

State Space Search Algorithms

Vedran Pintarić

15.11.2018.

Problem definition

- a set of states (state space)
- initial state
- transitions between states
- goal state test

Problem definition

```
class Problem:  
    # Should return a State object  
    def getInitialState(self):  
        pass  
  
    # Should check if given  
    # state is goal state  
    def isGoalState(self, state):  
        pass  
  
    # Should return a list  
    # of triplets (state, action, cost)  
    def getSuccessors(self, state):  
        pass
```

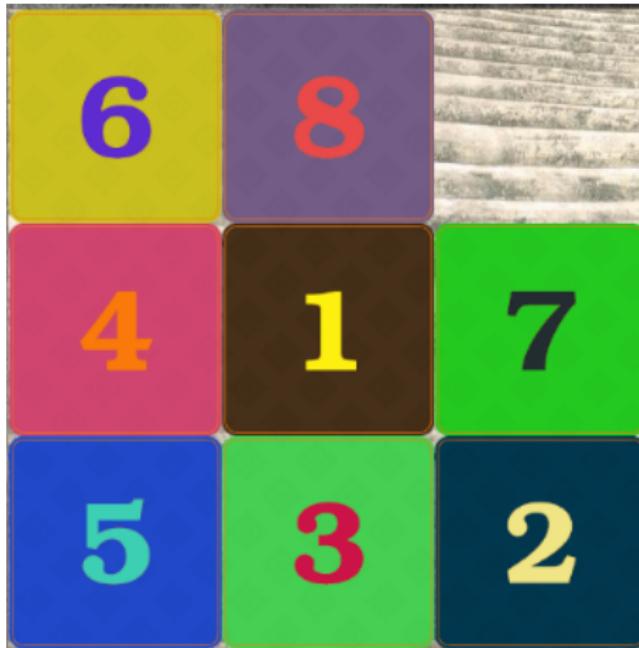


Figure 1: 8-Puzzle initial state

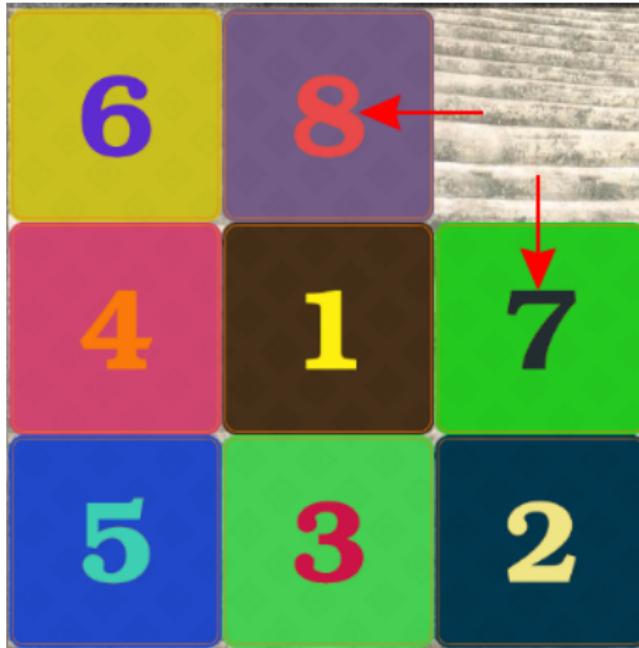
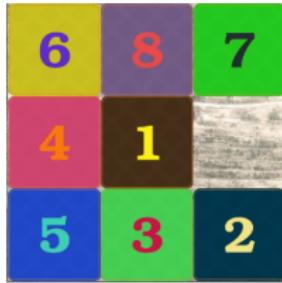
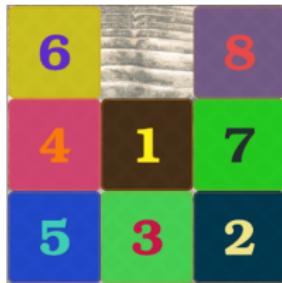


Figure 2: 8-Puzzle possible actions



MoveDown 1



MoveLeft 1

Figure 3: 8-Puzzle list of successor triplets

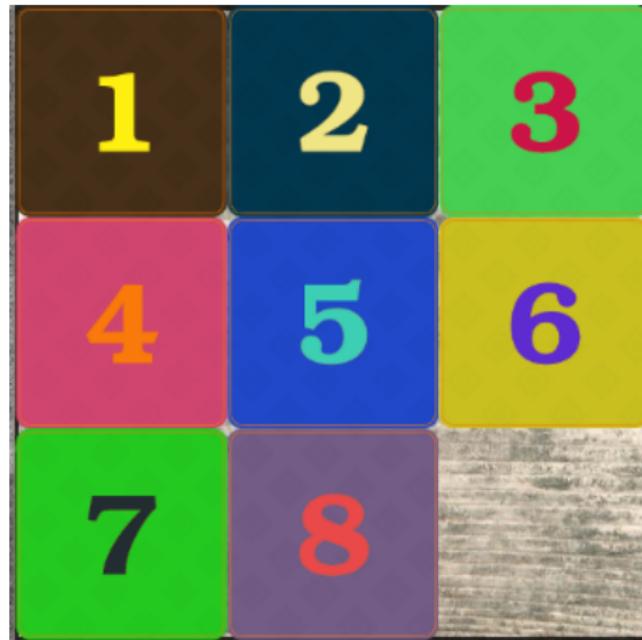


Figure 4: 8-Puzzle goal state

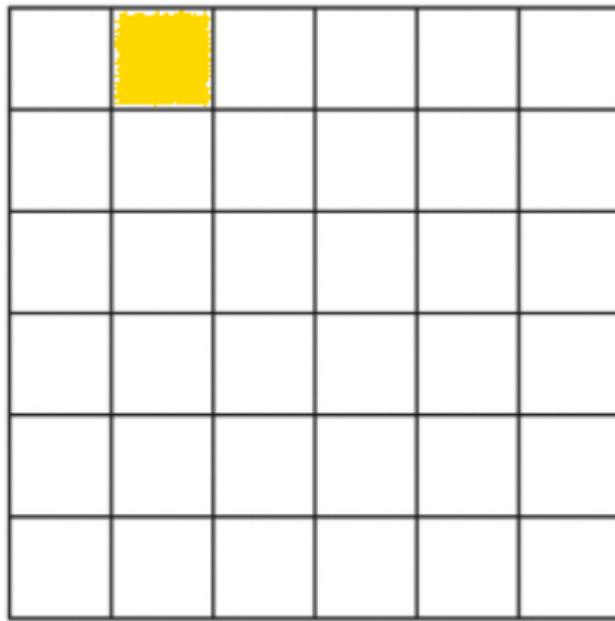


Figure 5: Pathfinding initial state

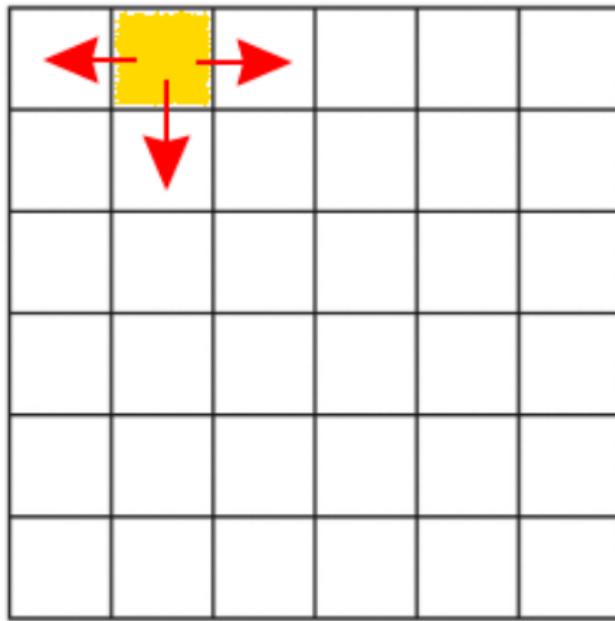
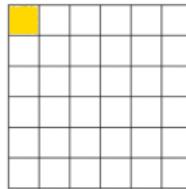
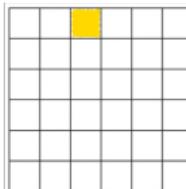


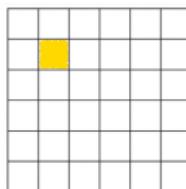
Figure 6: Pathfinding possible actions



MoveLeft 1



MoveRight 1



MoveDown 1

Figure 7: Pathfinding list of successor triplets

State space graph

State space graph

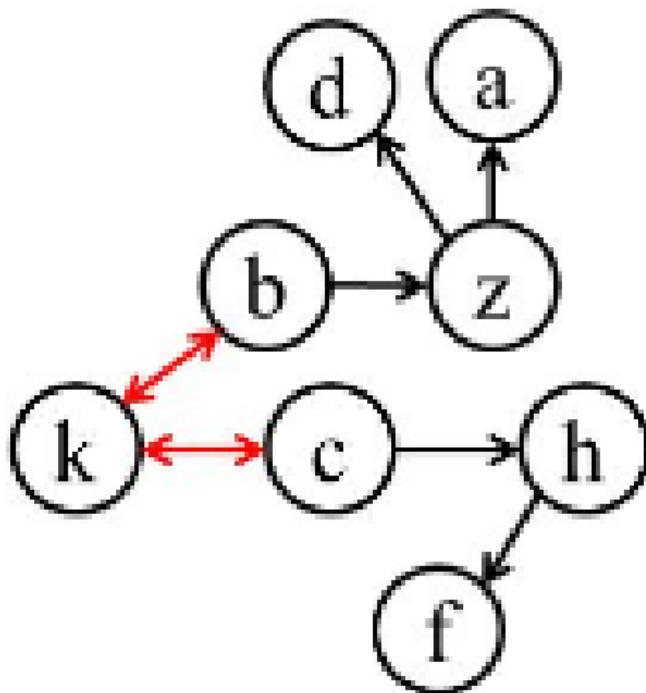


Figure 8: State space graph

Search tree

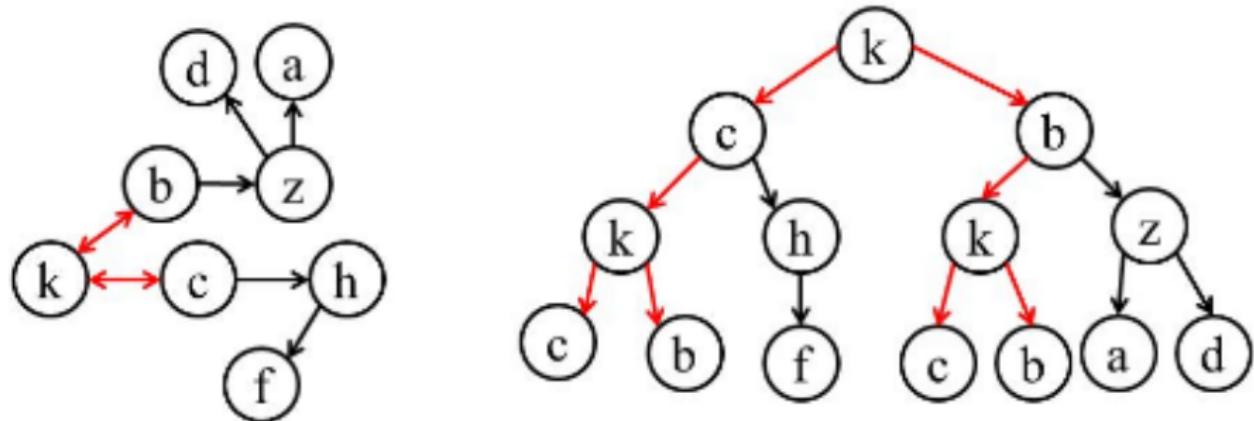


Figure 9: State space vs search tree

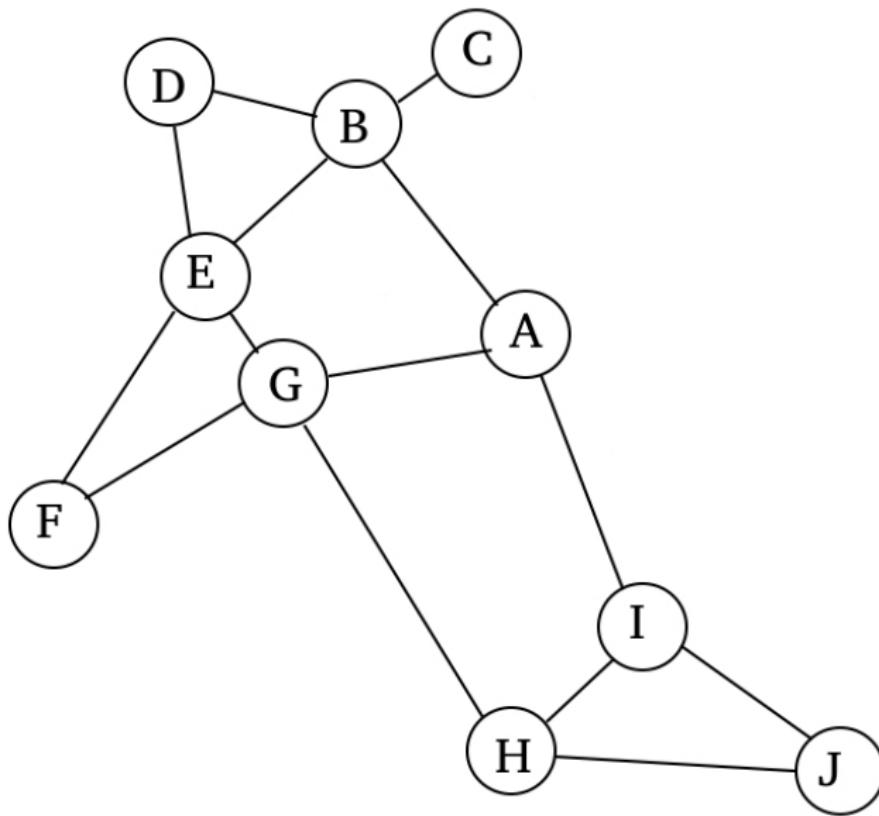
Nodes

- Open nodes - known but unexplored
- Closed nodes - explored i.e. checked against goal state and their successors were queried

General algorithm pseudocode

```
closedNodes = {}  
openNodes = {}  
  
openNodes.insert(getInitialState())  
while openNodes not empty  
    node = openNodes.get()  
    if isGoalState(node)  
        return node  
  
    if node in closedNodes  
        continue  
  
    for childNode in getSuccessors(node)  
        if childNode not in closedNodes  
            openNodes.insert(childNode)  
    closedNodes.insert(node)
```

