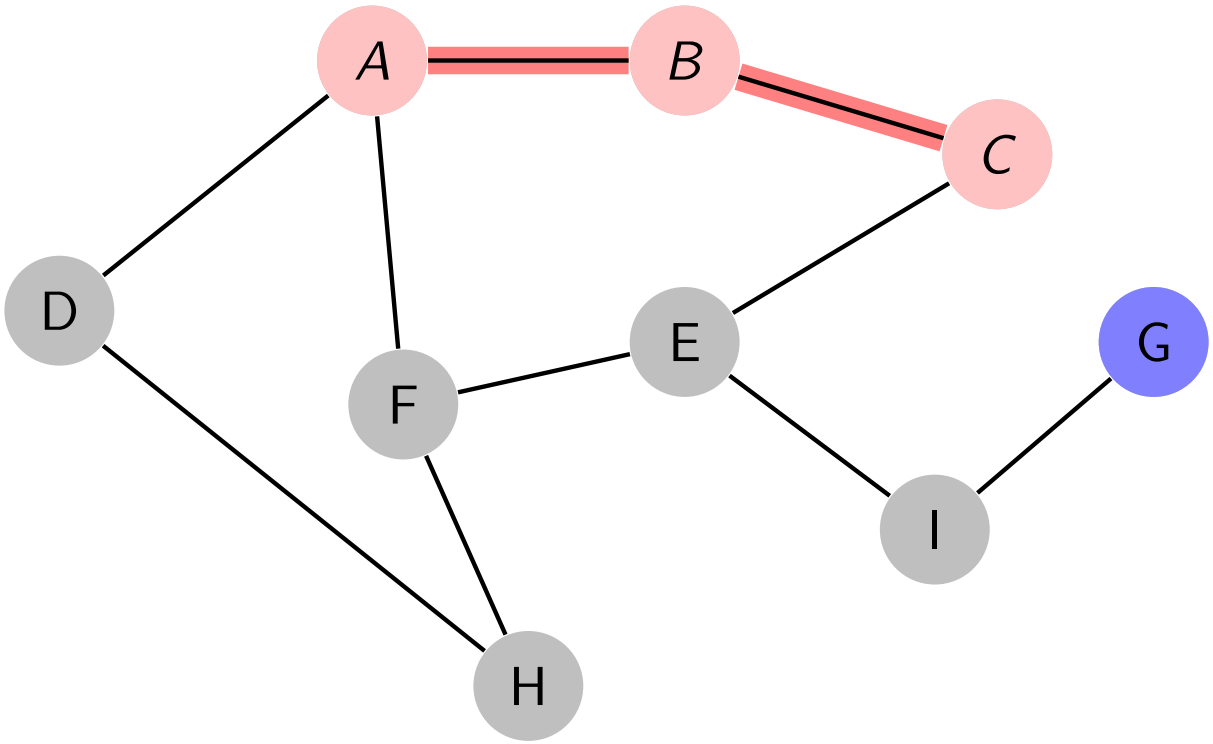
Depth-First Search example



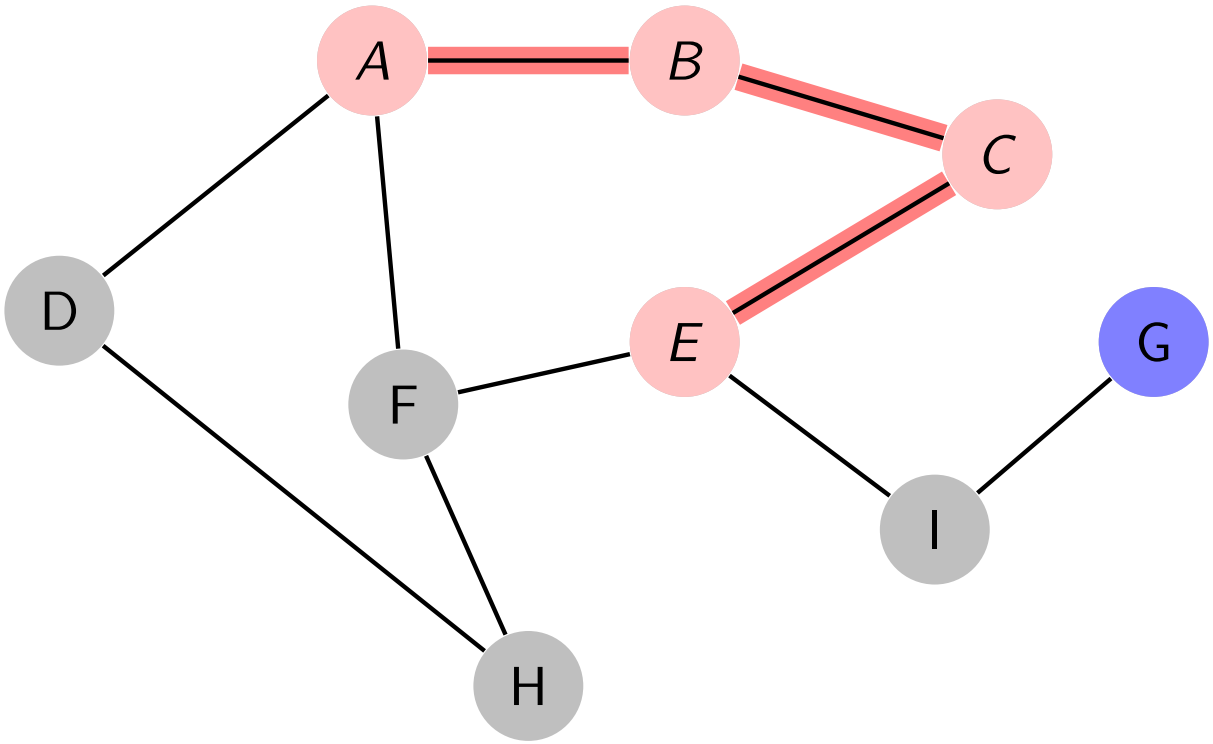
Depth-First Search example



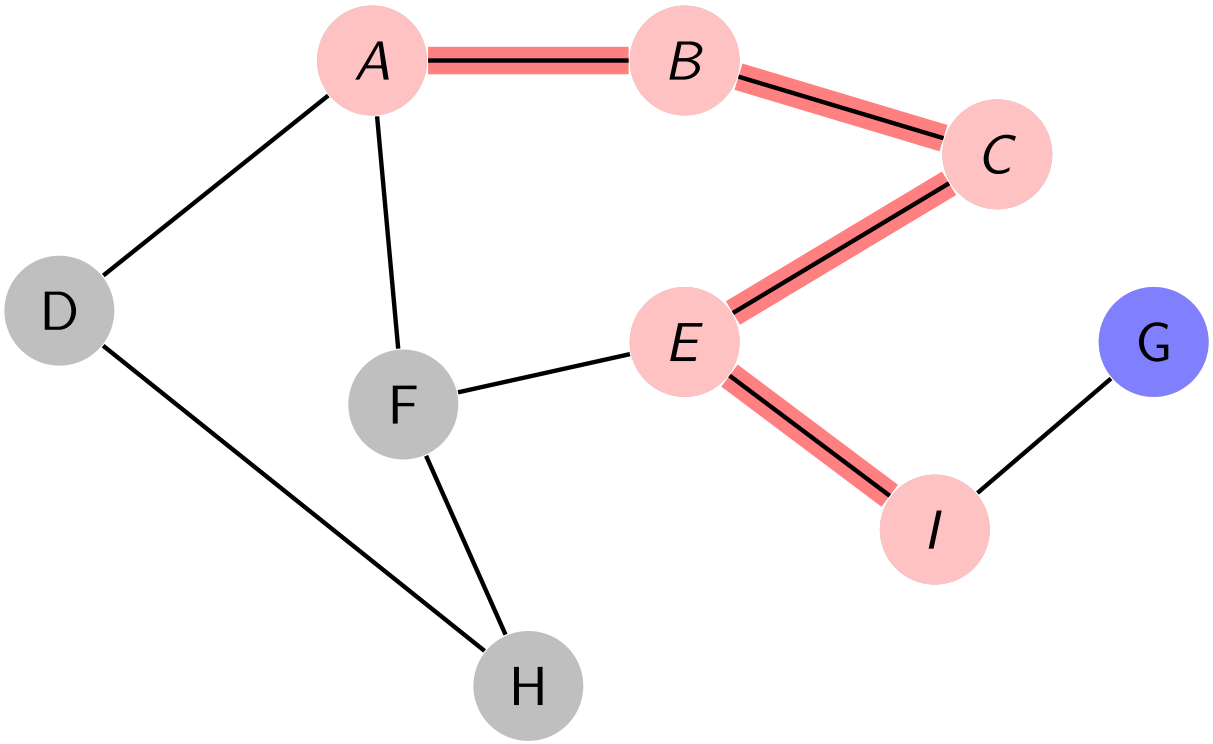
Depth-First Search example



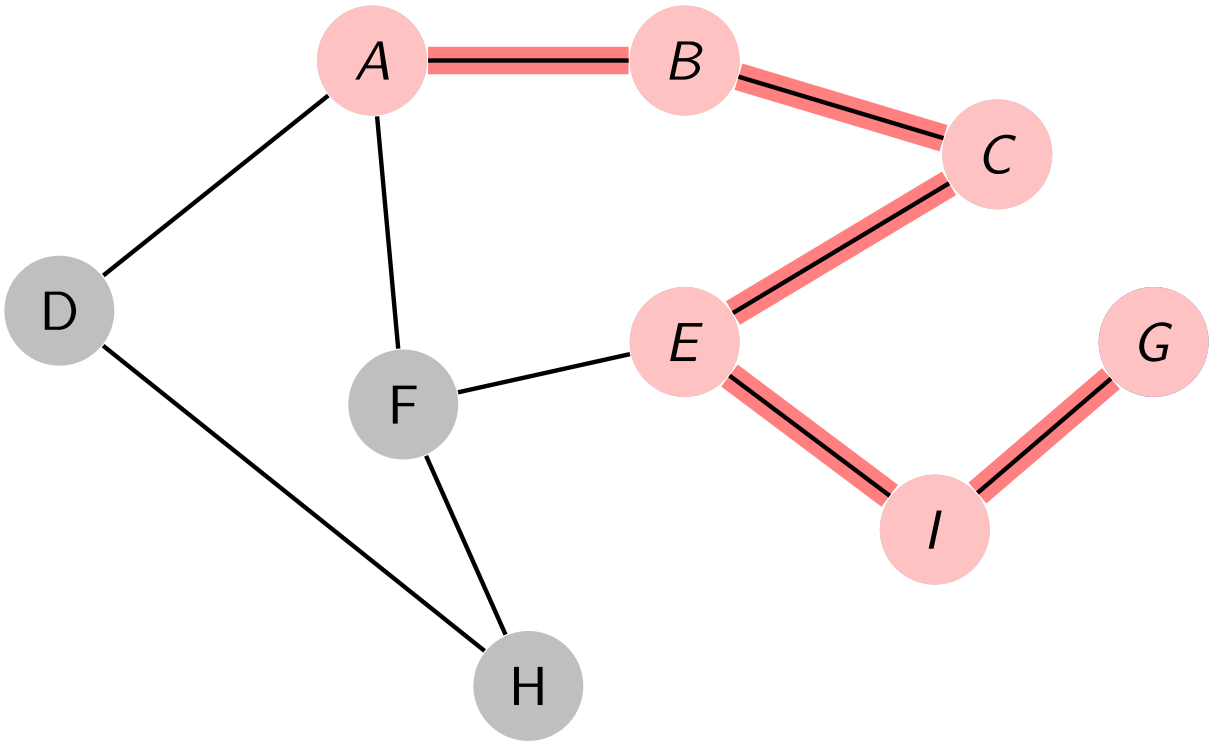
Depth-First Search example



Depth-First Search example



Depth-First Search example



DFS with path return pseudocode

- If we store path in Stack alongside vertices, we can output path to the end vertex

```
1  DFS(Graph, startVertex, endVertex)
2    for each vertex in Graph
3      vertex.isVisited = 0;
4    new Stack
5    Stack.push((startVertex, "startVertex"))
6    while(Stack is not empty)
7      (vertex, path) = Stack.pop()
8      if vertex == endVertex
9        return path
10     vertex.isVisited = 1
11     path += ", " + vertex
12     foreach neighbourVertex in vertex.neighbours
13       Stack.push((neighbourVertex, path))
```

Course plan

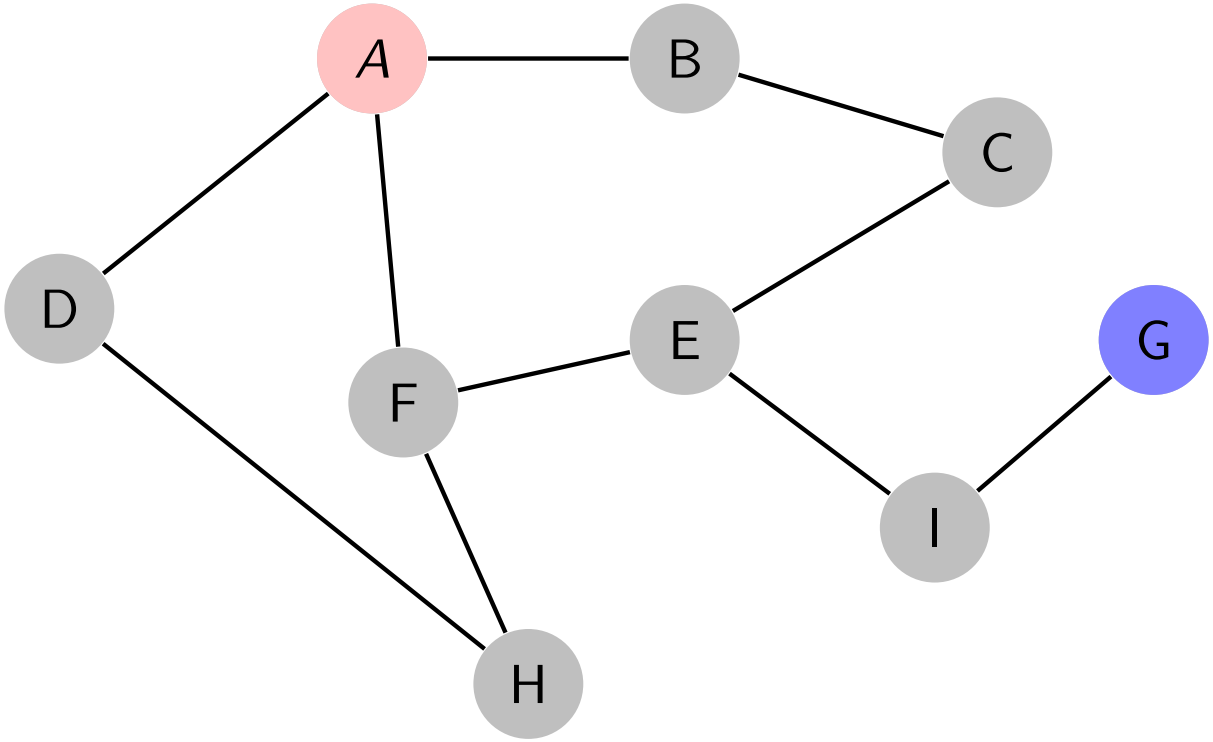
Breadth-First Search

- In Breadth-First Search (BFS) algorithm we first explore all neighbours of a vertex before going down to next level
- In different terms we first traverse siblings of node instead of its children
- It produces optimal result, but might take a longer time than DFS
- For BFS all vertices must have an additional property `isVisited` initially set to 0, to avoid visiting the same node multiple times (in case of cycle in graph)
- We maintain a queue of vertices to visit
- As we travel to new vertex, we add all of its unvisited neighbours to the queue
- BFS is used in Cheney's algorithm for garbage collection