Working example



Blind algorithms



- no additional information about the problem
- needs to figure everything out on its own

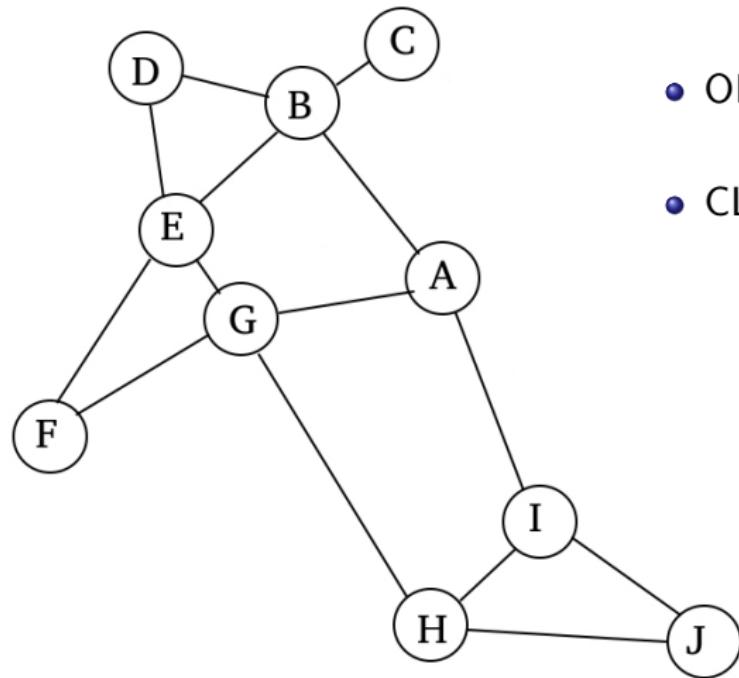
Breadth First Search (BFS)

```
closedNodes = {}

# First In First Out
openNode = Queue()

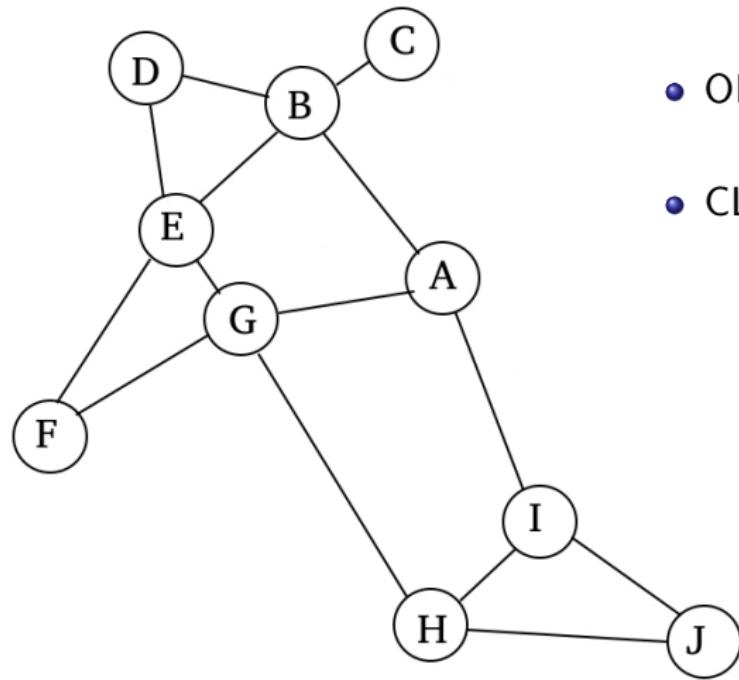
openNodes.insert(getInitialState())
while openNodes not empty
    node = openNodes.get()
    if isGoalState(node)
        return node
    if node in closedNodes
        continue
    for childNode in getSuccessors(node)
        if childNode not in closedNodes
            openNodes.insert(childNode)
    closedNodes.insert(node)
```

Breadth First Search (BFS)



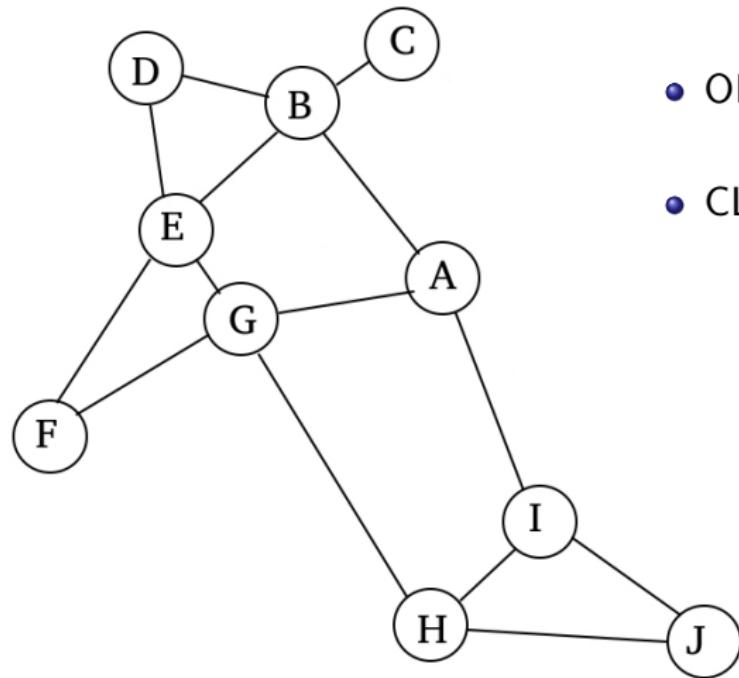
- Find a path from B to I
- OPEN: [B]
- CLOSED: {}

Breadth First Search (BFS)



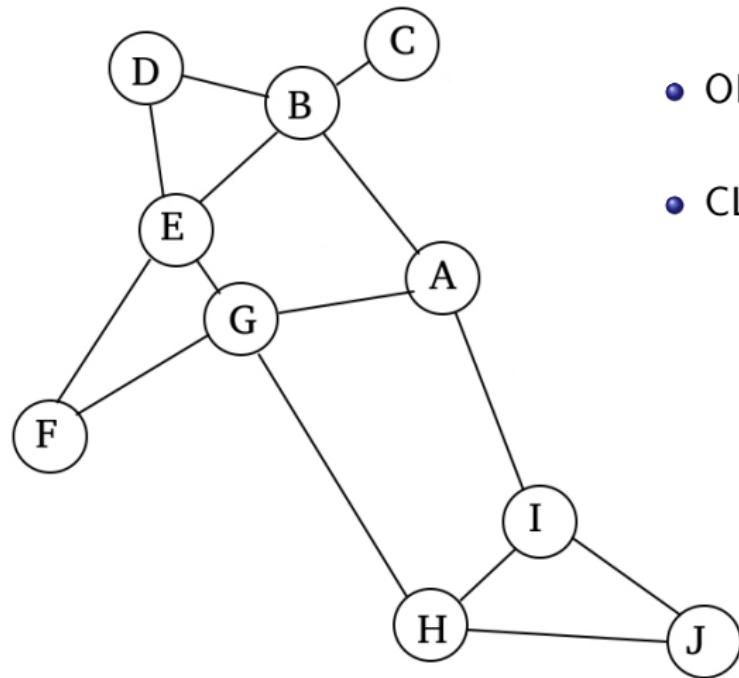
- Find a path from B to I
- OPEN: [C, D, E, A]
- CLOSED: { B }

Breadth First Search (BFS)



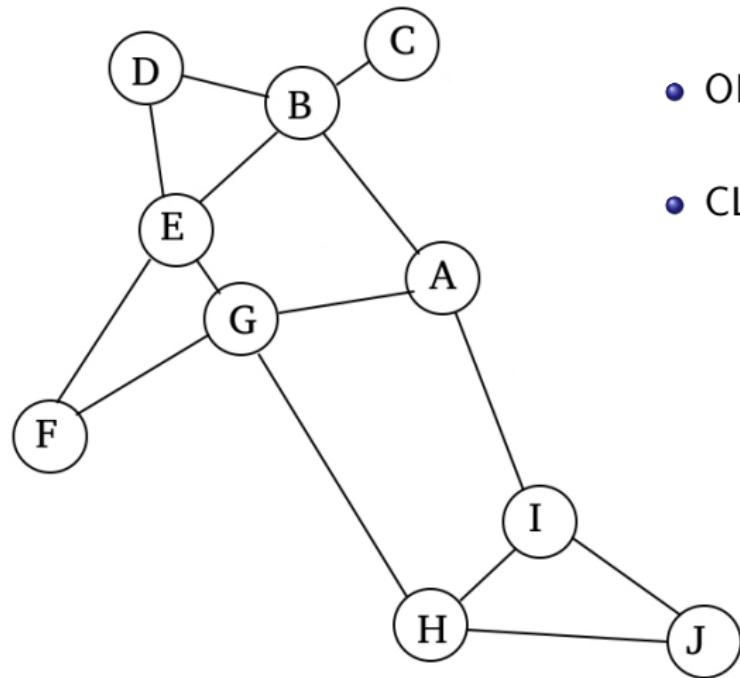
- Find a path from B to I
- OPEN: [D, E, A]
- CLOSED: { B, C }

Breadth First Search (BFS)



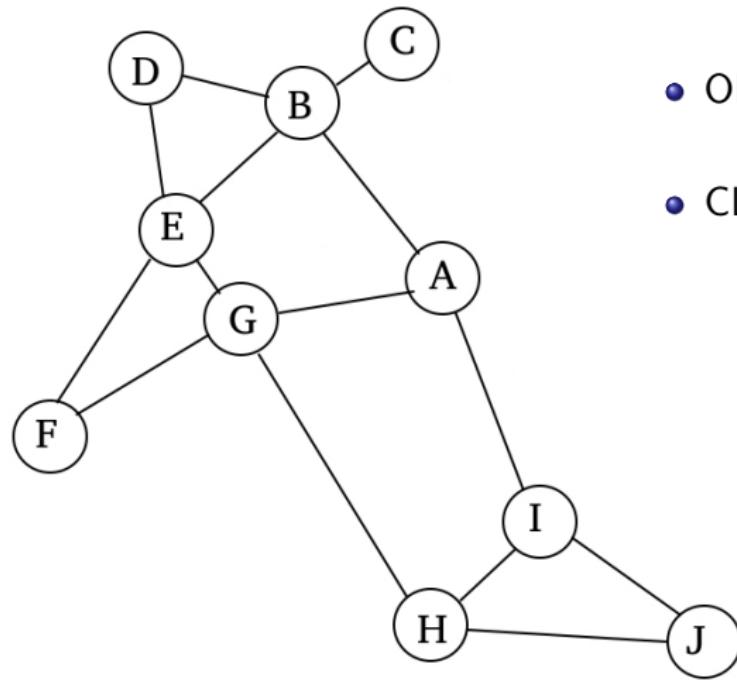
- Find a path from B to I
- OPEN: [E, A]
- CLOSED: { B, C, D }

Breadth First Search (BFS)



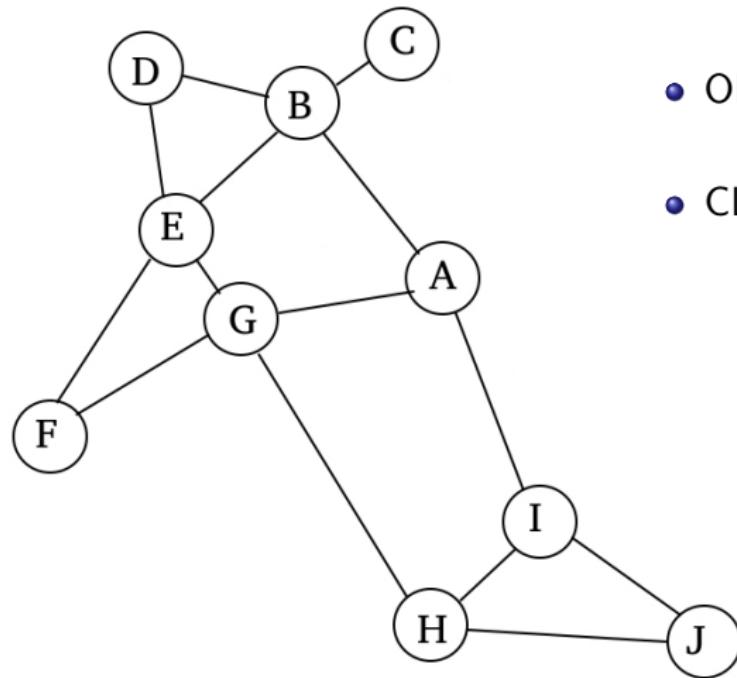
- Find a path from B to I
- OPEN: [A, F, G]
- CLOSED: { B, C, D, E }

Breadth First Search (BFS)



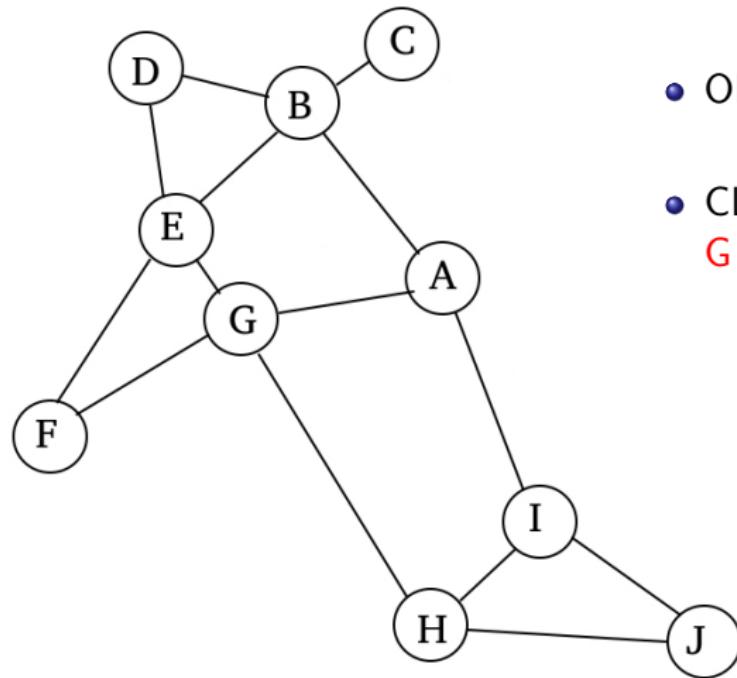
- Find a path from B to I
- OPEN: [F, G, I]
- CLOSED: { B, C, D, E, A }

Breadth First Search (BFS)