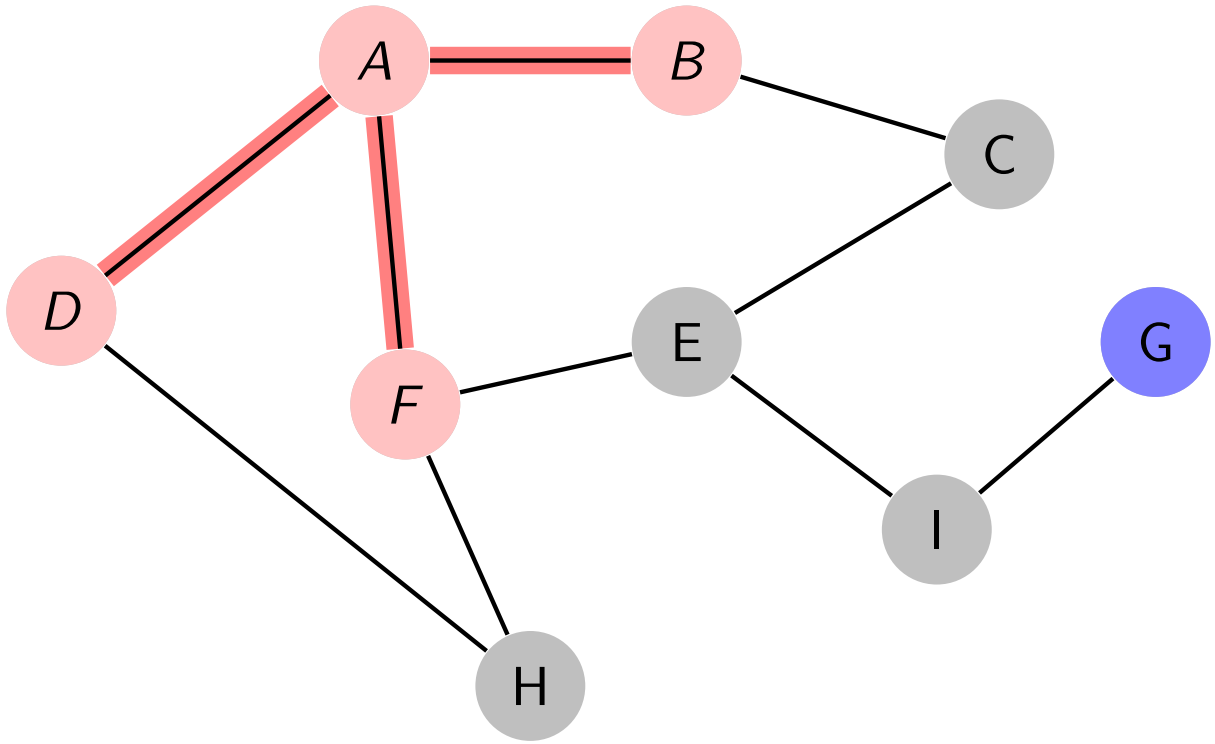
BFS pseudocode

```
1  BFS(Graph, startVertex, endVertex)
2    for each vertex in Graph
3      vertex.isVisited = 0;
4    new Queue
5    Queue.enqueue(startVertex)
6    while(Queue is not empty)
7      vertex = Queue.dequeue()
8      if vertex == endVertex
9        return
10     vertex.isVisited = 1
11     foreach neighbourVertex in vertex.neighbours
12       Queue.enqueue(neighbourVertex)
```

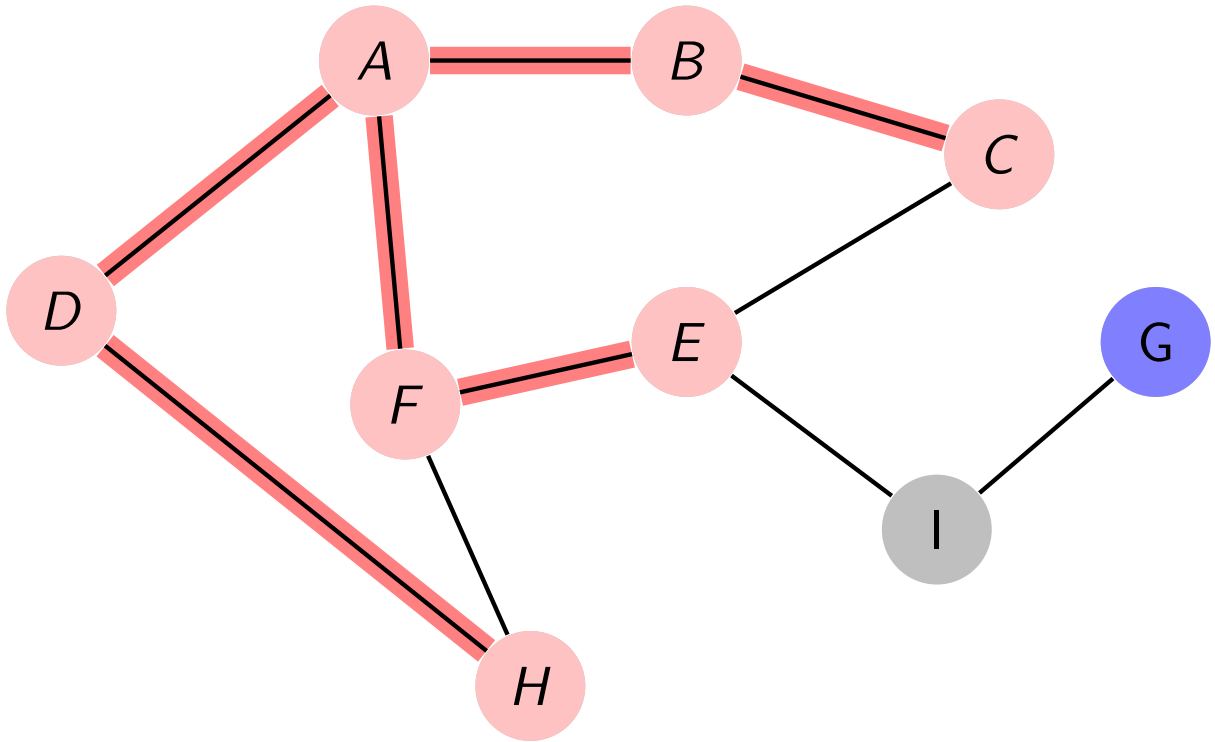
Breadth-First Search example



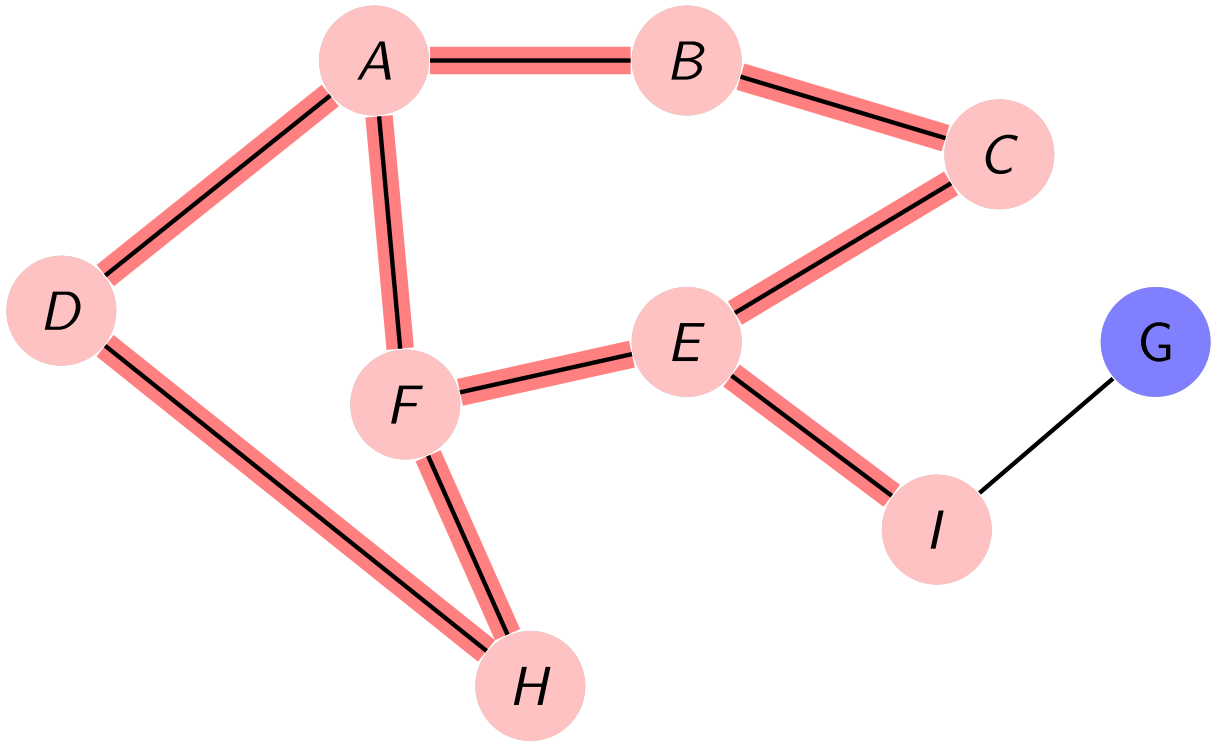
Breadth-First Search example



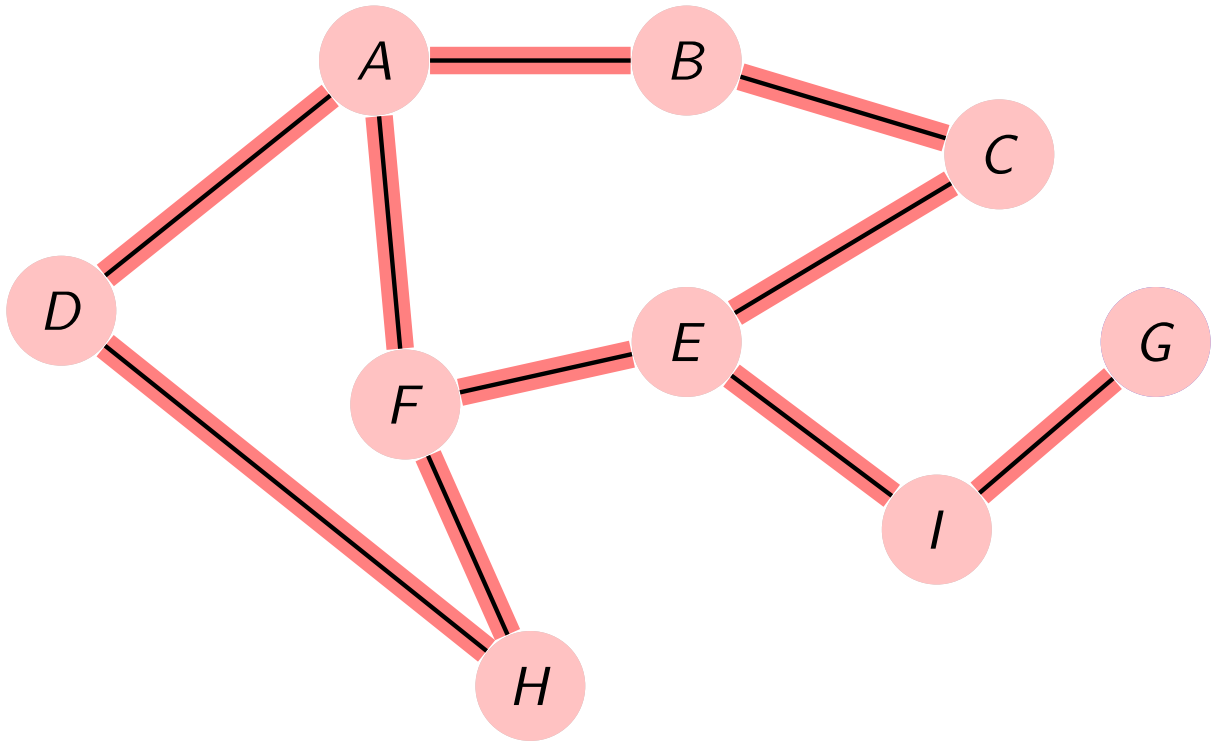
Breadth-First Search example



Breadth-First Search example



Breadth-First Search example



BFS with path return pseudocode

- If we store path in Stack alongside vertices, we can output path to the end vertex

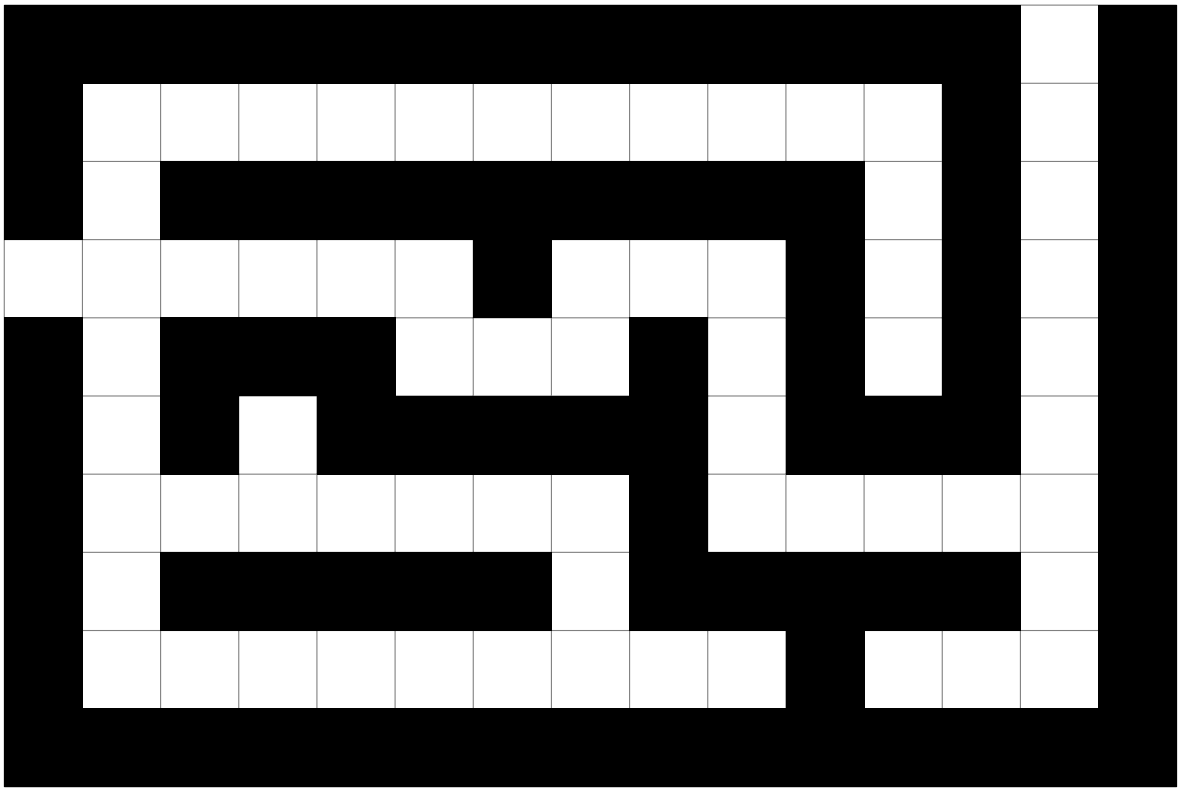
```
1  BFS(Graph, startVertex, endVertex)
2    for each vertex in Graph
3      vertex.isVisited = 0;
4    new Queue
5    Queue.enqueue((startVertex, "startVertex"))
6    while(Queue is not empty)
7      (vertex, path) = Queue.dequeue()
8      if vertex == endVertex
9        return path
10     vertex.isVisited = 1
11     path += ", " + vertex
12     foreach neighbourVertex in vertex.neighbours
13       Queue.enqueue((neighbourVertex, path))
```

DFS vs BFS

- Time complexity of both algorithms is the same
- BFS is more memory intensive - we need to store a lot more pointers than in DFS
- DFS uses stack to store vertices to visit, BFS uses queue
- If we mark all visited nodes, BFS will produce wide and short tree, while DFS narrow and long

Course plan

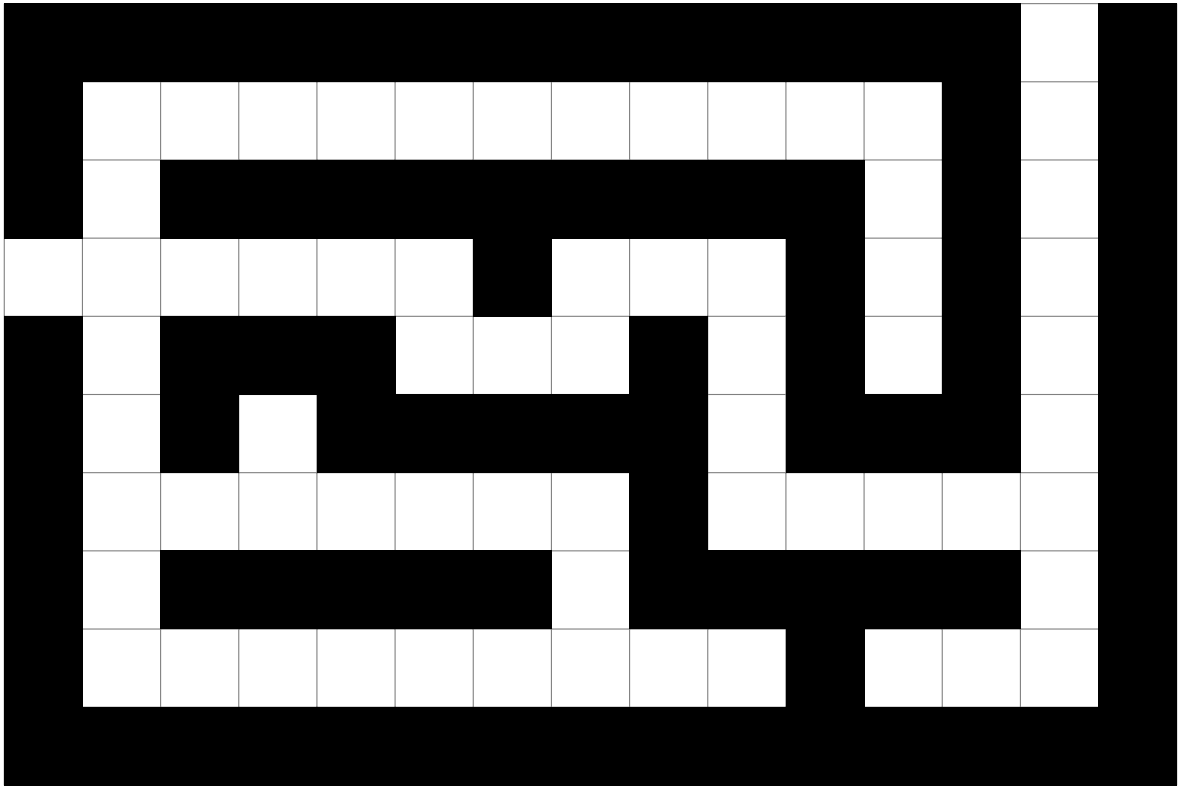
Maze



Graph maze solving

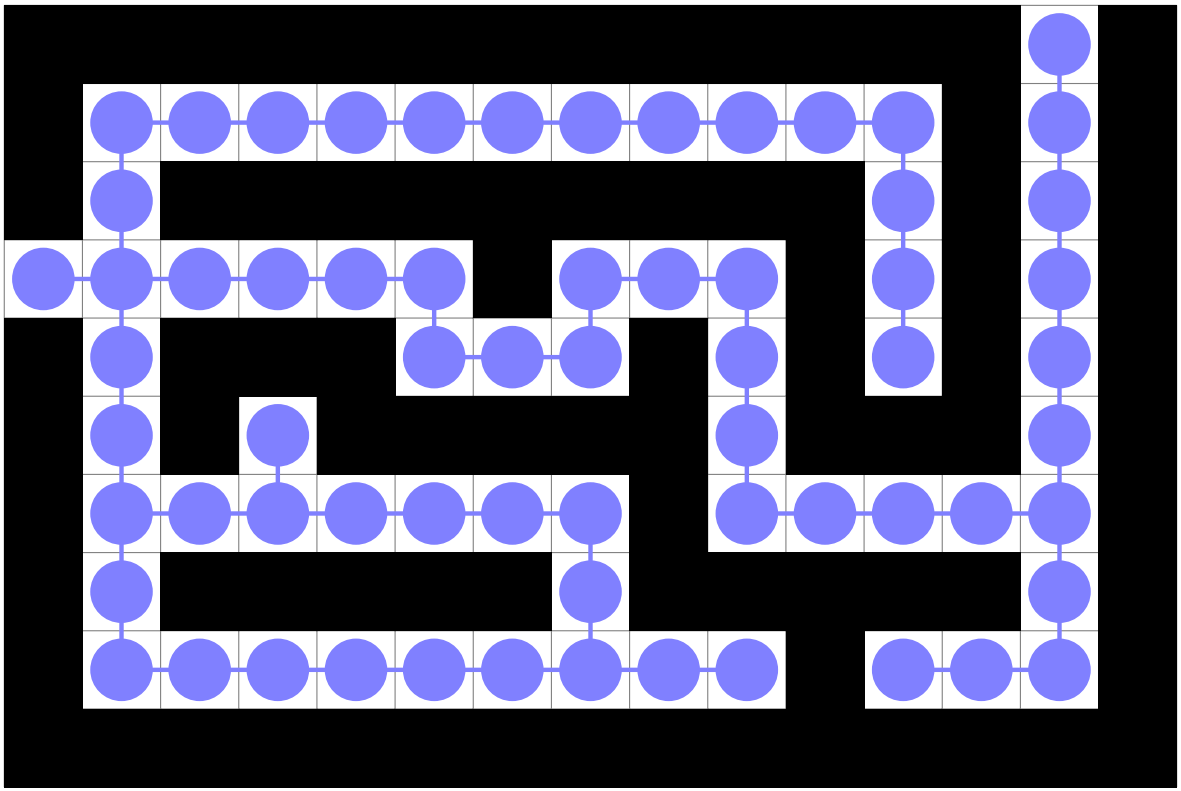
1. Assuming that black cells are walls, and white cells are paths, place a vertex at each white cell and connect it to all adjacent vertices
2. A vertex can now have the following number of neighbours:
   - **1** - it's either a dead end, or start/end of the maze
   - **2** - it's just connection between two other vertices
   - **3+** - it's a crossing
3. Remove all verties that have only two neighbours and instead just connect the neighbours
4. We now have a simple graph, where each node is either crossing, dead end, start or end of the maze
5. To find a route from start to end use BFS or DFS algorithms

Maze



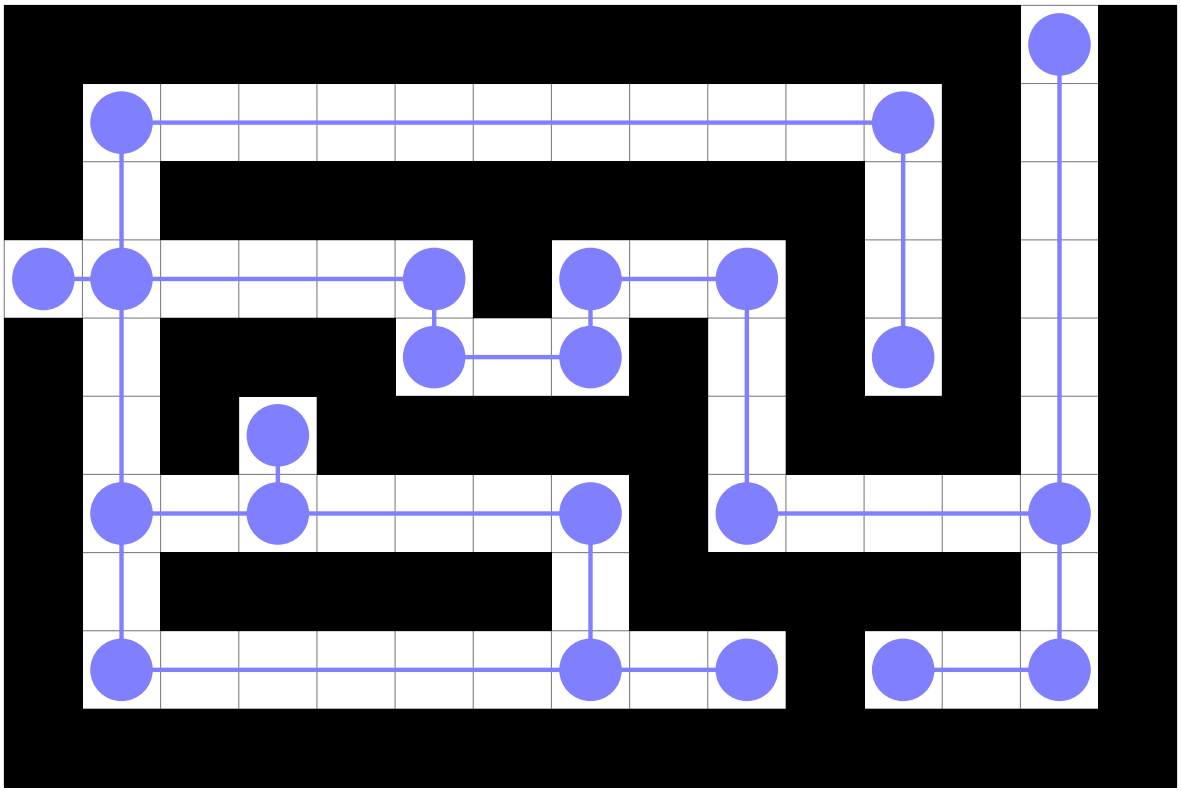
Print a maze

Maze



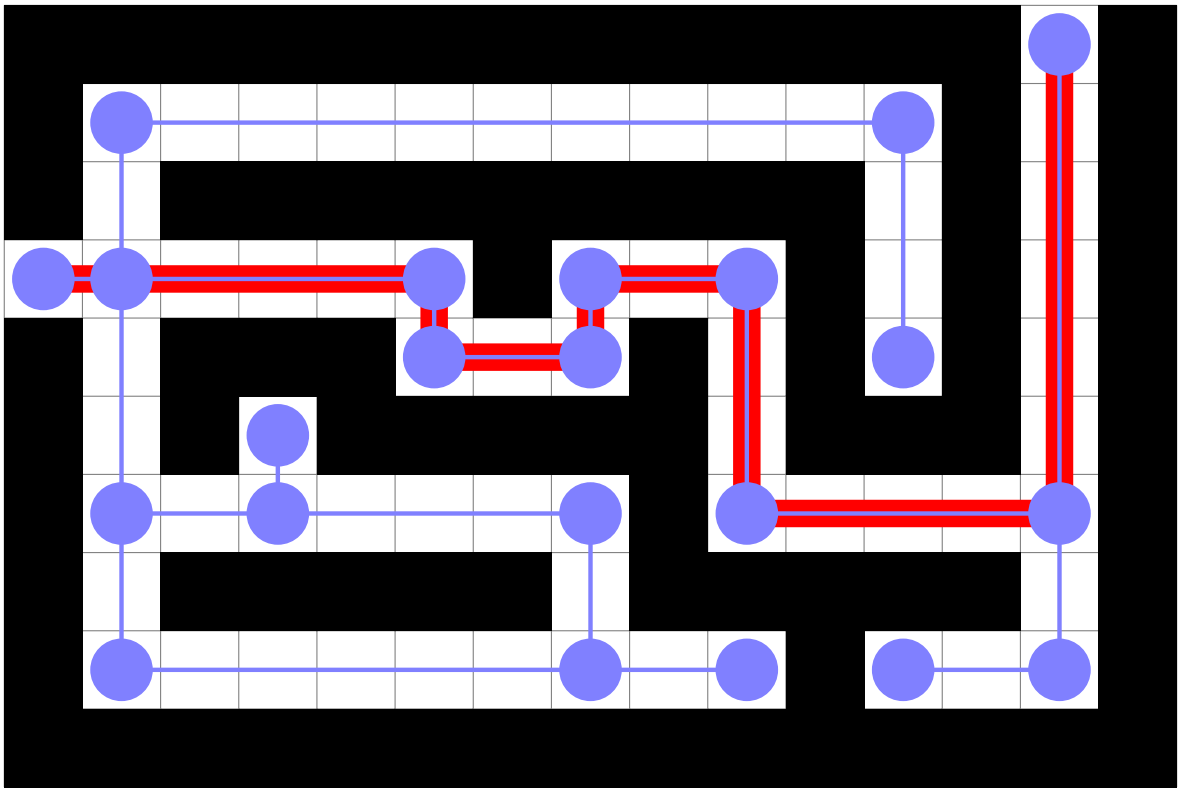
Insert vertices and edges

Maze



Remove unnecessary vertices

Maze



Find a path using BFS or DFS

Course plan

Best First Search algorithms

- In BFS and DFS algorithms every node is treated the same, the only difference is where we expand
- Those algorithms for sure will succeed in finding a path in our graph, but might take some time to produce a result
- However, if we have some previous knowledge about our system, we can take advantage of that!
- **Greedy algorithms** - at every decision point choose the best option according to some heuristics
- In Best First Search algorithms we follow the greedy approach - we assign every node priority according to some heuristics and then expand to the node with the highest priority
- The algorithm is exactly the same as for the BFS, but instead of using queue to store vertices to visit, we keep them in priority queue