- Find a path from B to I
- OPEN: [G, I]
- CLOSED: { B, C, D, E, A, **F** }

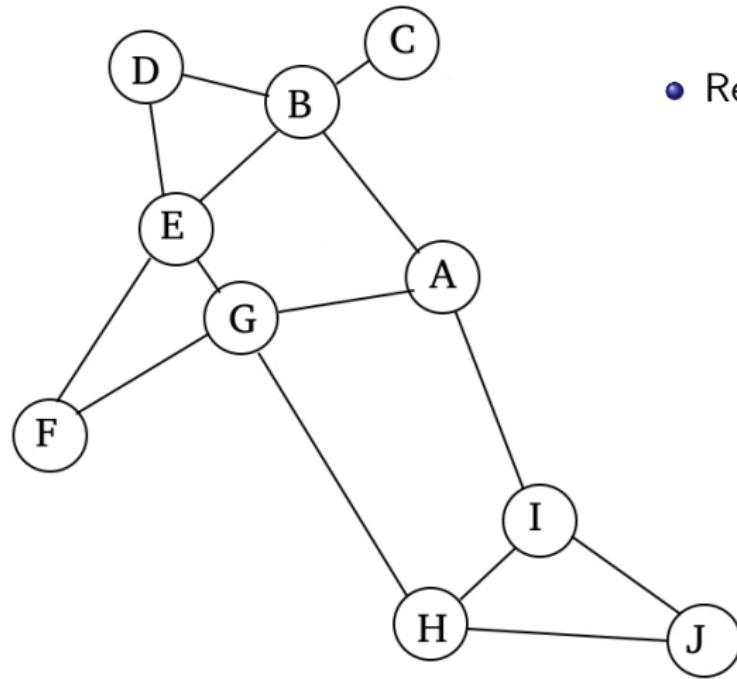
Breadth First Search (BFS)



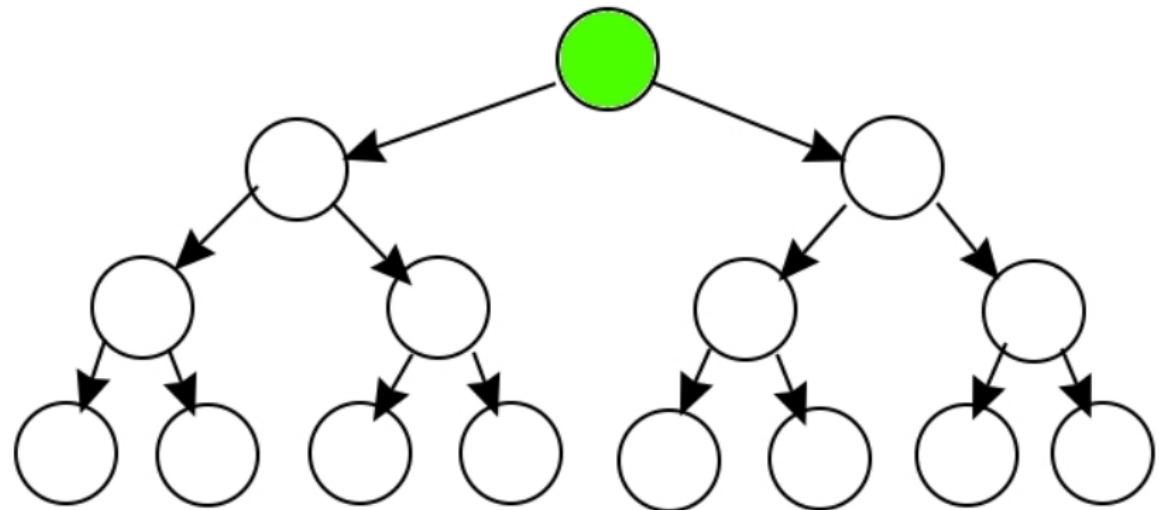
- Find a path from B to I
- OPEN: [I, **H**]
- CLOSED: { B, C, D, E, A, F, **G** }

Breadth First Search (BFS)

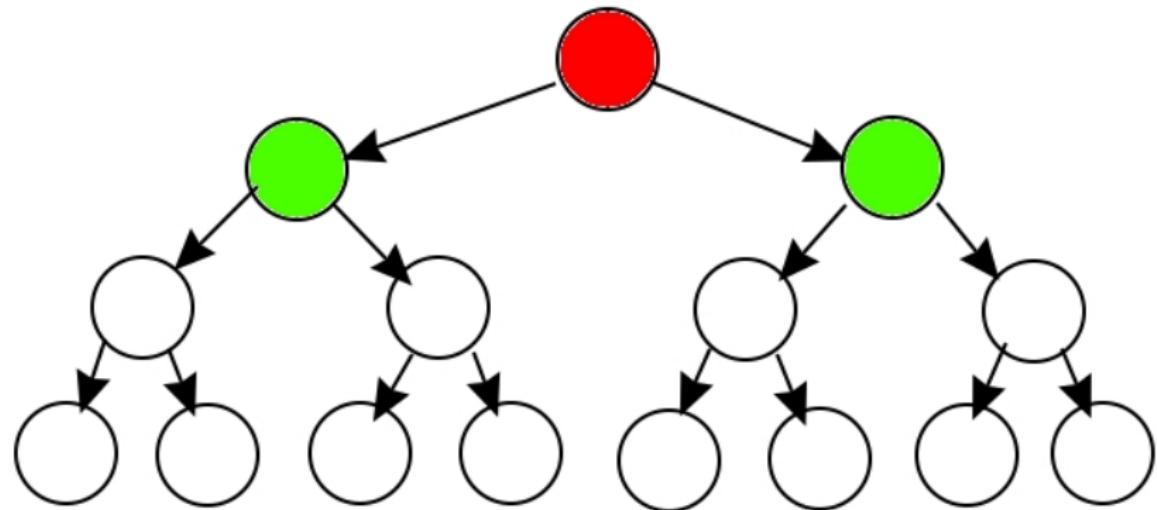
- Find a path from B to I
- Reached goal I



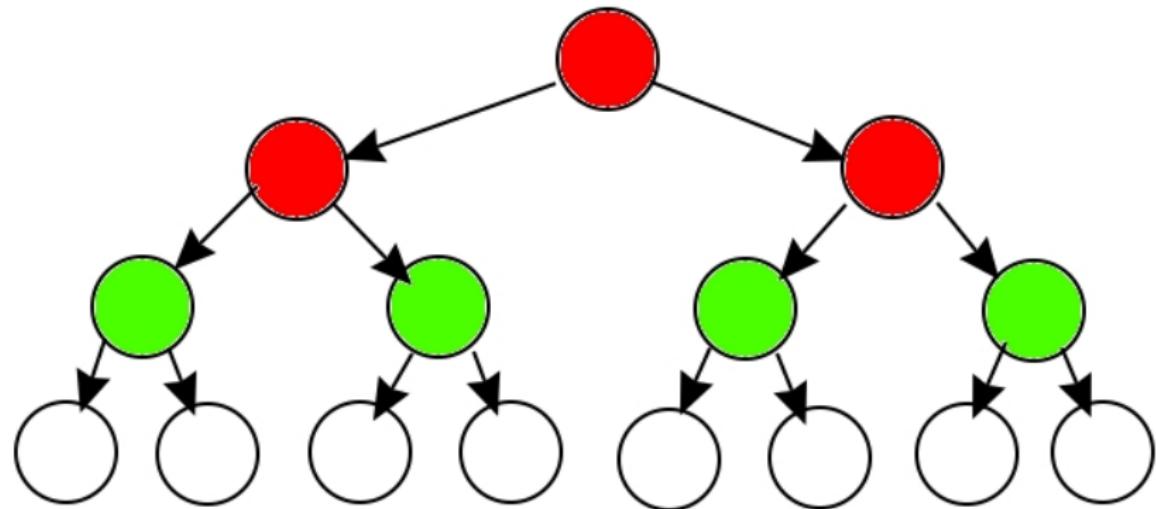
Breadth First Search (BFS)



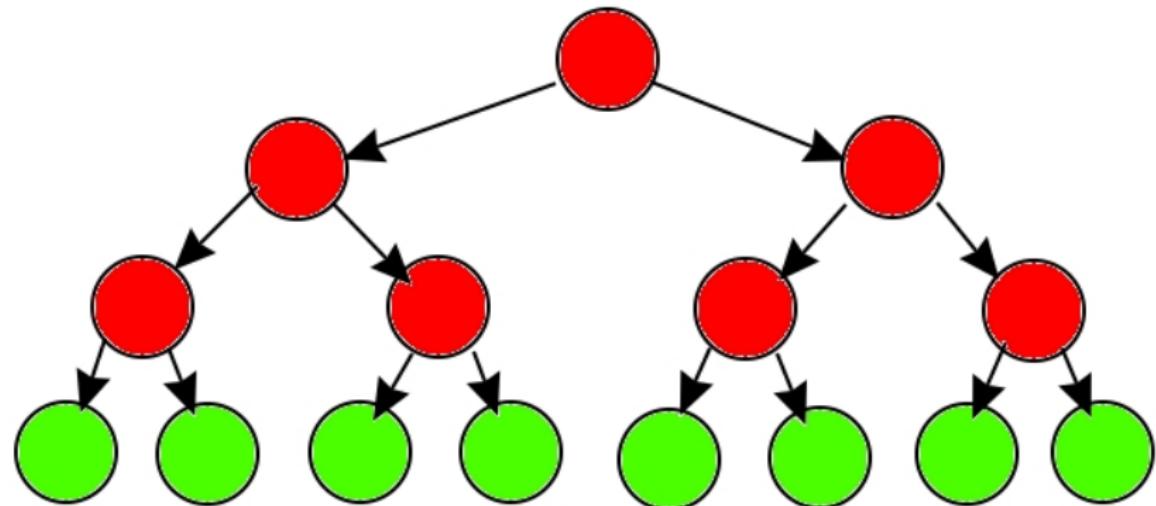
Breadth First Search (BFS)



Breadth First Search (BFS)



Breadth First Search (BFS)



Breadth First Search (BFS)

- The good:

Breadth First Search (BFS)

- The good:
 - Always finds the solution
 - Optimal route to the solution

Breadth First Search (BFS)

- The good:
 - Always finds the solution
 - Optimal route to the solution
- The bad:

Breadth First Search (BFS)

- The good:
 - Always finds the solution
 - Optimal route to the solution
- The bad:
 - Kinda dumb...

Breadth First Search (BFS)

- The good:
 - Always finds the solution
 - Optimal route to the solution
- The bad:
 - Kinda dumb...
- The ugly:

Breadth First Search (BFS)

- The good:
 - Always finds the solution
 - Optimal route to the solution
- The bad:
 - Kinda dumb...
- The ugly:
 - Memory consumption on larger graphs

Depth First Search (DFS)

```
closedNodes = {}
```

```
# Last In First Out
```

```
openNode = Stack()
```

```
openNodes.insert(getInitialState())
```

```
while openNodes not empty
```

```
    node = openNodes.get()
```

```
    if isGoalState(node)
```

```
        return node
```

```
    if node in closedNodes
```

```
        continue
```

```
    for childNode in getSuccessors(node)
```

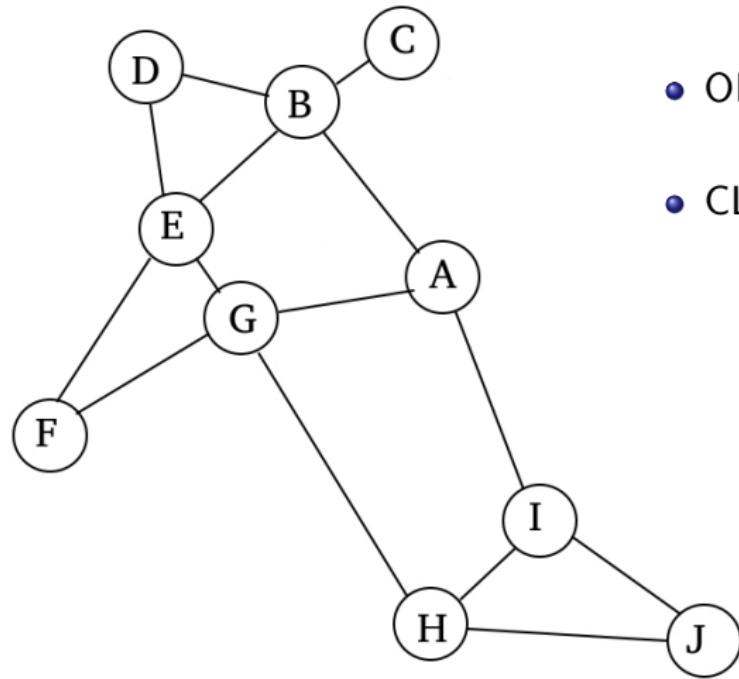
```
        if childNode not in closedNodes
```

```
            openNodes.insert(childNode)
```

```
            closedNodes.insert(node)
```

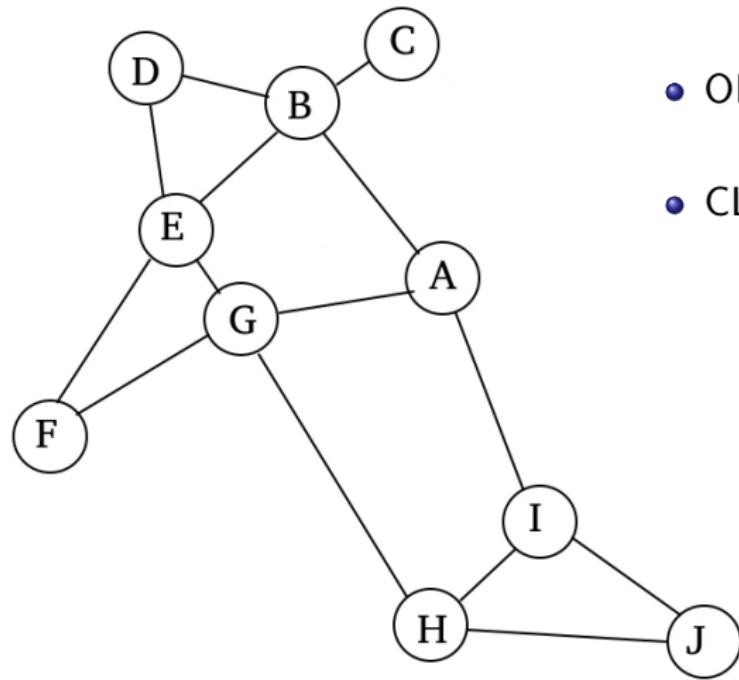
Depth First Search (DFS)