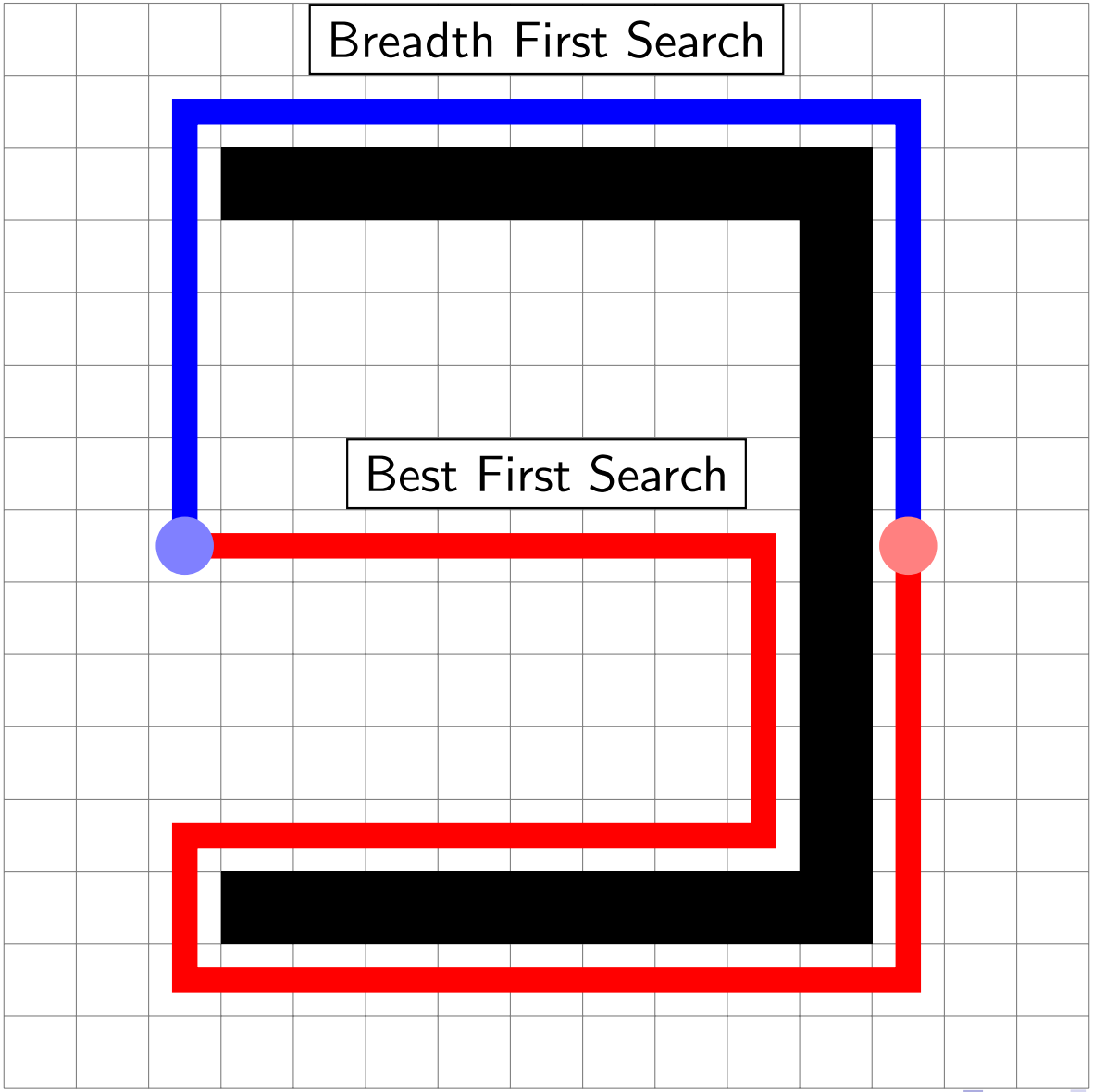
Best First Search pseudocode

```
1  BestFirstSearch(Graph, startVertex, endVertex)
2    for each vertex in Graph
3      vertex.isVisited = 0;
4    new PriorityQueue
5    priority = Heuristics(startVertex)
6    PriorityQueue.enqueue(startVertex, priority)
7    while(PriorityQueue is not empty)
8      vertex = PriorityQueue.dequeue()
9      if startVertex == endVertex
10       return
11     vertex.isVisited = 1
12     foreach neighbourVertex in vertex.neighbours
13       priority = Heuristics(neighbourVertex)
14       PriorityQueue.enqueue(neighbourVertex, priority)
```

Best First Search heuristics

- Some examples of possible heuristics:
  - Weight of a path leading to vertex
  - Euclidean distance from node to exit
  - Manhattan distance
  - Degree of a vertex
- Choosing a proper heuristic can improve the effectiveness of finding a path in graph dramatically
- However, in some graphs even simple heuristic can lead to potentially catastrophic results

Euclidean heuristics example



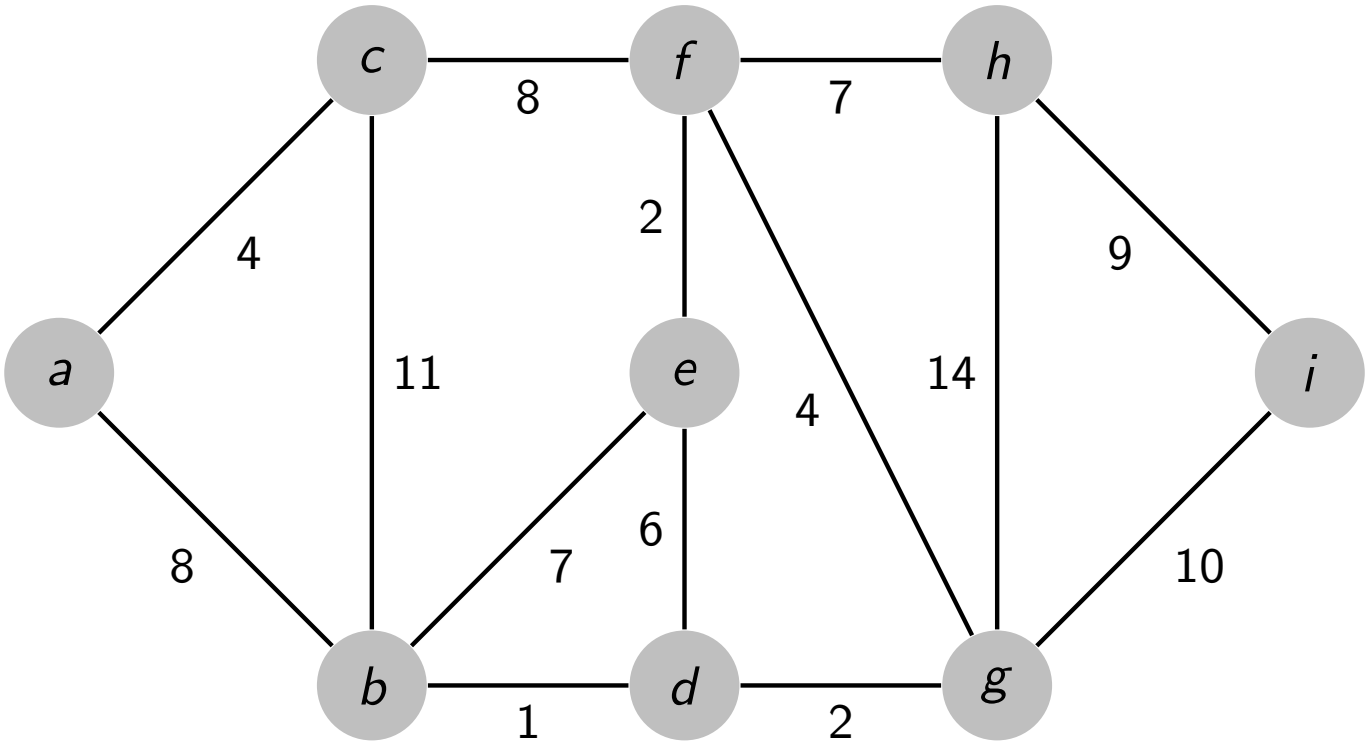
Course plan

1. Basic graph search
   - Depth-First Search
   - Breadth-First Search
   - Solving a maze
   - Best First Search algorithms
2. Weighted graph search
   - Dijkstra algorithm
   - Bellman–Ford algorithm
3. Pathfinding

Searching a weighted graph

- In weighted graph each edge has value attached to it - weight
- Weight tells us how costly it is to traverse this path
- All the algorithms presented before assumed, that weights of all the edges are the same therefore just comparing length of the path is enough to produce optimal result
- In weighted graph that might not be true - there might exist path containing more edges, but the sum of weights is lower than other, shorter paths

Weighted graph



Course plan

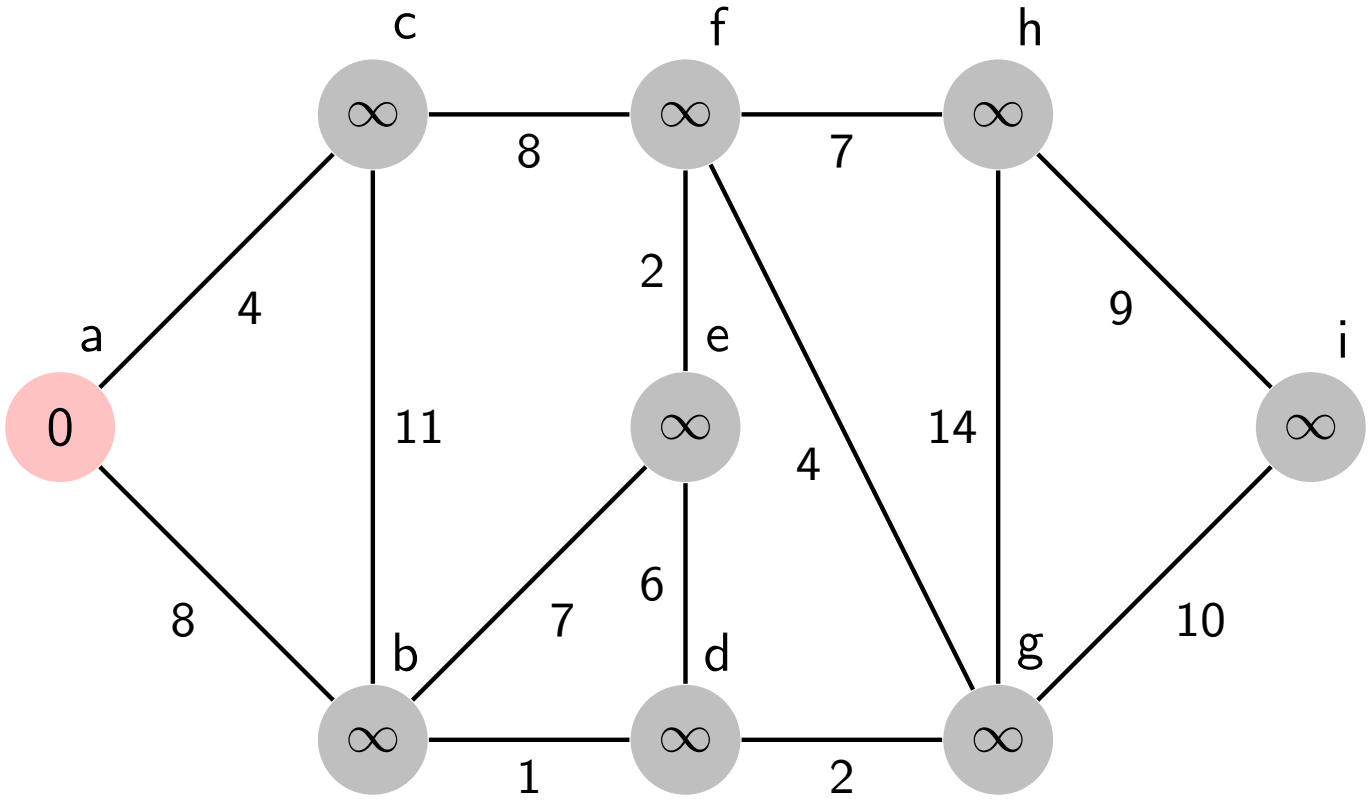
Dijkstra's algorithm

- Dijkstra's algorithm solves the problem of finding the shortest path from a single source vertex to every other vertex
- This algorithm work on every weighted graph (directed or undirected) with non-negative weights.
- The final output of the algorithm is a tree called Shortest Path Tree
- Dijkstra's algorithm uses relaxation to produce the final solution - initially costs of reaching vertices are overestimated and then at each step of the algorithm those values are improved

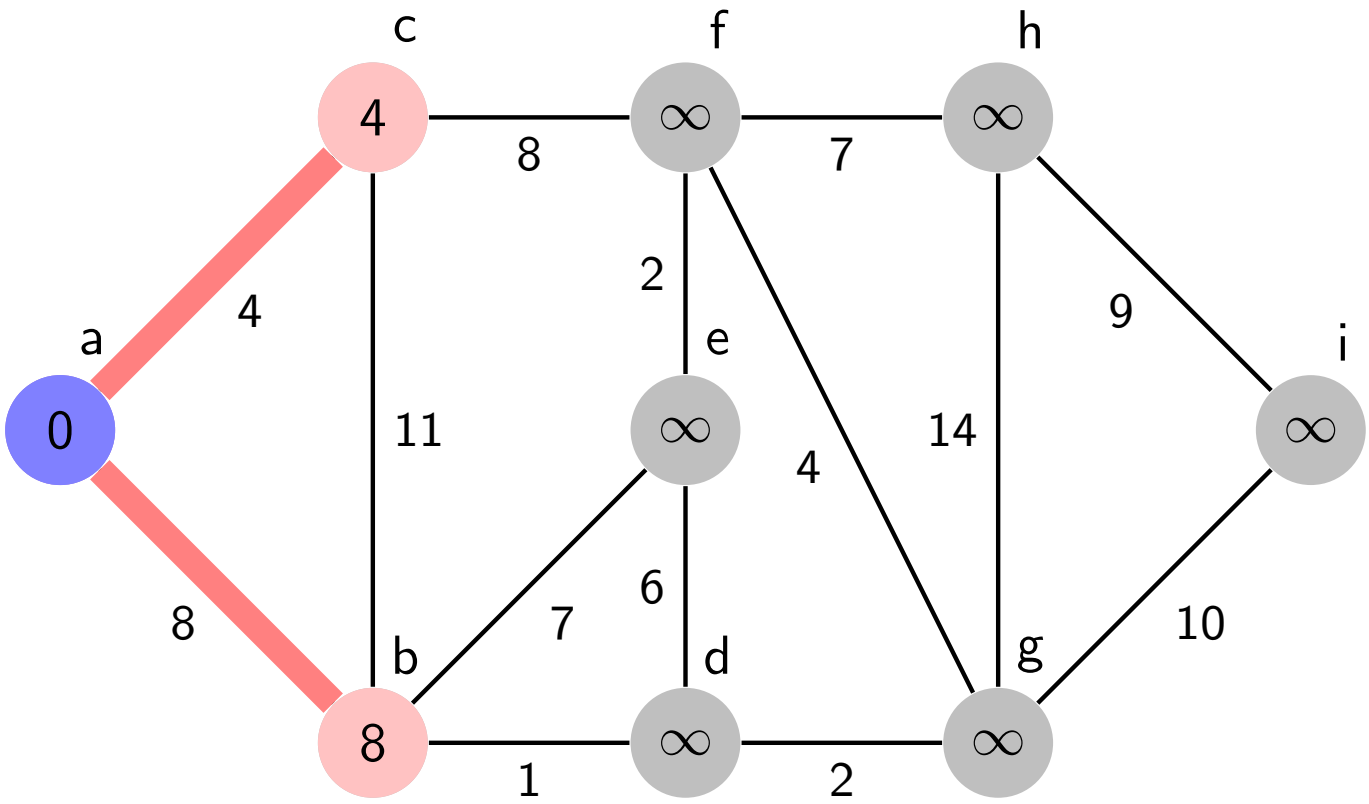
Dijkstra's algorithm

1. Mark all vertices unvisited
2. Assign tentative distance to each vertex: zero for initial and infinity for rest. Add starting vertex to priority queue with priority equal 0
3. Remove vertex from priority queue with minimum priority, set it as current and mark it as visited
4. For each neighbour of current vertex calculate tentative distance to that vertex by adding edge weight to current node tentative distance. For each of those neighbours update tentative distance if it's lesser than it's current
5. Add all neighbouring vertices that still not visited to priority queue with priority equaling their tentative distance
6. Go back to step 3