- Find a path from B to I
- OPEN: [B]
- CLOSED: {}



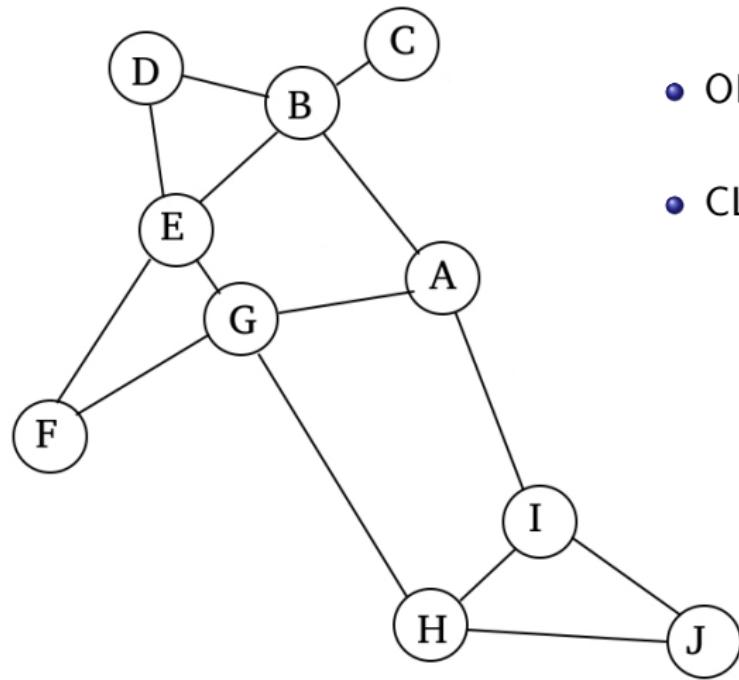
Depth First Search (DFS)

- Find a path from B to I
- OPEN: [C, D, E, A]
- CLOSED: { B }



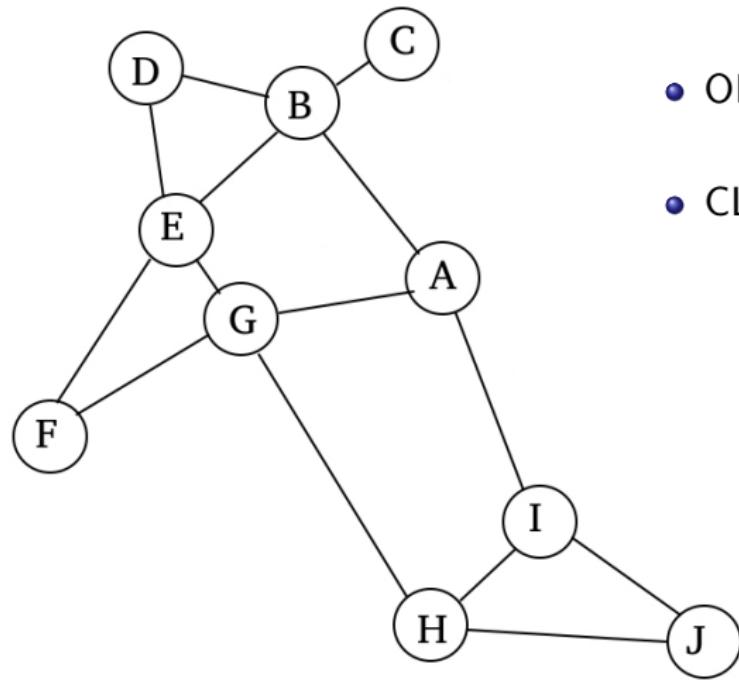
Depth First Search (DFS)

- Find a path from B to I
- OPEN: [D, E, A]
- CLOSED: { B, C }



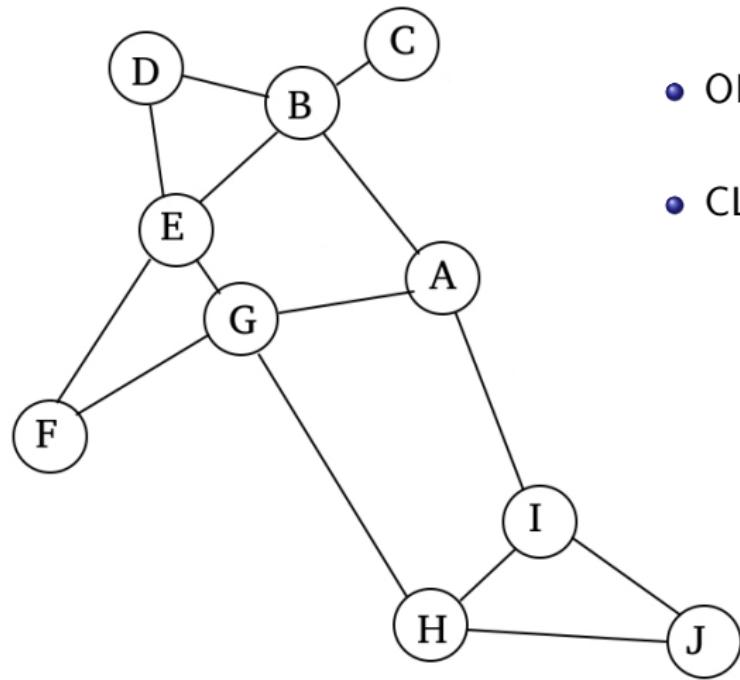
Depth First Search (DFS)

- Find a path from B to I
- OPEN: [E, A]
- CLOSED: { B, C, D }

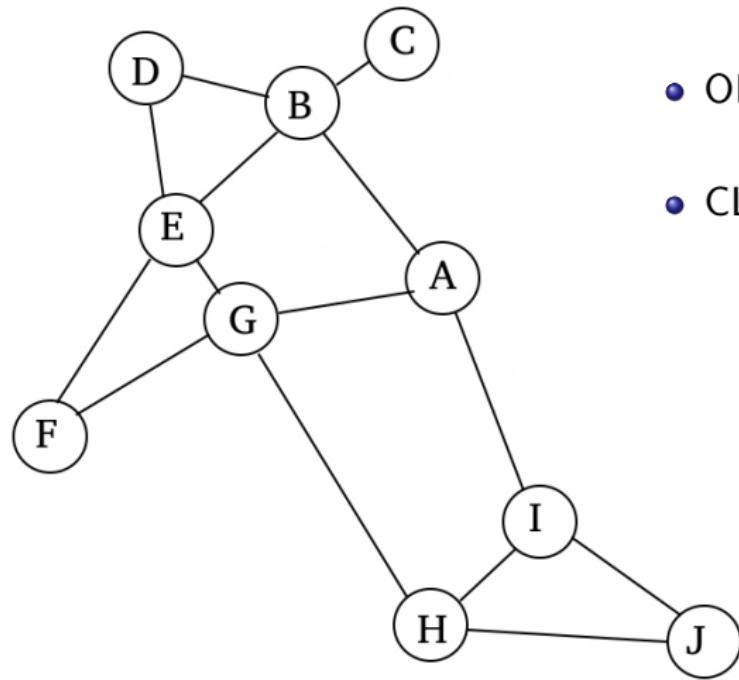


Depth First Search (DFS)

- Find a path from B to I
- OPEN: [F, G, A]
- CLOSED: { B, C, D, E }



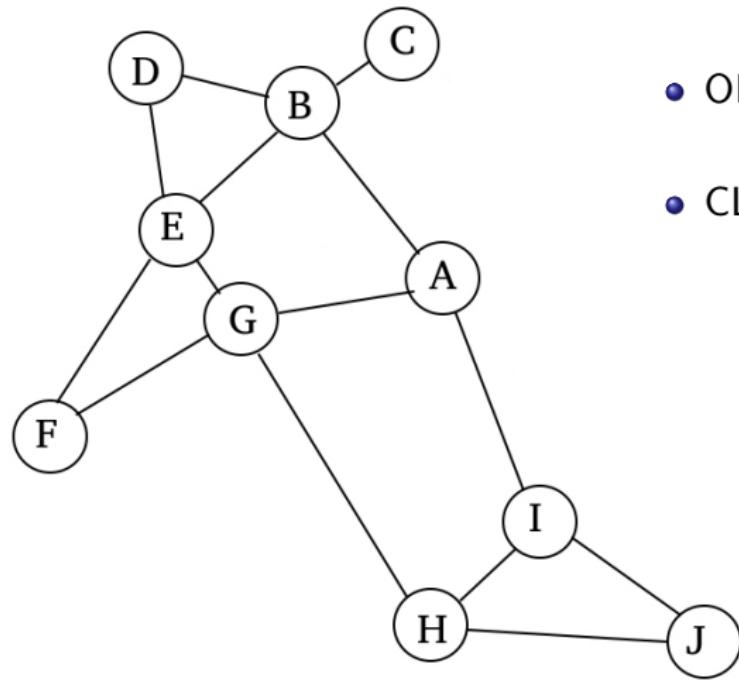
Depth First Search (DFS)



- Find a path from B to I
- OPEN: [G, A]
- CLOSED: { B, C, D, E, F }

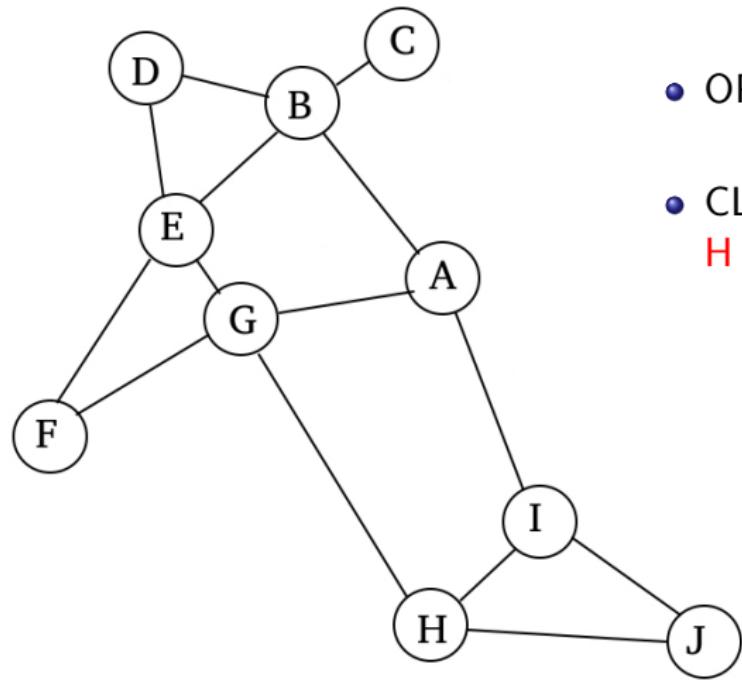
Depth First Search (DFS)

- Find a path from B to I
- OPEN: [H, A]
- CLOSED: { B, C, D, E, F, G }



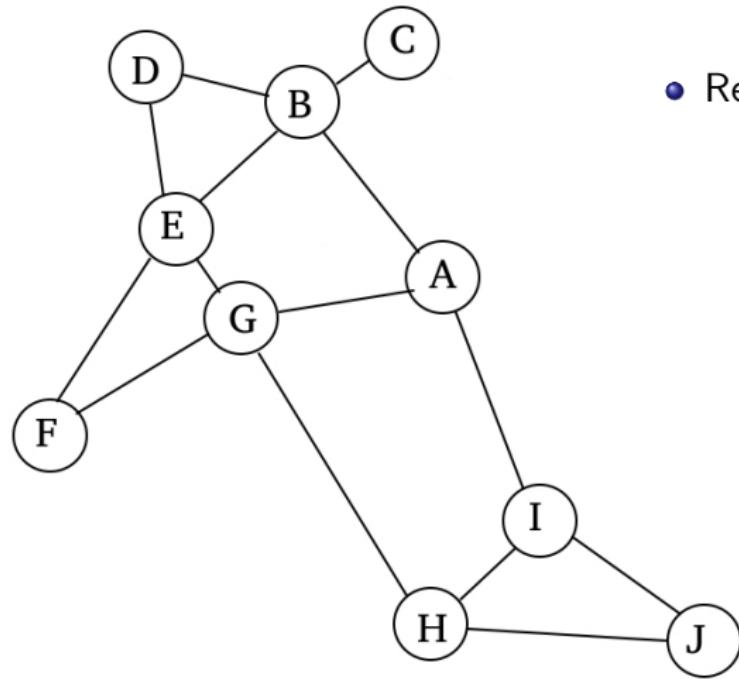
Depth First Search (DFS)

- Find a path from B to I
- OPEN: [I, J, A]
- CLOSED: { B, C, D, E, F, G, H }

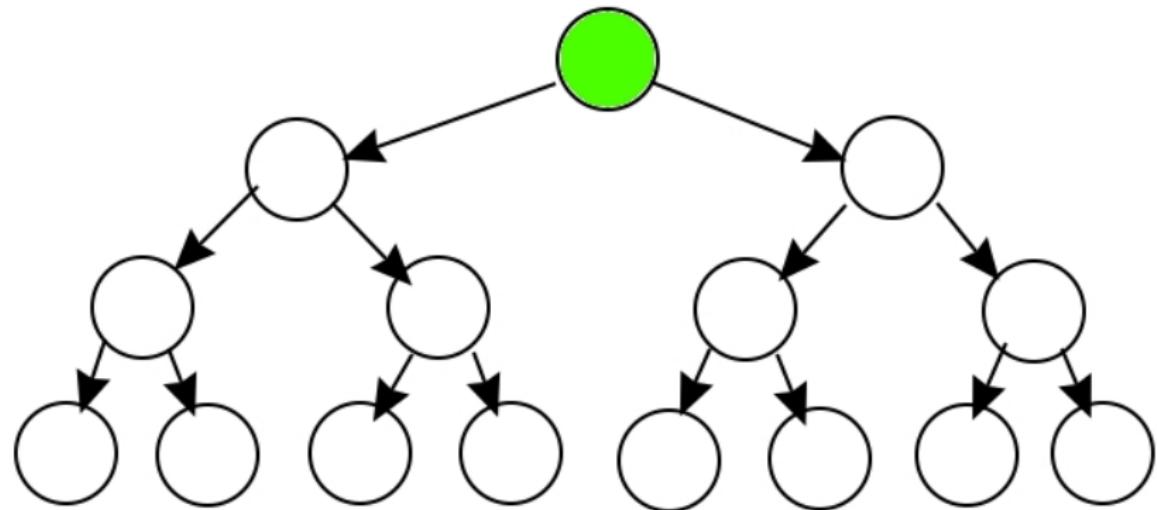


Depth First Search (DFS)