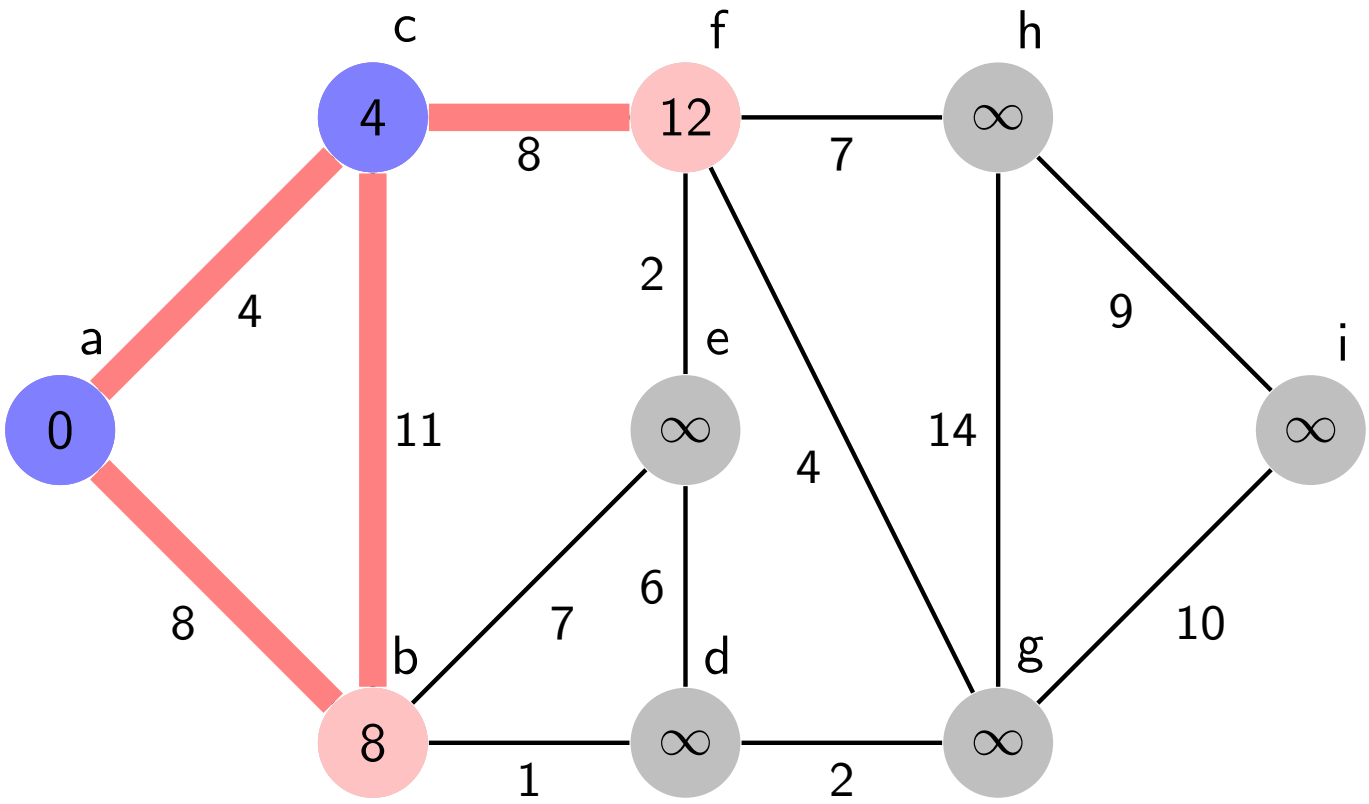
Dijkstra's algorithm example



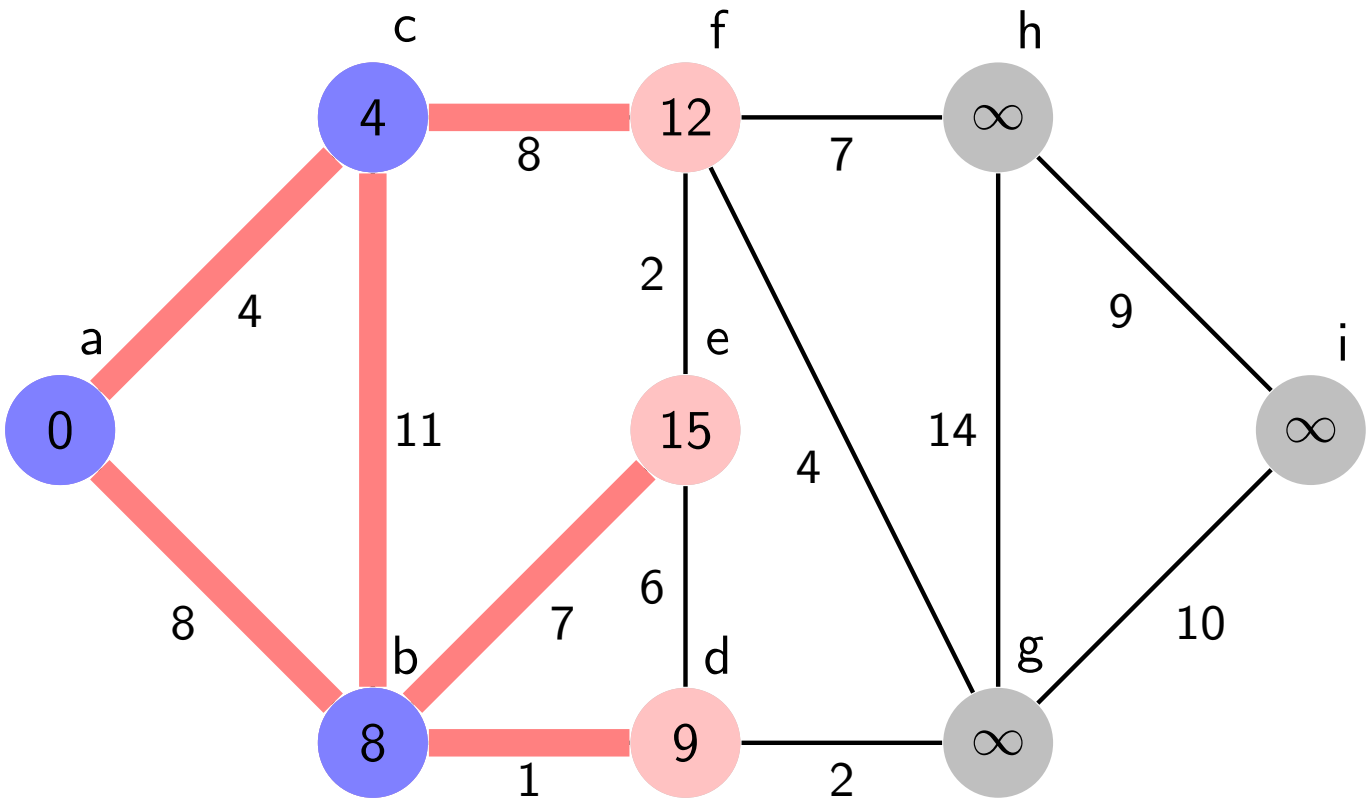
Dijkstra's algorithm example



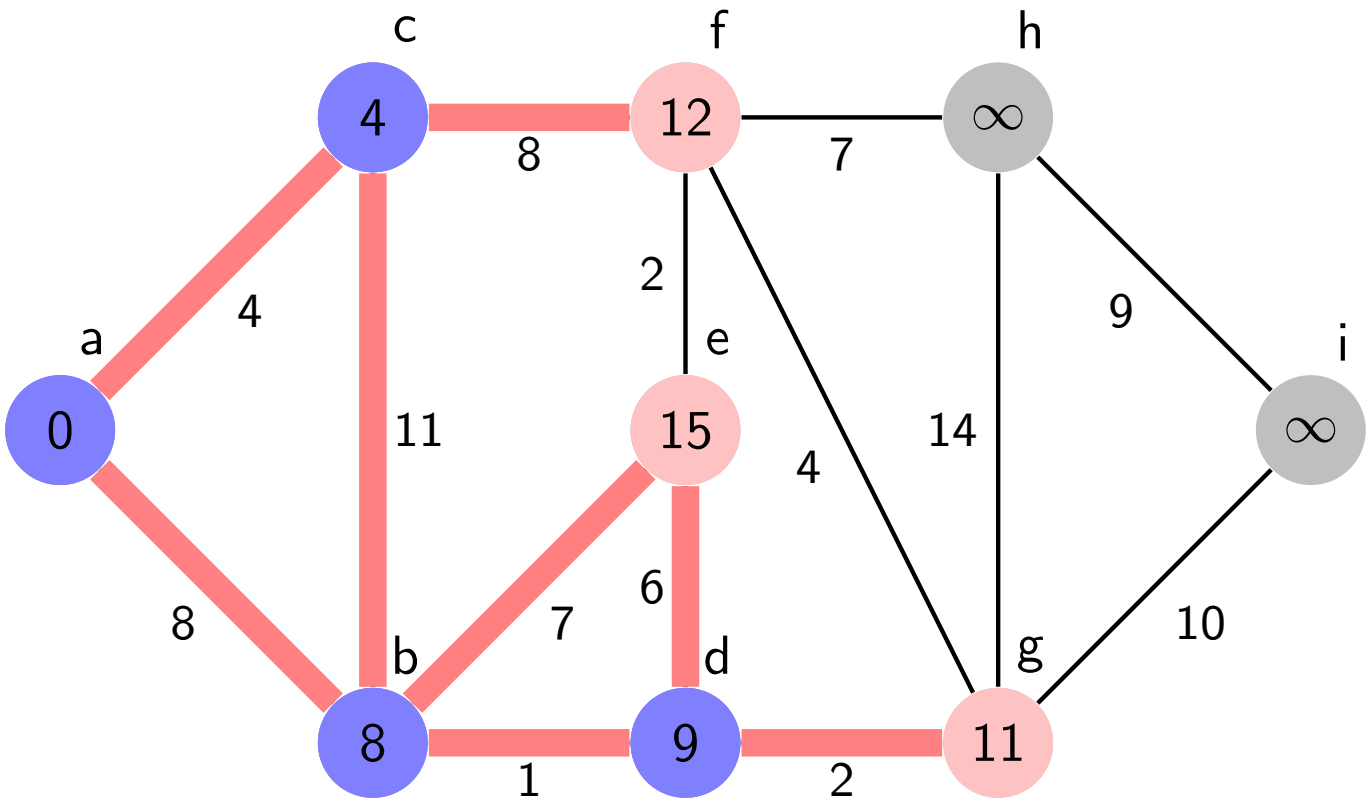
Dijkstra's algorithm example



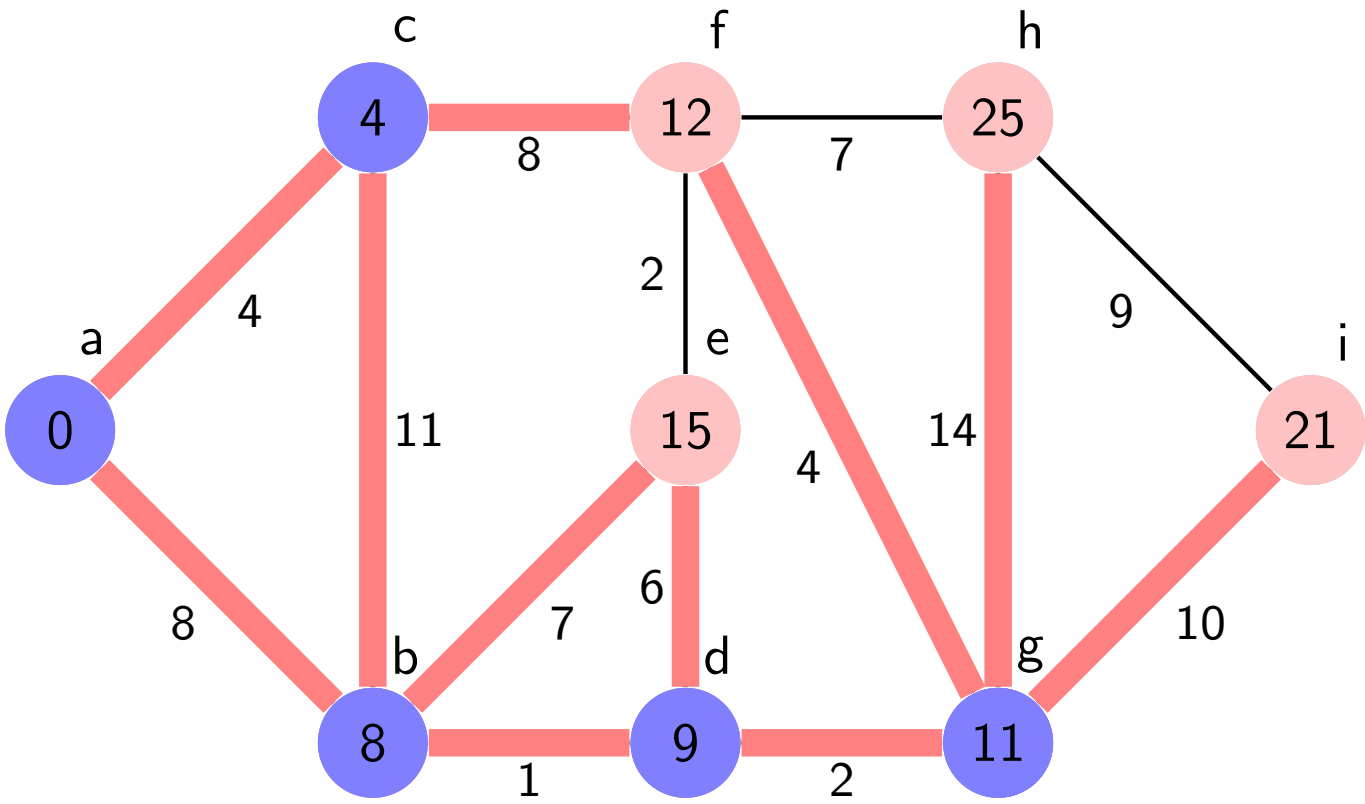
Dijkstra's algorithm example



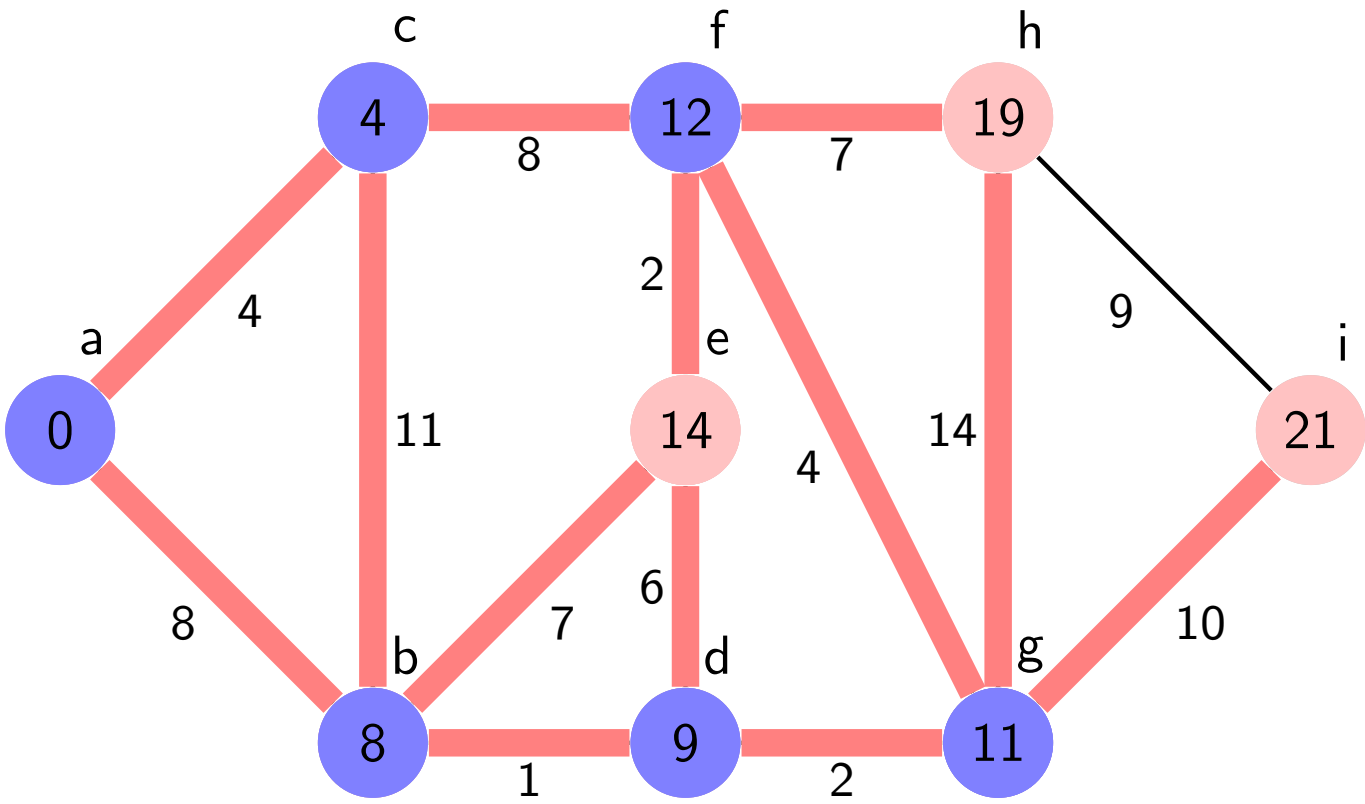
Dijkstra's algorithm example



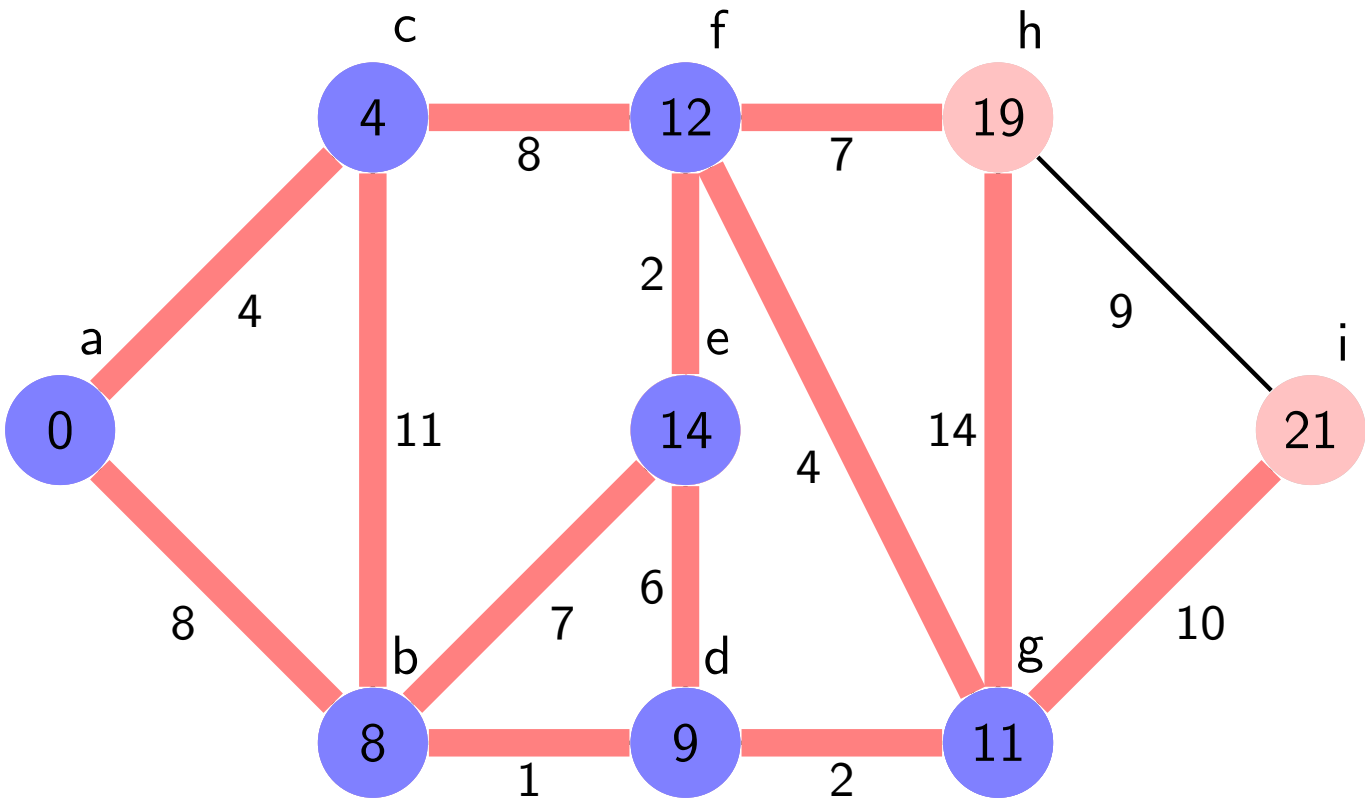
Dijkstra's algorithm example



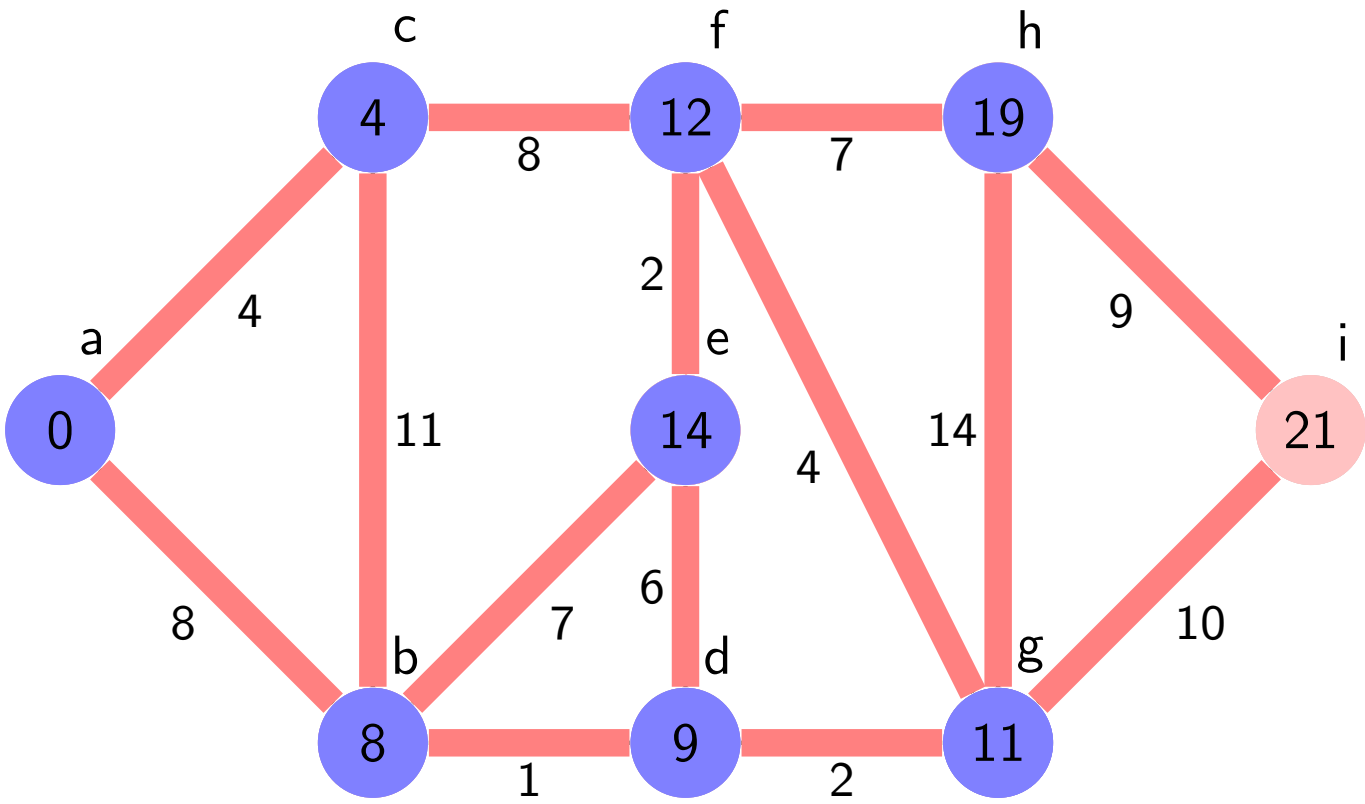
Dijkstra's algorithm example



Dijkstra's algorithm example



Dijkstra's algorithm example



Dijkstra's algorithm pseudocode

```
 1 Dijkstra(Graph, source)
 2   foreach vertex in Graph
 3     vertex.dist = infinity
 4   source.dist = 0
 5   source.previous = null
 6   new PriorityQueue
 7   PriorityQueue.Enqueue(source, source.dist)
 8   while (PriorityQueue.IsNotEmpty())
 9     currVertex = PriorityQueue.Dequeue
10     currVertex.visited = true
11     foreach nVertex in currVertex.neighbours()
12       tentativeDistance = currVertex.dist + Graph.distance(
       currVertex, nVertex)
13       if tentativeDistance < nVertex.dist
14         nVertex.dist = tentativeDistance
15         nVertex.previous = currVertex
16         PriorityQueue.setPriority(nVertex, nVertex.dist)
17       if nVertex.visited == false
18         PriorityQueue.enqueue(nVertex, nVertex.dist)
```

Dijkstra's algorithm analysis

- Complexity:
  - Building a priority queue: $O(V)$
  - Removing an element from priority queue: $O(lg\ V)$ and since we execute it once for every node complexity of this operation becomes $Vlg\ V$
  - Also for every edge `setPriority` operation is executued, resulting in $O(Elg\ V)$ complexity
  - Total complexity: $O((V + E)lg\ V)$
- Dijkstra's algorithm only work if no edge has negative weight
- This algorithm can work on directed graphs too
- This is greedy algorithm
- Edsger Dijkstra developed this algorithm during 20 minutes of coffee break, without using pen or pencil:)

Course plan

Bellman–Ford algorithm

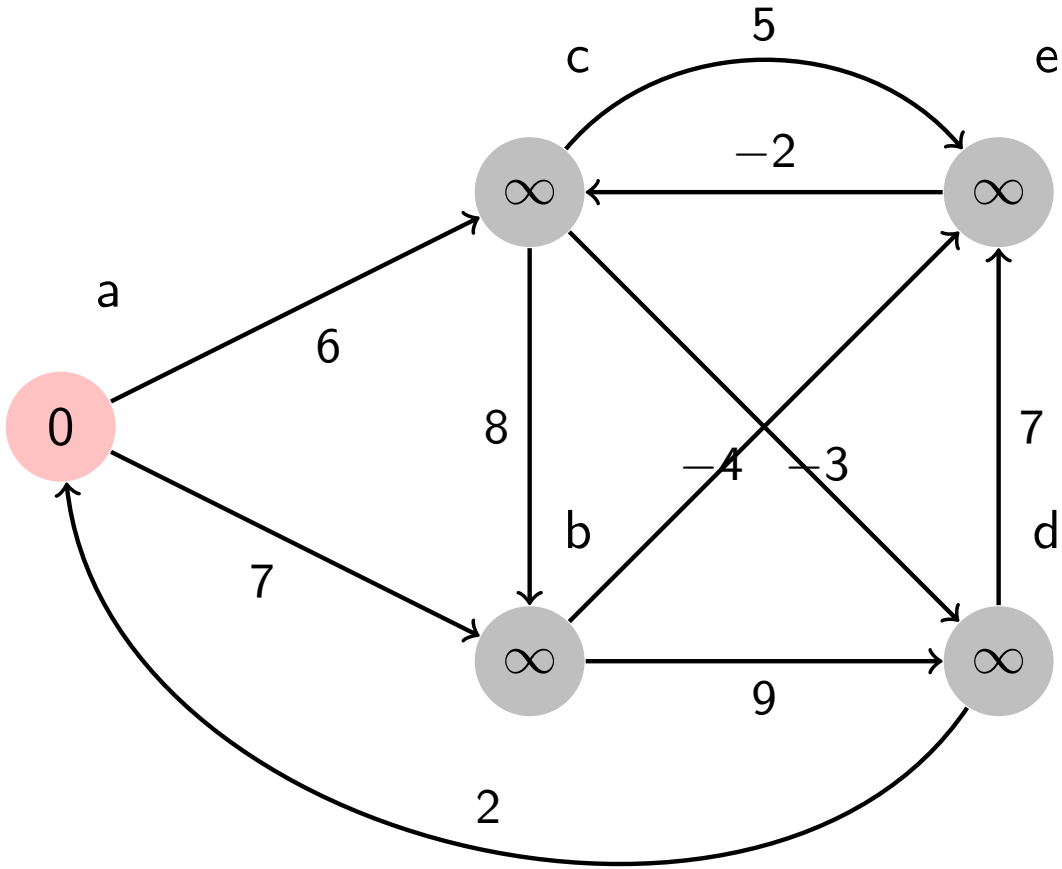
- One constraint of Dijkstra algorithm is that it cannot be used for graphs with negative weights
- Bellman-Ford algorithm can work for graphs with negative weights, but only if there is no negative cycle reachable from source
- If a negative cycle is found algorithm returns false - because of that it can be used to search for negative cycles in graphs
- This algorithm uses dynamic programming method - it first finds shortest paths for paths with at most one edge, then with at most two etc.

Bellman–Ford algorithm

1. Assign tentative distance to each vertex equal infinity except the source vertex, for which set zero
2. Do the following V-1 times:
   - For every edge (u,v) in graph, if $v.dist > u.dist + edge.weight$, then $v.dist = u.dist + edge.weight$
3. For every edge $u - v$ in graph
   - if $v.dist > u.dist + edge.weight$, then return FALSE (graph contains negative cycles)
4. return TRUE

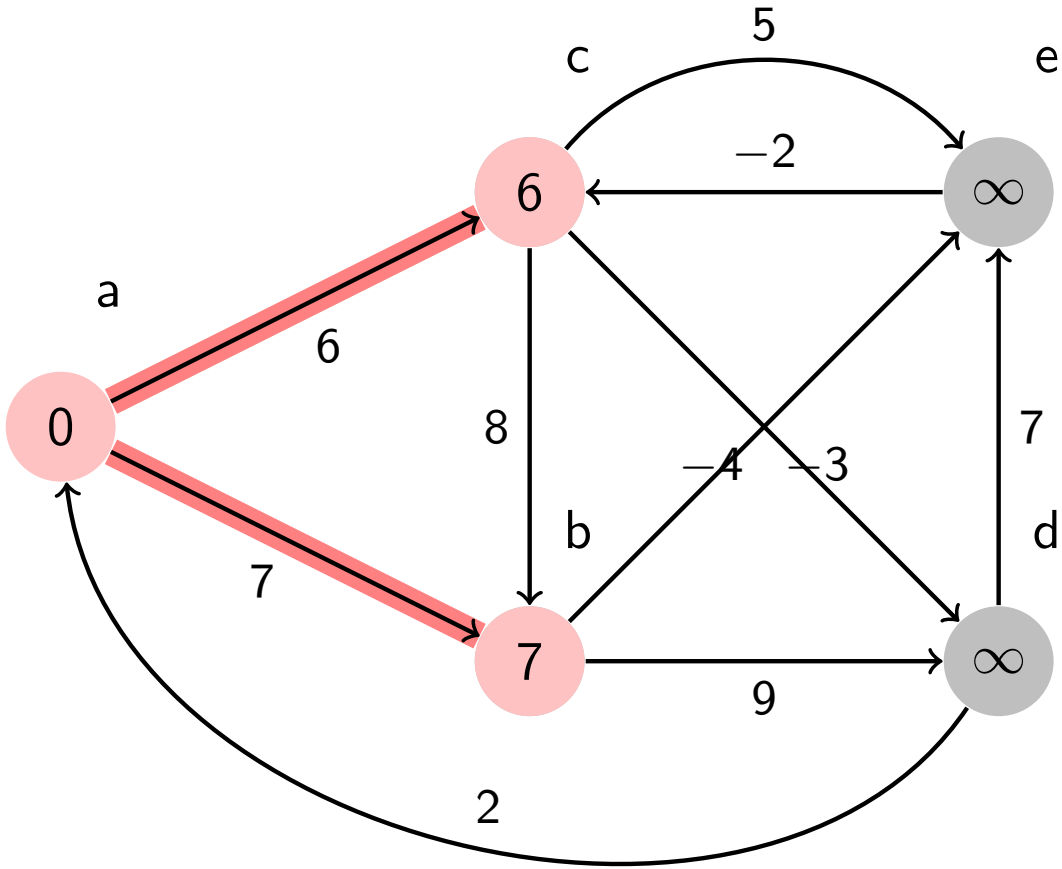
- First loop (line 2) looks for shortest path. We do it $V - 1$ times, because the longest path must be at most $V - 1$ long
- Second loop (line 3) checks, if doing one more iteration would reduce a distance to any node. If so it means there is a negative cycle.

Bellman-Ford algorithm example



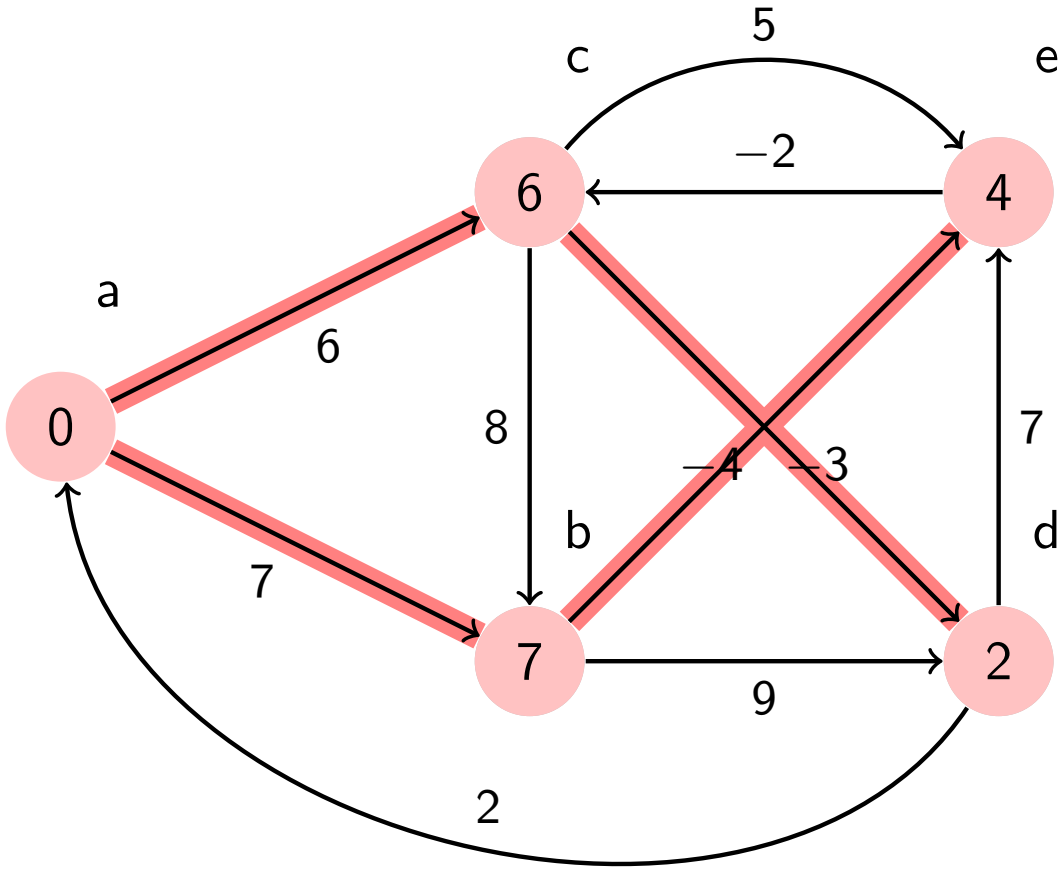
- Edge order: (c,e), (c,b), (c,d), (x,c), (b,e), (b,d), (d,e), (d,a), (a,c), (a,b)

Bellman-Ford algorithm example



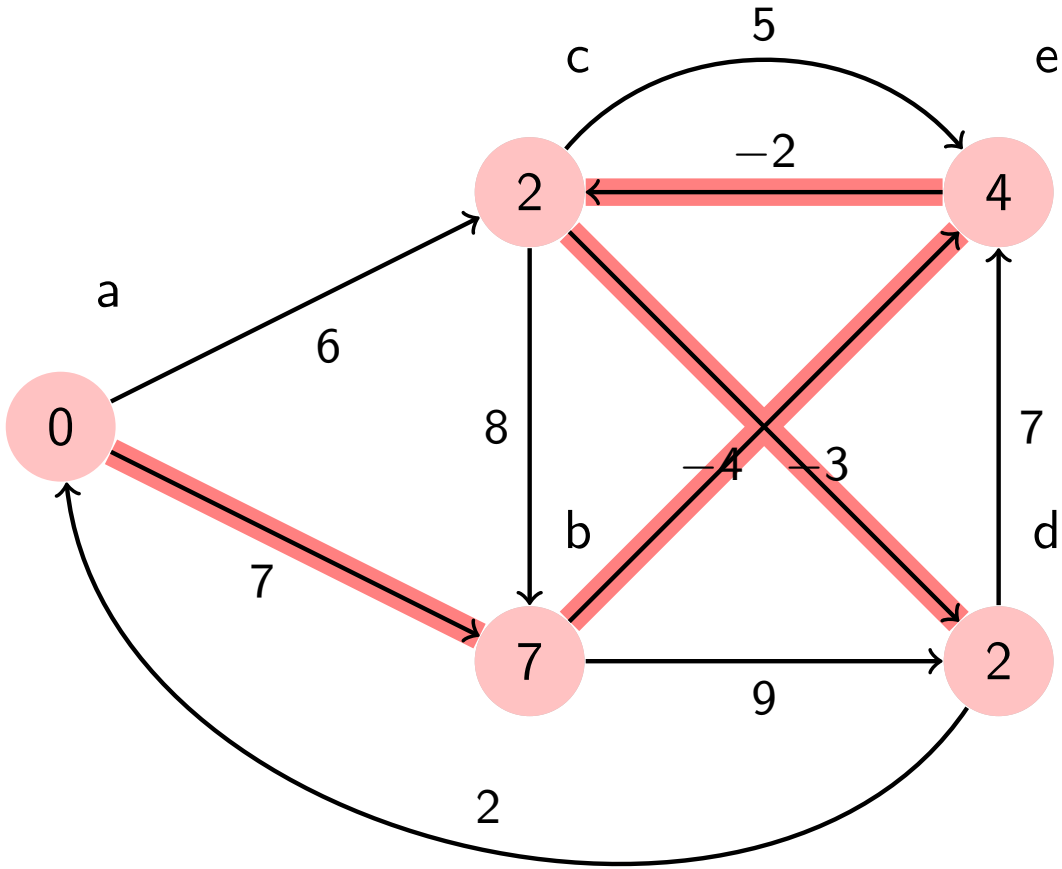
- Edge order: (c,e), (c,b), (c,d), (x,c), (b,e), (b,d), (d,e), (d,a), (a,c), (a,b)