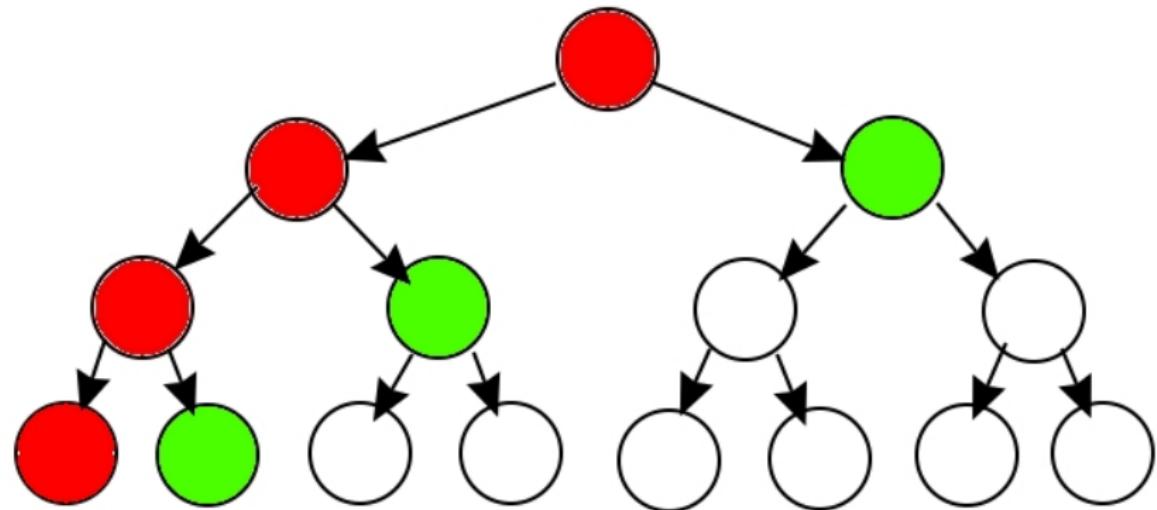
- Find a path from B to I
- Reached goal I



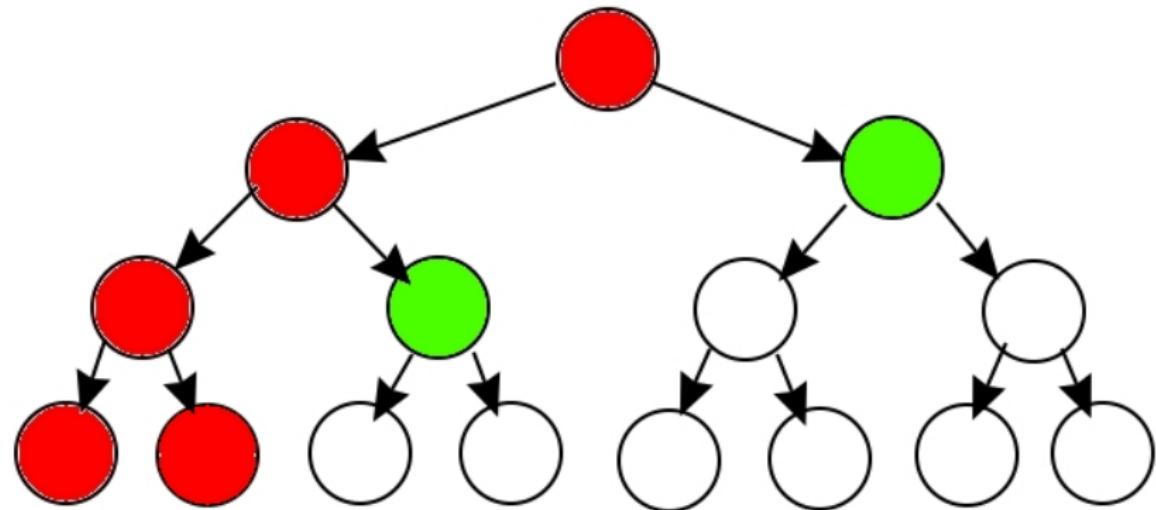
Depth First Search (DFS)



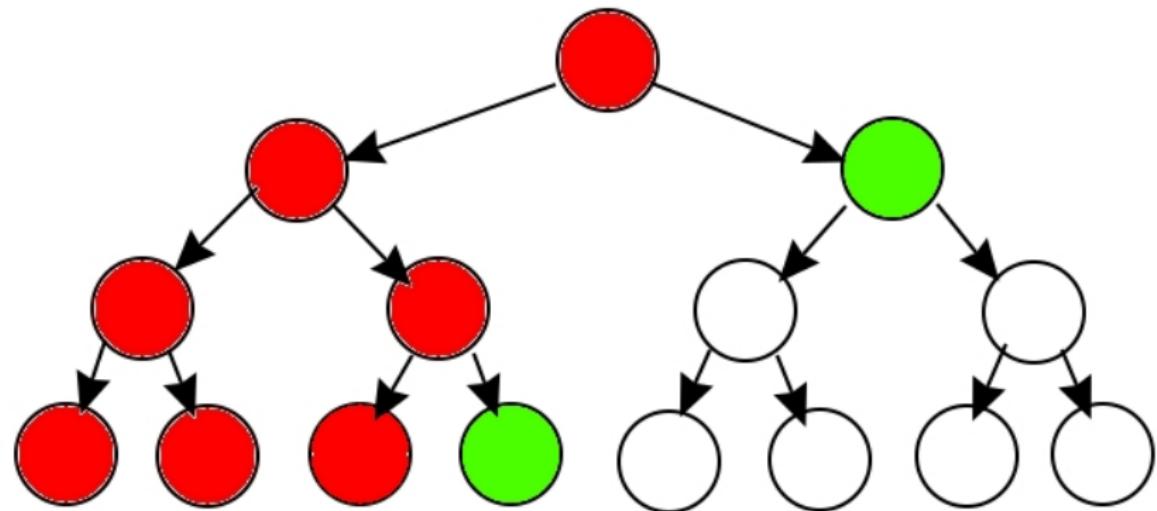
Depth First Search (DFS)



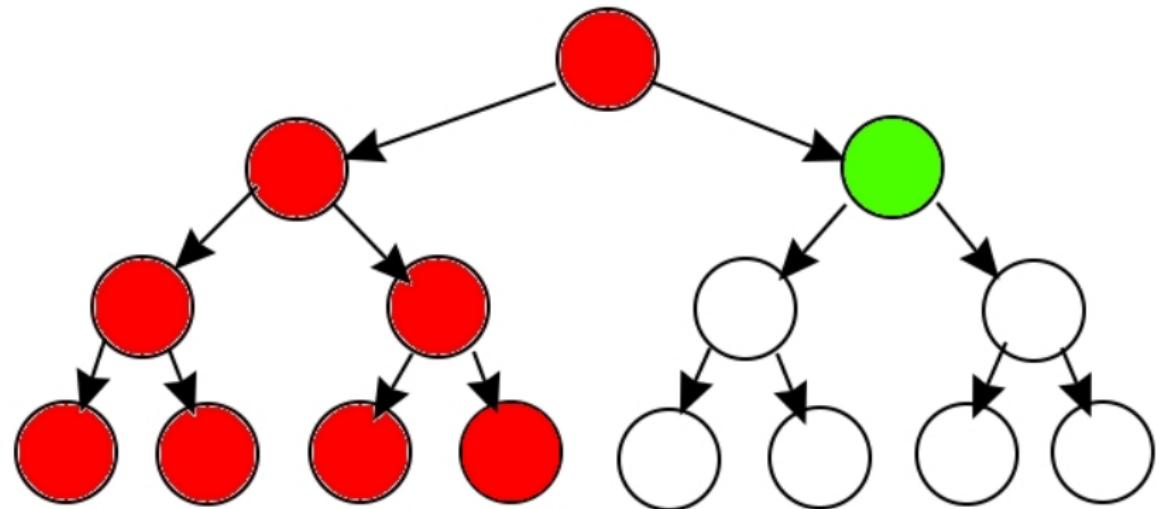
Depth First Search (DFS)



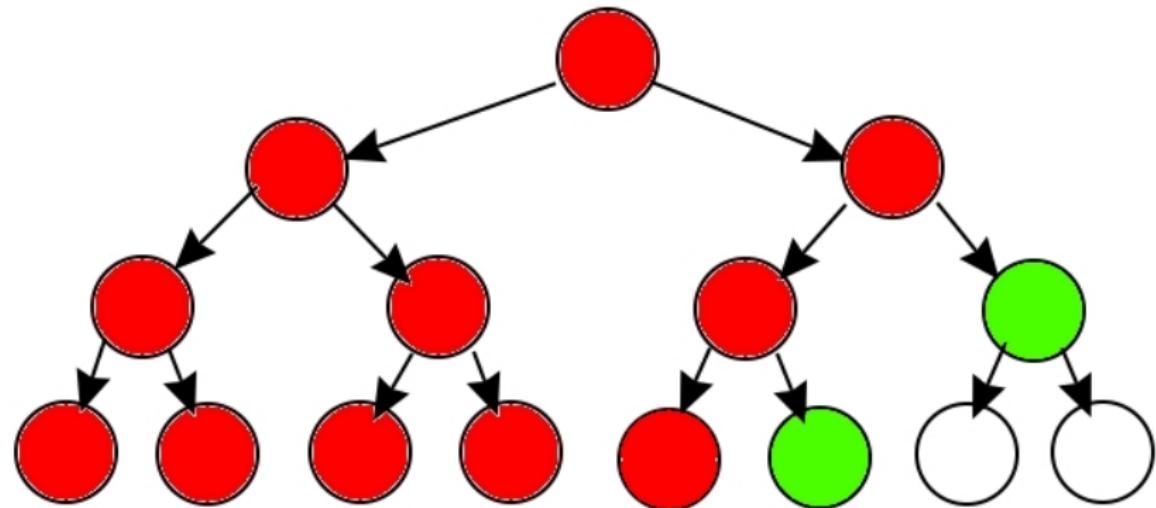
Depth First Search (DFS)



Depth First Search (DFS)



Depth First Search (DFS)



Depth First Search (DFS)

- The good:

Depth First Search (DFS)

- The good:
 - Always finds the solution (when taking care not to revisit states multiple times)

Depth First Search (DFS)

- The good:
 - Always finds the solution (when taking care not to revisit states multiple times)
- The bad:

Depth First Search (DFS)

- The good:
 - Always finds the solution (when taking care not to revisit states multiple times)
- The bad:
 - Not the optimal solution though

Depth First Search (DFS)

- The good:
 - Always finds the solution (when taking care not to revisit states multiple times)
- The bad:
 - Not the optimal solution though
- The ugly:

Depth First Search (DFS)

- The good:
 - Always finds the solution (when taking care not to revisit states multiple times)
- The bad:
 - Not the optimal solution though
- The ugly:
 - Kinda dumb...er

Uniform Cost Search (UCS)

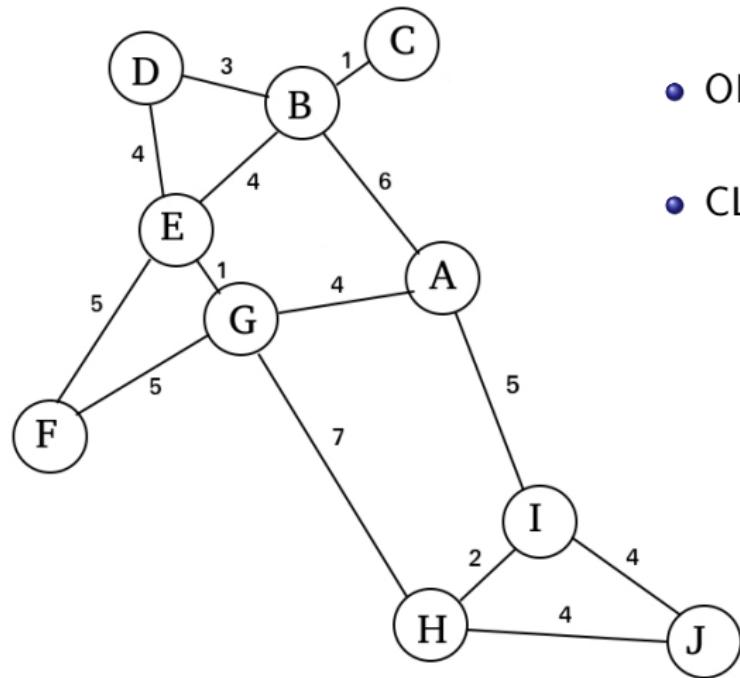
```
closedNodes = {}

# Elements sorted by priority
openNode = PriorityQueue()

openNodes.insert((0, getInitialState()))
while openNodes not empty
    cost, node = openNodes.get()
    if isGoalState(node)
        return node
    if node in closedNodes
        continue
    for childNode, actionCost in getSuccessors(node)
        if childNode not in closedNodes
            openNodes.insert((cost + actionCost, childNode))
    closedNodes.insert(node)
```

Uniform Cost Search (UCS)

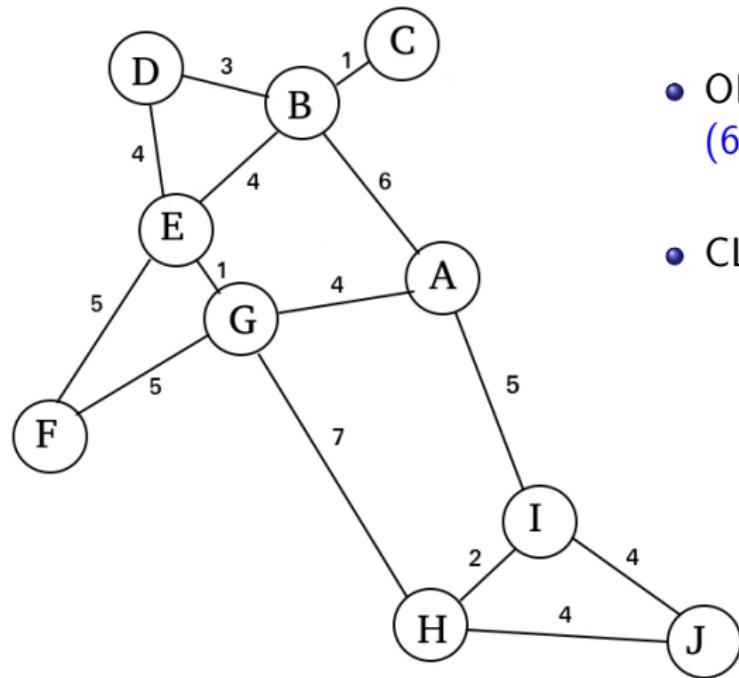
- Find a path from B to I



- OPEN: [(0, B)]

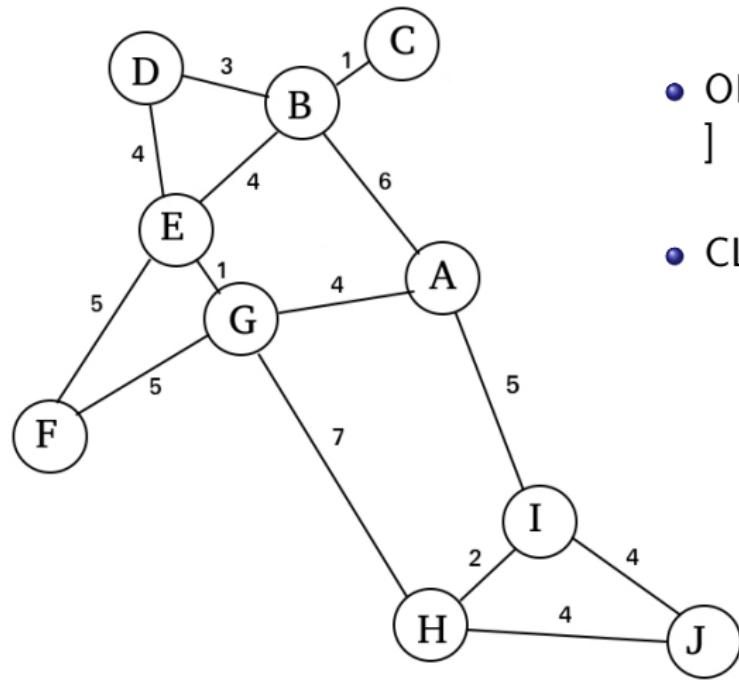
- CLOSED: {}

Uniform Cost Search (UCS)



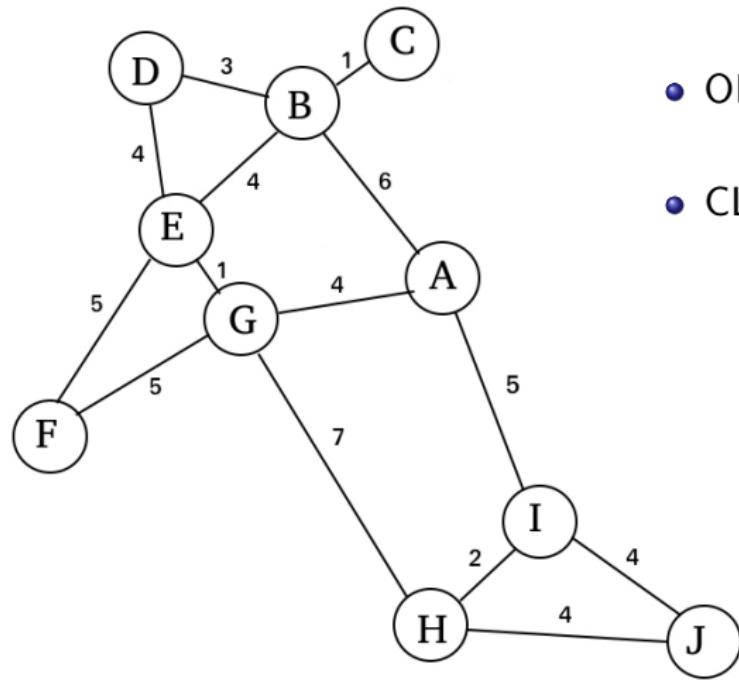
- Find a path from B to I
- OPEN: [(1, C), (3, D), (4, E), (6, A)]
- CLOSED: { B }

Uniform Cost Search (UCS)



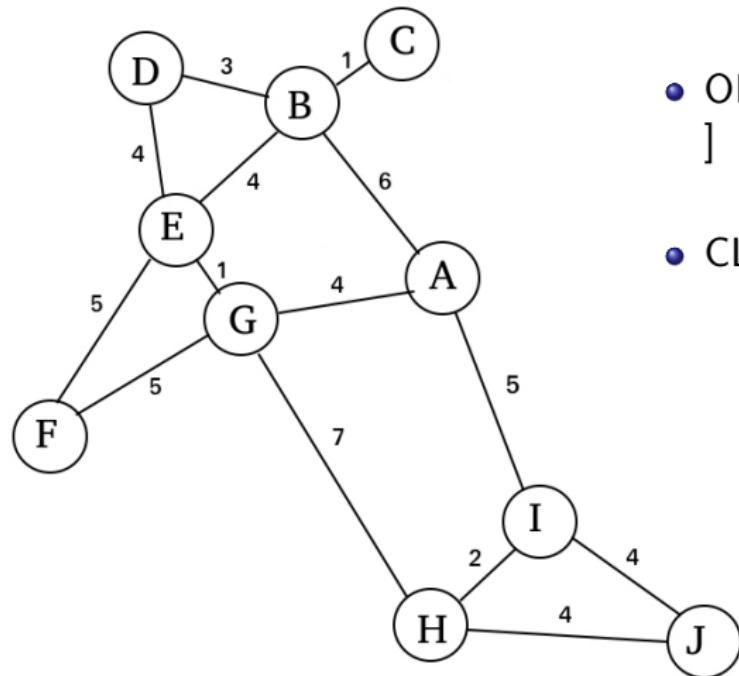
- Find a path from B to I
- OPEN: [(3, D), (4, E), (6, A)]
- CLOSED: { B, C }

Uniform Cost Search (UCS)



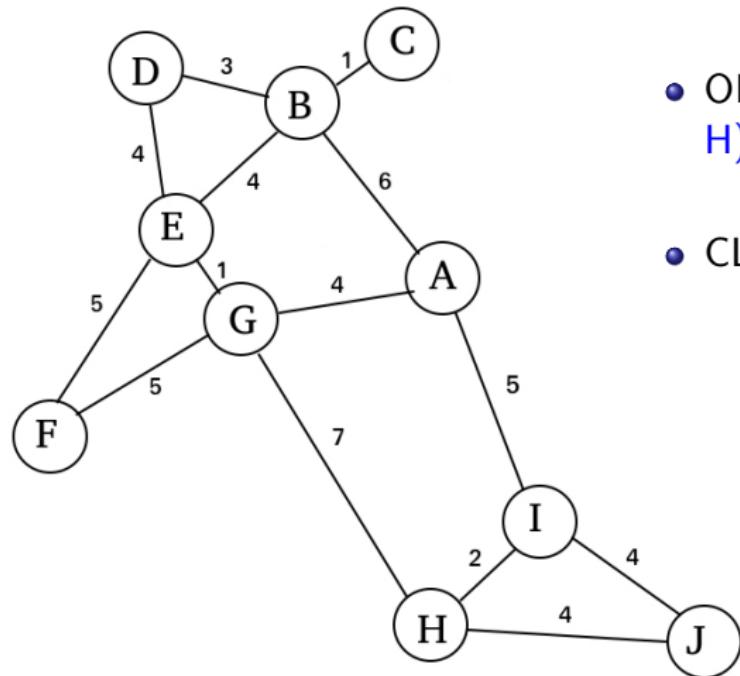
- Find a path from B to I
- OPEN: [(4, E), (6, A)]
- CLOSED: { B, C, D }

Uniform Cost Search (UCS)



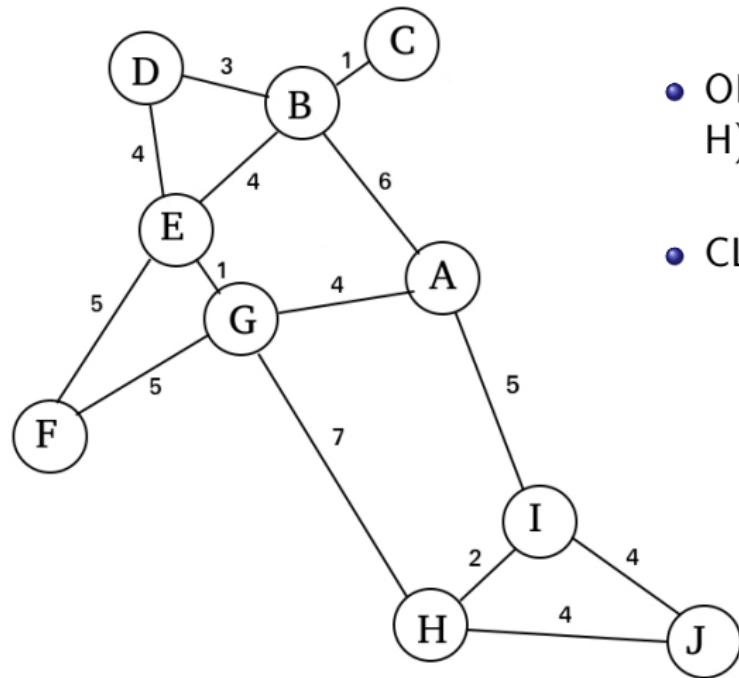
- Find a path from B to I
- OPEN: [(5, G), (6, A), (9, F)]
- CLOSED: { B, C, D, E }

Uniform Cost Search (UCS)



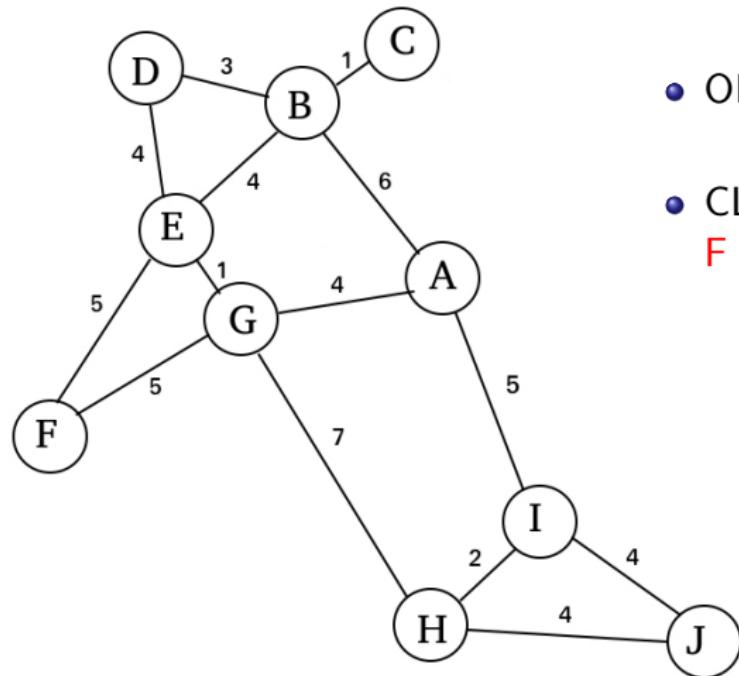
- Find a path from B to I
- OPEN: [(6, A), (9, F), (12, H)]
- CLOSED: { B, C, D, E, G }

Uniform Cost Search (UCS)



- Find a path from B to I
- OPEN: [(9, F), (11, I), (12, H)]
- CLOSED: { B, C, D, E, G, A }

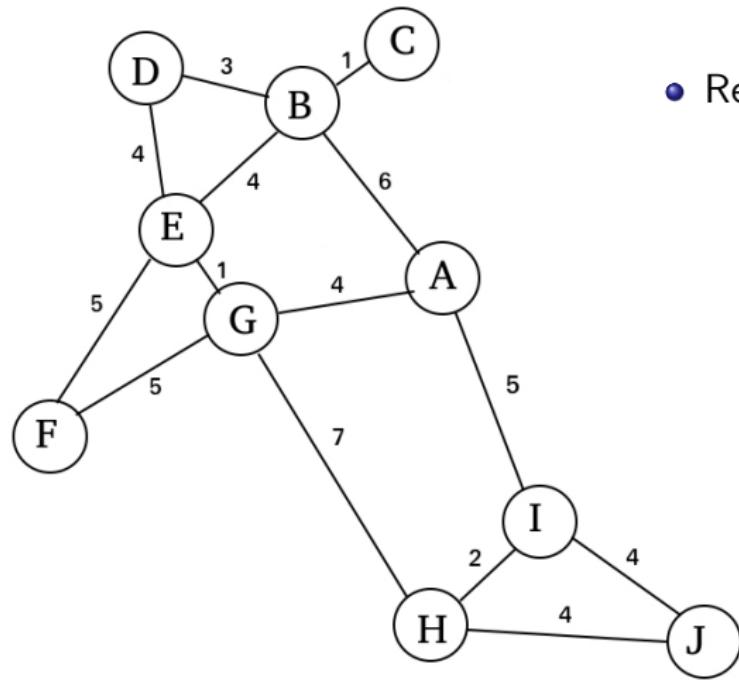
Uniform Cost Search (UCS)



- Find a path from B to I
- OPEN: [(11, I), (12, H)]
- CLOSED: { B, C, D, E, G, A, F }

Uniform Cost Search (UCS)

- Find a path from B to I
- Reached goal I



Uniform Cost Search (UCS)

- Same as BFS but with variable cost transitions

Heuristic algorithms

- You can give them some hints

Heuristic algorithms

- You can give them some hints
- They may or may not listen to you though...

What is a heuristic?



- Estimation of the distance to the goal
- Examples:

What is a heuristic?



- Estimation of the distance to the goal
- Examples:
 - Manhattan distance on a grid

What is a heuristic?



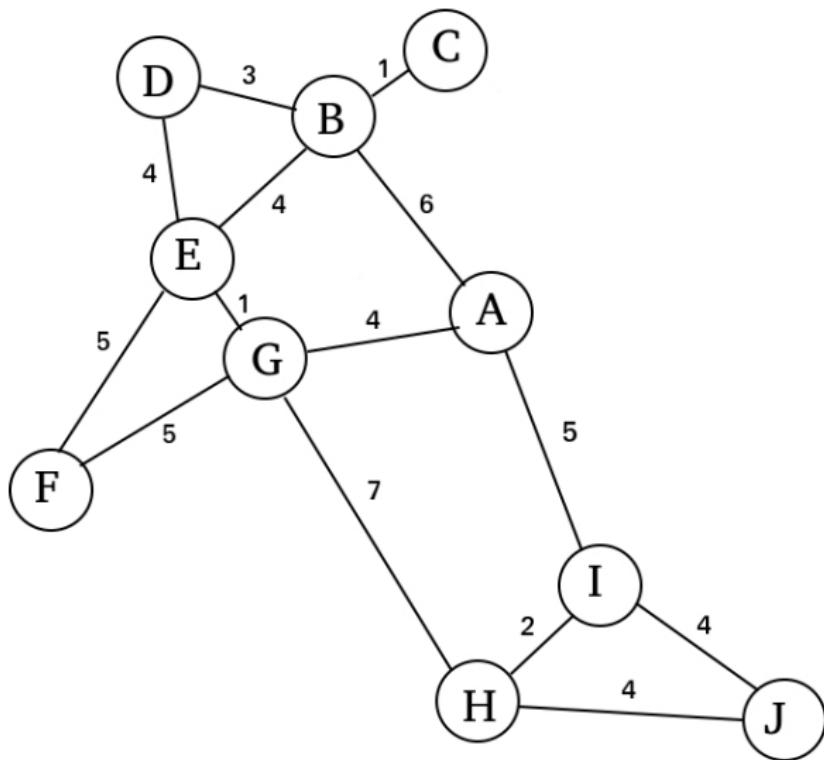
- Estimation of the distance to the goal
- Examples:
 - Manhattan distance on a grid
 - Euclidean distance (air distance) between cities

What is a heuristic?