Bellman-Ford algorithm example



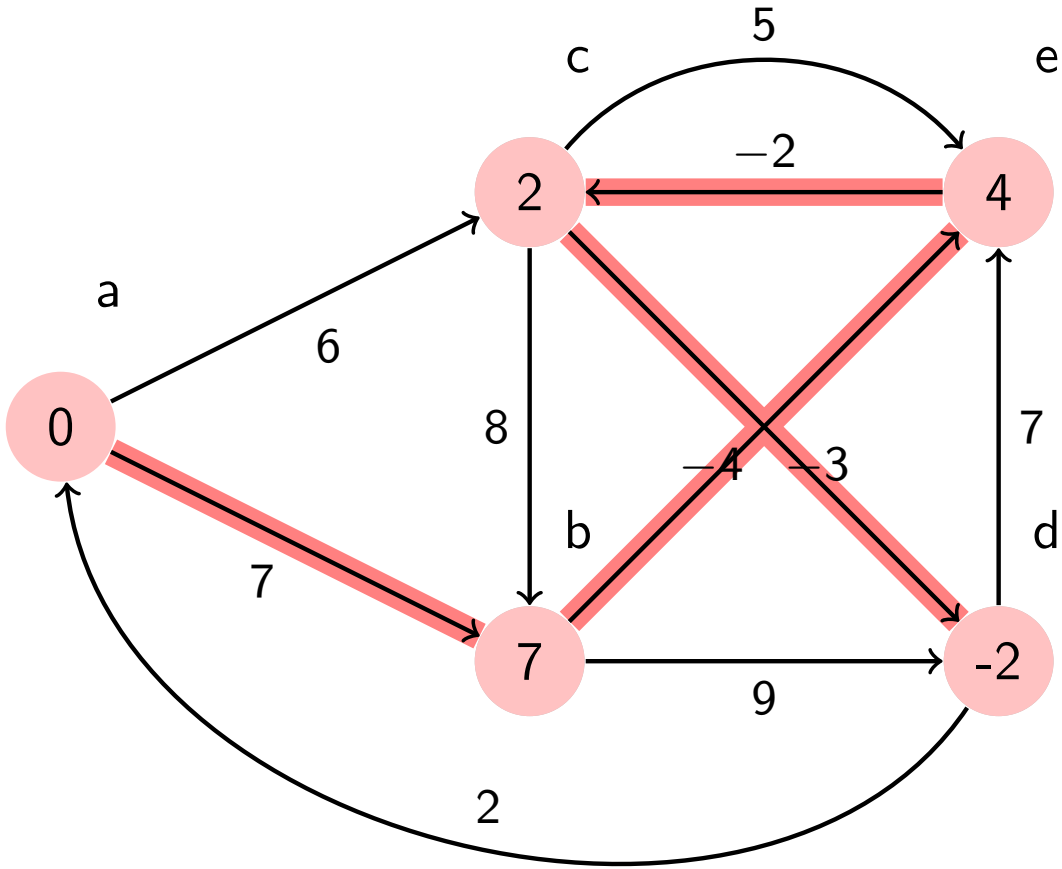
- Edge order: (c,e), (c,b), (c,d), (x,c), (b,e), (b,d), (d,e), (d,a), (a,c), (a,b)

Bellman-Ford algorithm example



- Edge order: (c,e), (c,b), (c,d), (x,c), (b,e), (b,d), (d,e), (d,a), (a,c), (a,b)

Bellman-Ford algorithm example



- Edge order: (c,e), (c,b), (c,d), (x,c), (b,e), (b,d), (d,e), (d,a), (a,c), (a,b)

Bellman-Ford algorithm analysis

- Complexity - for each vertex in V we need to check every edge in E, so complexity is $O(VE)$
- It's worse than Dijkstra's $O((V + E) lg\ V)$
- Bellman-Ford algorithm can be applied to graphs with negative weights (but without negative cycles), Dijkstra's algorithm cannot
- It can also be used to search for negative cycle in graph
- Bellman-Ford algorithm applies dynamic programming method, Dijkstra's algorithm is greedy algorithm
- Graph problems with negative weights:
  - While modeling cashflow some operations produce negative income
  - Chemical reactions can either produce or absorb energy
  - Planning transport routes for ships - if ship carries cargo then edge has positive weight, but if it's traveling without one then edge weight is negative

Course plan

1. Basic graph search
   - Depth-First Search
   - Breadth-First Search
   - Solving a maze
   - Best First Search algorithms
2. Weighted graph search
   - Dijkstra algorithm
   - Bellman–Ford algorithm
3. Pathfinding

Pathfinding problem

- If we need to find a path on graph we have at our disposal:
  - Breadth-first search algorithm, that will for sure find an optimal path, but will take very long time to finish
  - Best-first search, that produces results very quickly, but might produce suboptimal path
  - Dijkstra's algorithm, that will find optimal path, but wastes a lot of time exploring unimportant directions
- Let's create a function $f(n)$, that when applied to a node would return how likely are we to visit that node in our path search
- In BFS algorithm $f(n) = 1$, that is we are equally likely to visit every node in our graph
- In Best First Search algorithm $f(n) = h(n)$, that is we apply some heuristic function h to calculate if we should visit specific node. Function $h(n)$ depends on distance between node n and exit.
- In Dijkstra's algorithm $f(n) = g(n)$, where $g(n)$ is heuristic function depending on distance from source

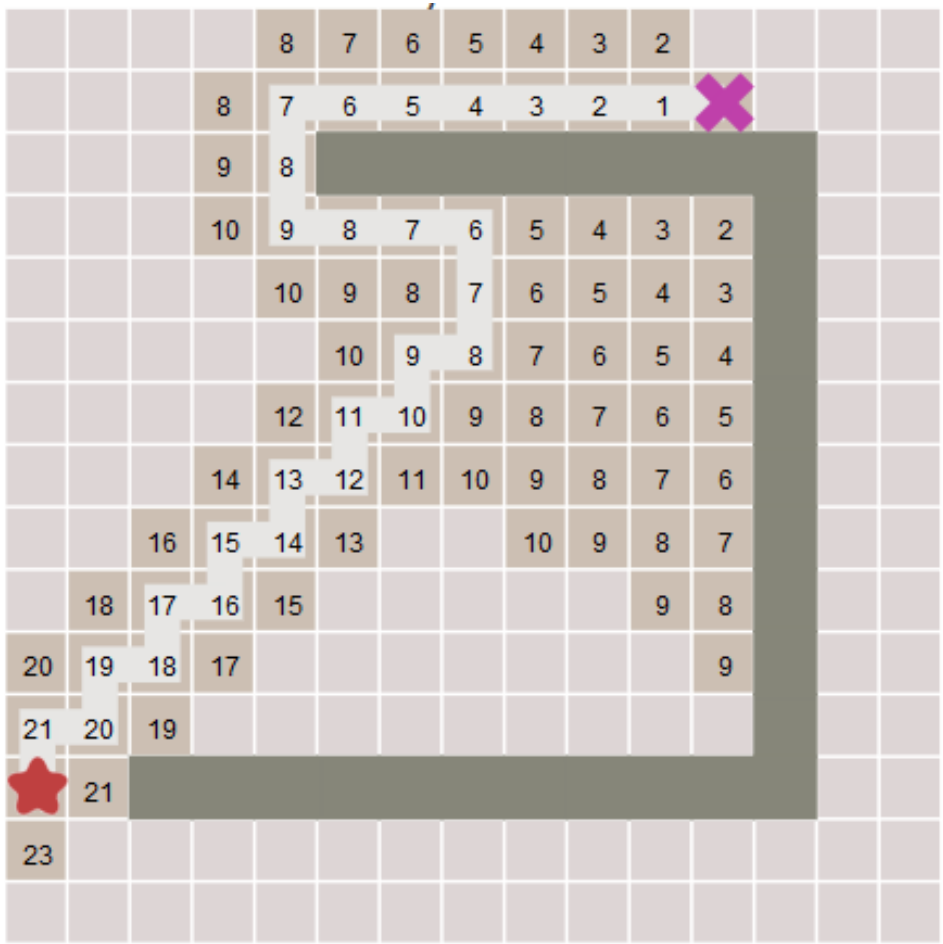
A^* algorithm

- A^* algorithm is an algorithm, for which $f(n) = g(n) + h(n)$, where $h(n)$ is a heuristic estimated distance from node n to the goal, and $g(n)$ is actual cost of getting to node n from the start
- Main idea of A^* algorithm is to avoid exploring paths that are already expensive
- It can be seen as an extensions to Dijkstra algorithm
- It was discovered in 1968, almost ten years after Dijkstra's algorithm invention
- Calculating $g(n)$ is easy, but to calculate $h(n)$ we need to use some function, like manhattan distance, euclidean distance etc.

A^* algorithm explained

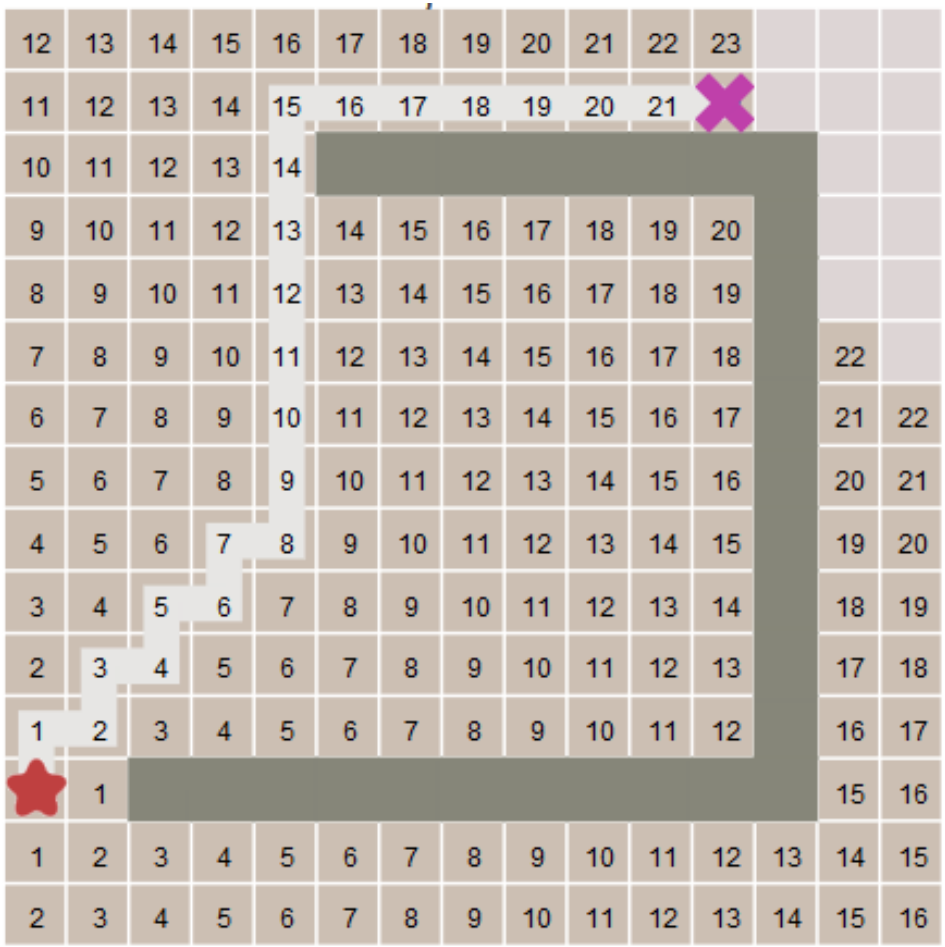
1. Create open and closed lists of nodes
2. Add the starting node to the open list
3. Repeat the following:
   1. Set current node as node with lowest f in open list.
   2. Move current node to closed list
   3. For each of current nodes neighbours:
      1. Skip it if it's in closed list
      2. If it isn't in the open list, add it to the open list. Make the current node the parent of this neighbour. Record the f, g, h values of the neighbour
      3. If it is on the open list already, check to see if this path to that node is better, using g cost as the measure. A lower g cost means that this is a better path. If so, change the parent of the neighbour to the current node, and recalculate the g and f scores of the neighbour.
      4. Stop when destination is added to closed list (path has been found), or open list is empty (there is no path)
4. Reconstruct the path, by picking destination node and recursively going through parents

Example - Best-first search



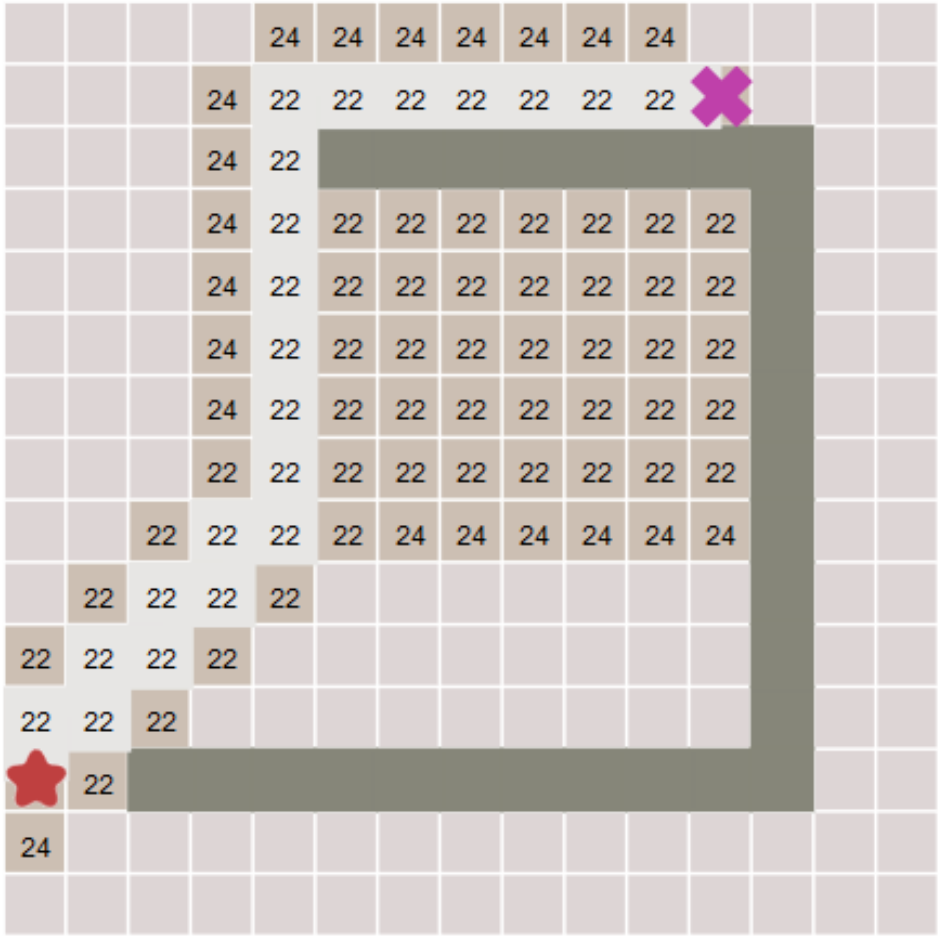
Numbers indicate distance from exit

Example - Dijkstra algorithm



Numbers indicate distance from source

Example - $A*$ algorithm



Numbers indicate distance from start + distance from exit

A^* algorithm analysis

- A^* algorithm is complete, it is guaranteed to find a path (if it exists)
- It is also optimal - it can be proven, that no other algorithm with the same heuristics will visit fewer nodes.
- If improper heuristics are chosen however, algorithm might return suboptimal path
- A^* and A^*-based algorithms are widely used in pathfinding problems, common for example in video games
- If we set $h = 0$ for all nodes, then this algorithm becomes Dijkstra algorithm
- Efficient implementations of A^* use priority queue and hash tables as open/closed lists
- Depending on how ties are handled in priority queue of nodes to visit slightly different behaviour can be obtained - either algorithm will go deep, or wide.