- Estimation of the distance to the goal
- Examples:
 - Manhattan distance on a grid
 - Euclidean distance (air distance) between cities
 - In our example let's use minimal number of jumps needed to get to the goal

What is a heuristic?



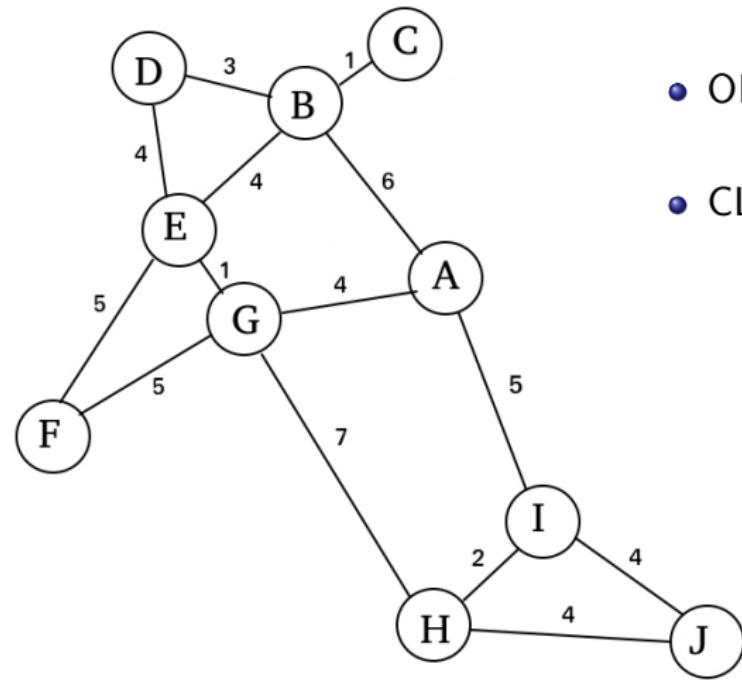
Greedy Search

```
closedNodes = {}

# Elements sorted by priority
openNode = PriorityQueue()

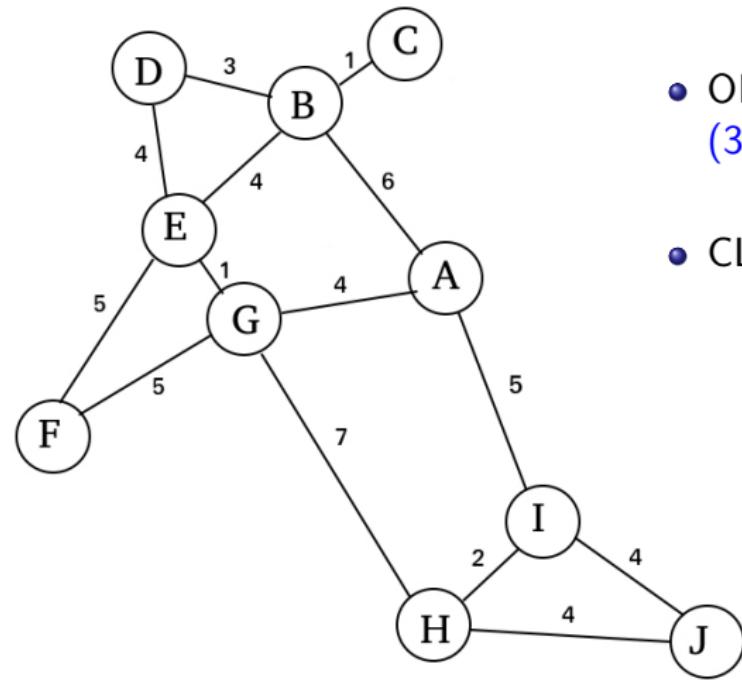
openNodes.insert((0, getInitialState()))
while openNodes not empty
    cost, node = openNodes.get()
    if isGoalState(node)
        return node
    if node in closedNodes
        continue
    for childNode, actionCost in getSuccessors(node)
        if childNode not in closedNodes
            openNodes.insert((h(childNode), childNode))
    closedNodes.insert(node)
```

Greedy Search



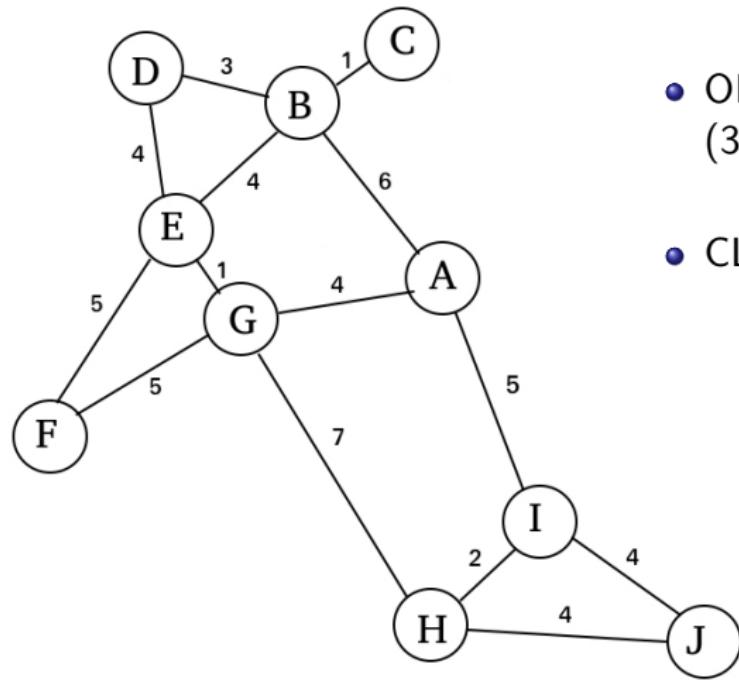
- Find a path from B to I
- OPEN: [(0, B)]
- CLOSED: {}

Greedy Search



- Find a path from B to I
- OPEN: [(1, A), (3, D), (3, E), (3, C)]
- CLOSED: { B }

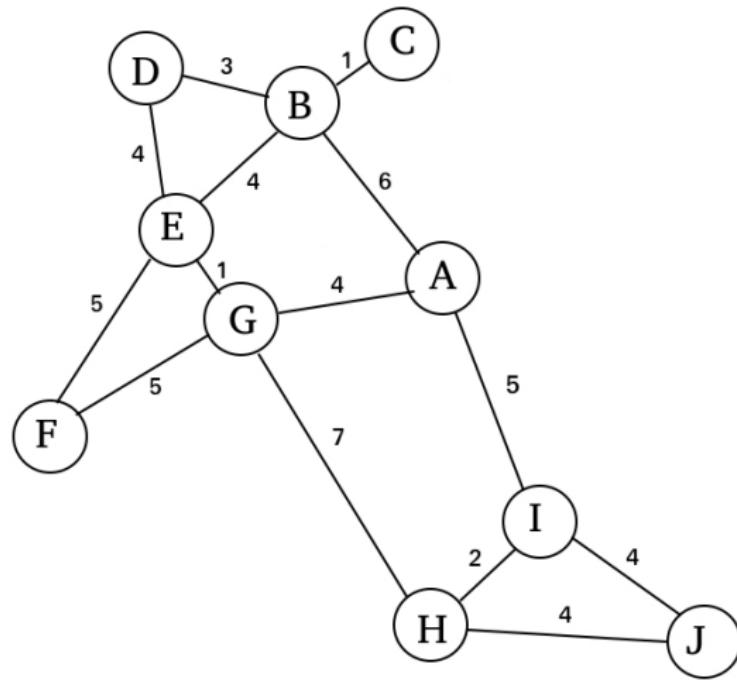
Greedy Search



- Find a path from B to I
- OPEN: [(0, I), (3, D), (3, E), (3, C)]
- CLOSED: { B, A }

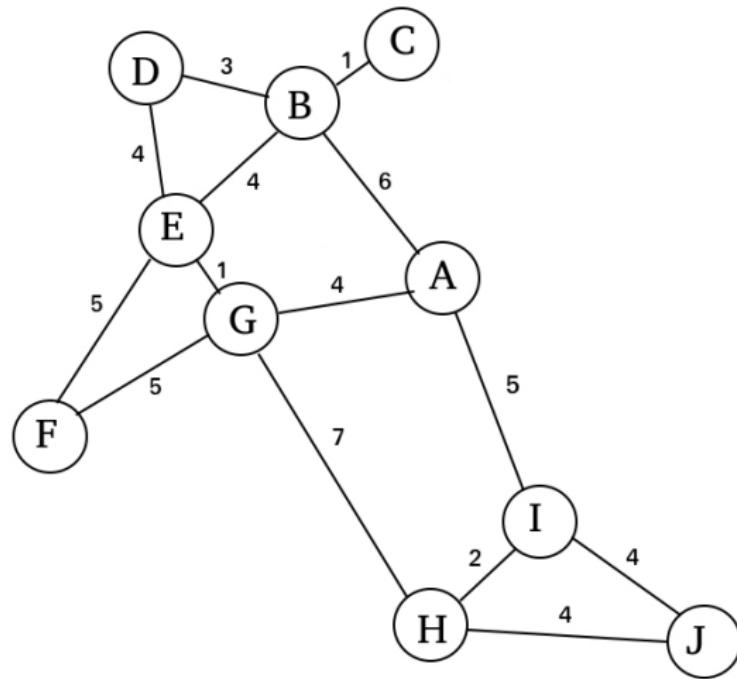
Greedy Search

- Find a path from B to I



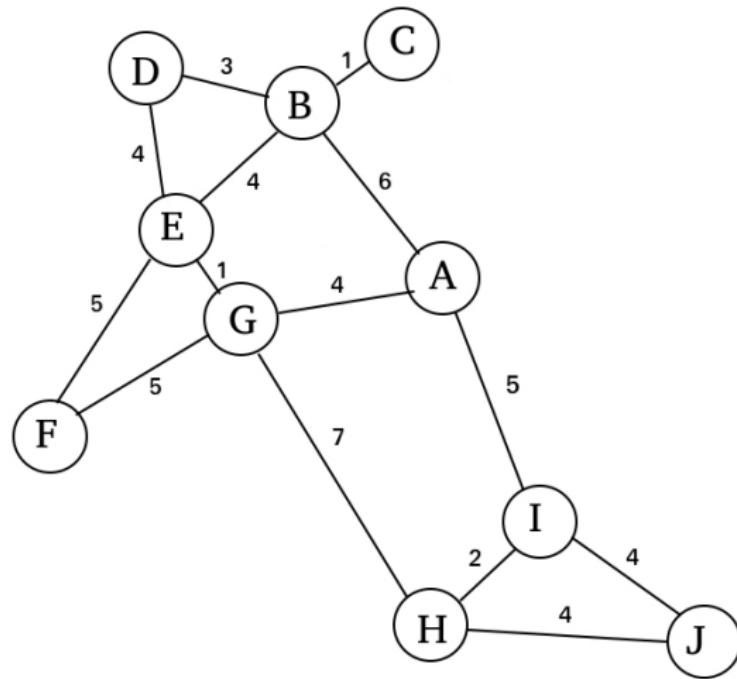
Greedy Search

- Find a path from B to I



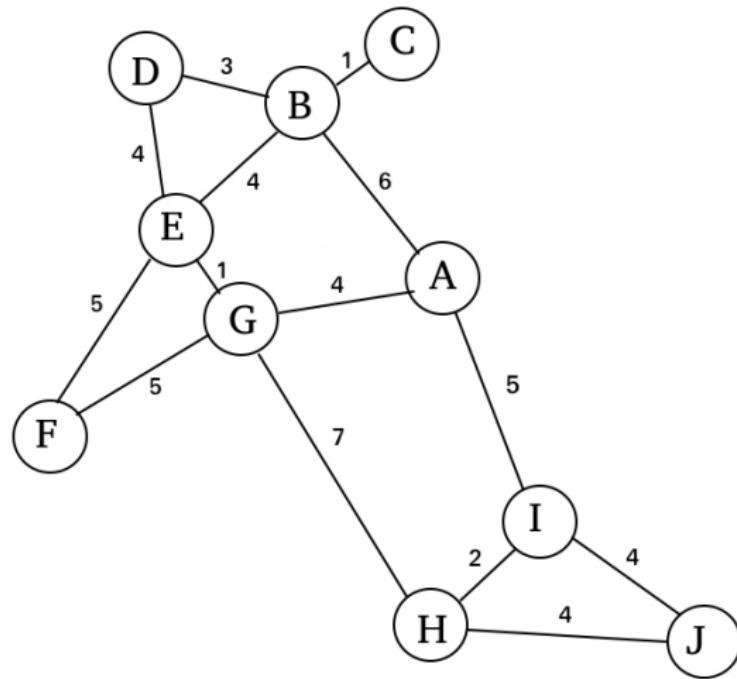
Greedy Search

- Find a path from B to I



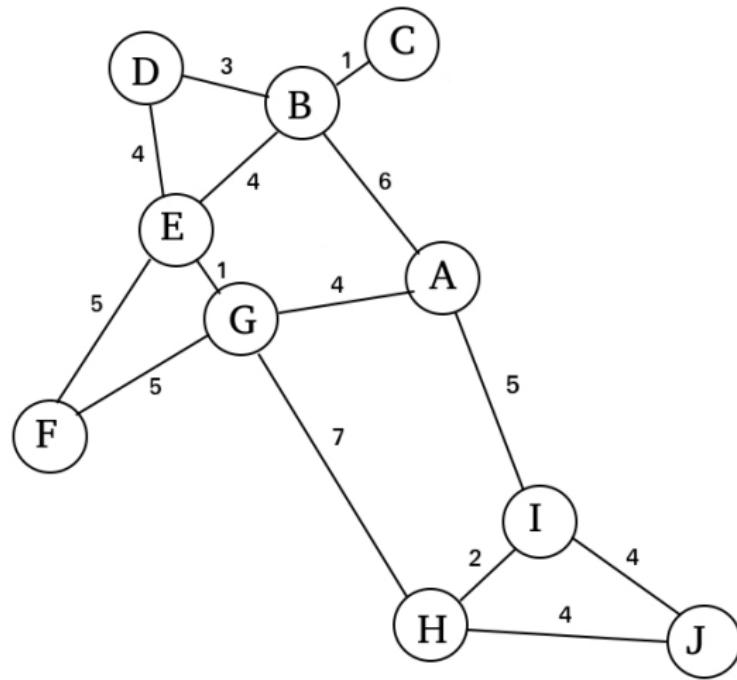
Greedy Search

- Find a path from B to I



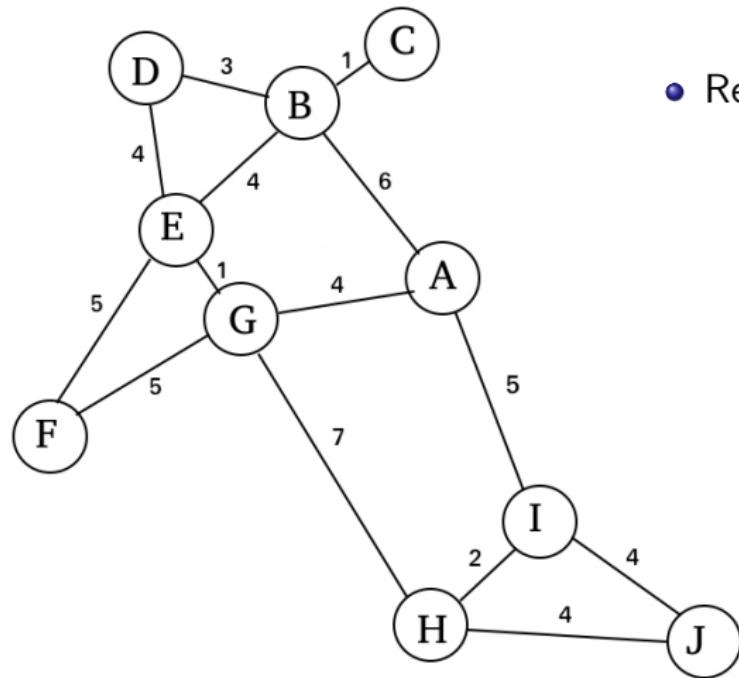
Greedy Search

- Find a path from B to I



Greedy Search

- Find a path from B to I
- Reached goal I



Greedy Search

- The good:

Greedy Search

- The good:
 - Always finds the solution (when taking care not to revisit states multiple times)
 - Quickly finds a solution

Greedy Search

- The good:
 - Always finds the solution (when taking care not to revisit states multiple times)
 - Quickly finds a solution
- The bad:

Greedy Search

- The good:
 - Always finds the solution (when taking care not to revisit states multiple times)
 - Quickly finds a solution
- The bad:
 - Not the optimal solution though

Greedy Search

- The good:
 - Always finds the solution (when taking care not to revisit states multiple times)
 - Quickly finds a solution
- The bad:
 - Not the optimal solution though
- The ugly:

Greedy Search

- The good:
 - Always finds the solution (when taking care not to revisit states multiple times)
 - Quickly finds a solution
- The bad:
 - Not the optimal solution though
- The ugly:
 - Trusts you a bit too much...

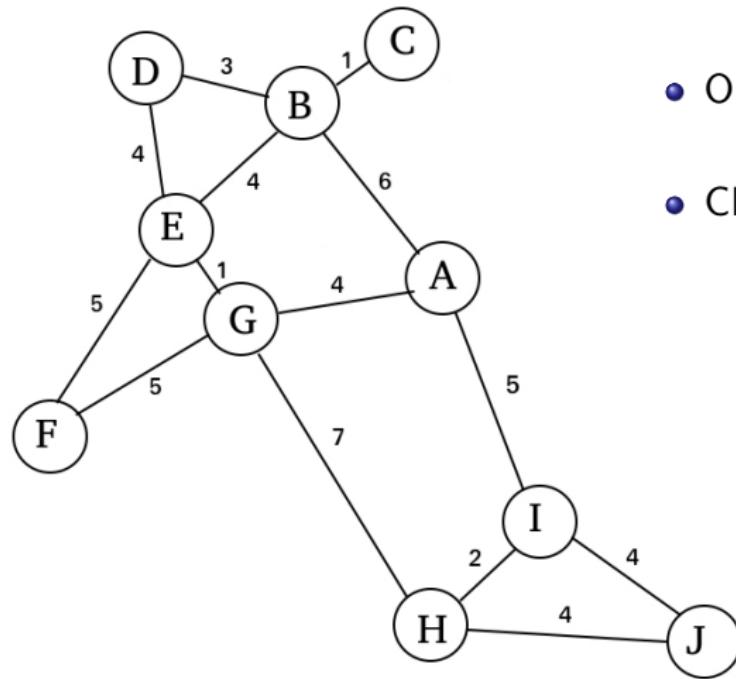
A* Search

```
closedNodes = {}

# Elements sorted by priority
openNode = PriorityQueue()

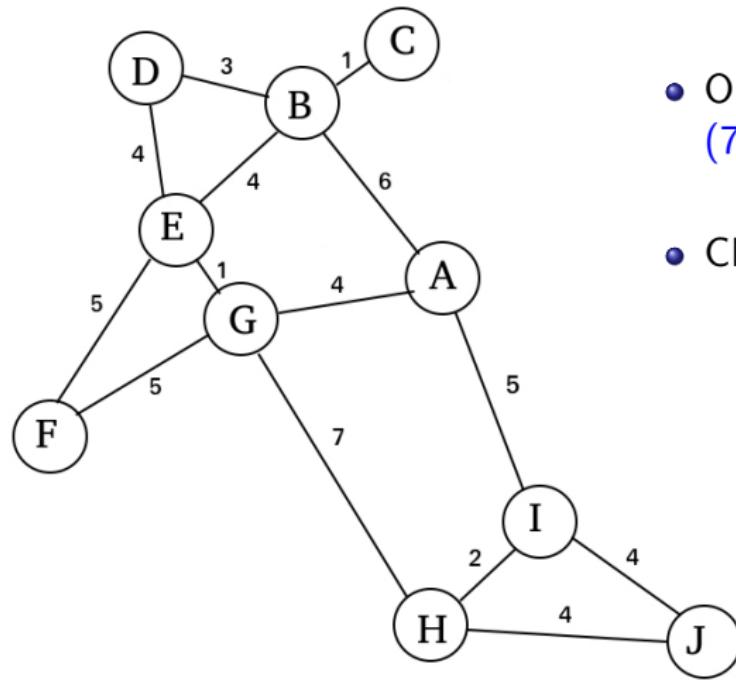
openNodes.insert((0, 0, getInitialState()))
while openNodes not empty
    _, cost, node = openNodes.get()
    if isGoalState(node)
        return node
    if node in closedNodes
        continue
    for childNode, actionCost in getSuccessors(node)
        if childNode not in closedNodes
            totalCost = actionCost + cost
            openNodes.insert((totalCost + h(childNode), totalCost,
                childNode))
```

A* Search



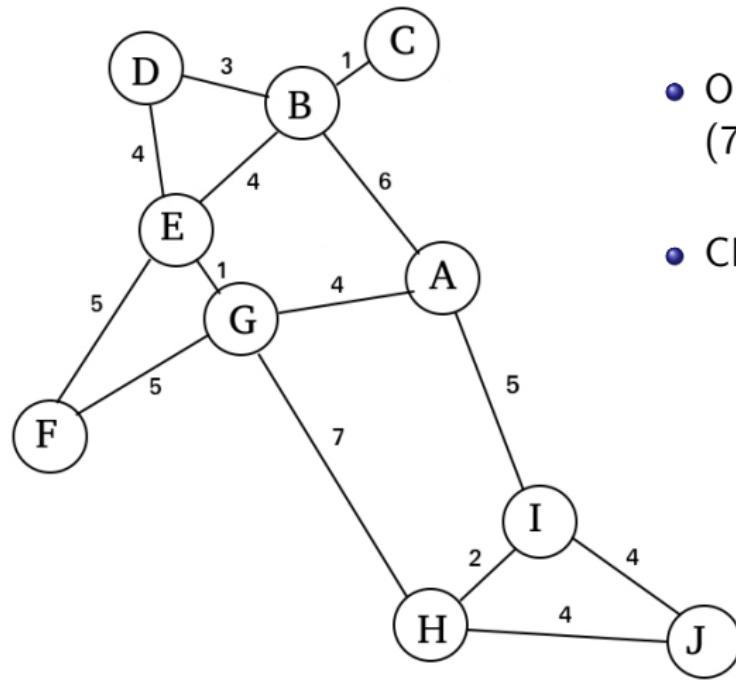
- Find a path from B to I
- OPEN: [(0, 0, B)]
- CLOSED: {}

A* Search



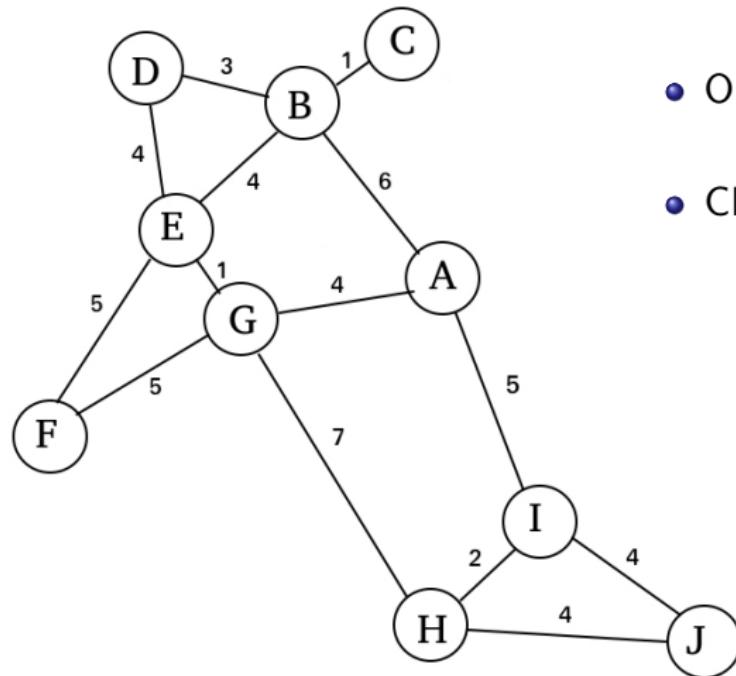
- Find a path from B to I
- OPEN: [(4, 1, C), (6, 3, D), (7, 4, E), (7, 6, A)]
- CLOSED: { B }

A* Search



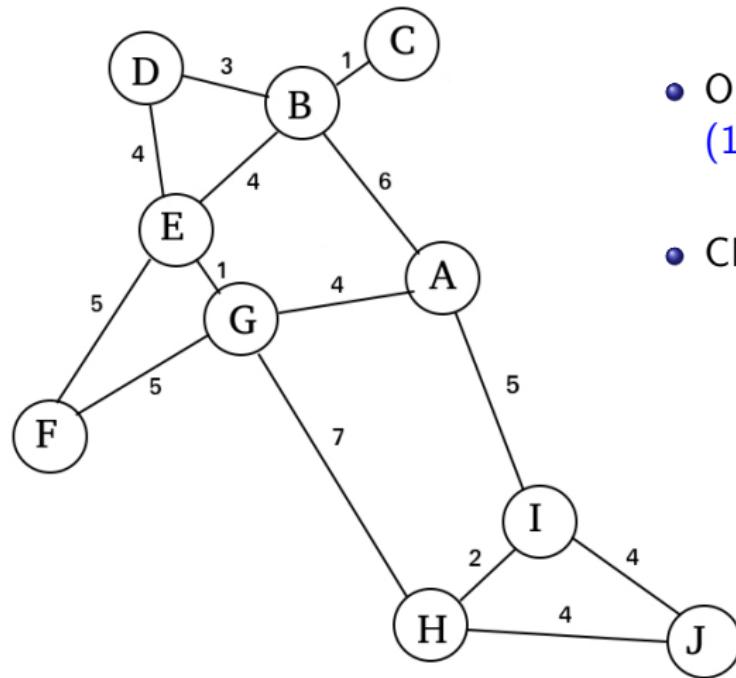
- Find a path from B to I
- OPEN: [(6, 3, D), (7, 4, E), (7, 6, A)]
- CLOSED: { B, C }

A* Search



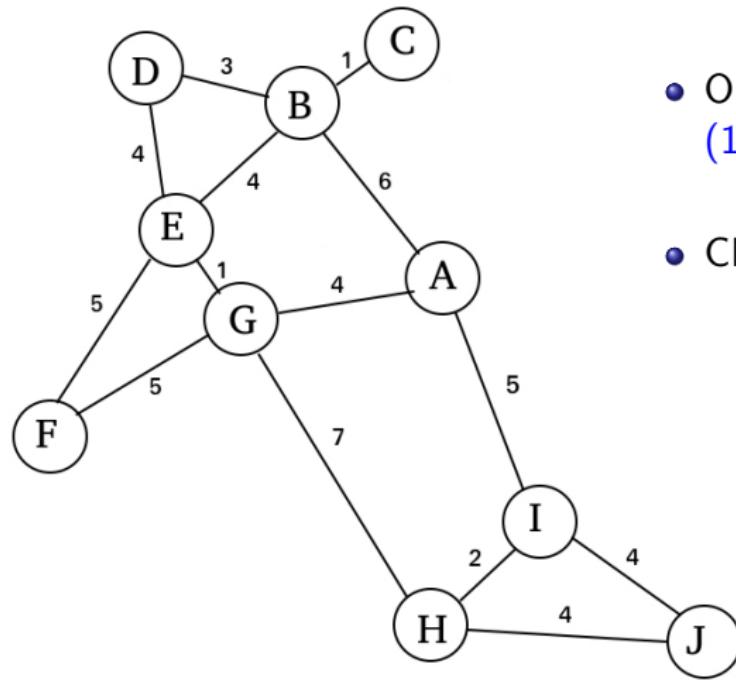
- Find a path from B to I
- OPEN: [(7, 4, E), (7, 6, A)]
- CLOSED: { B, C, D }

A* Search



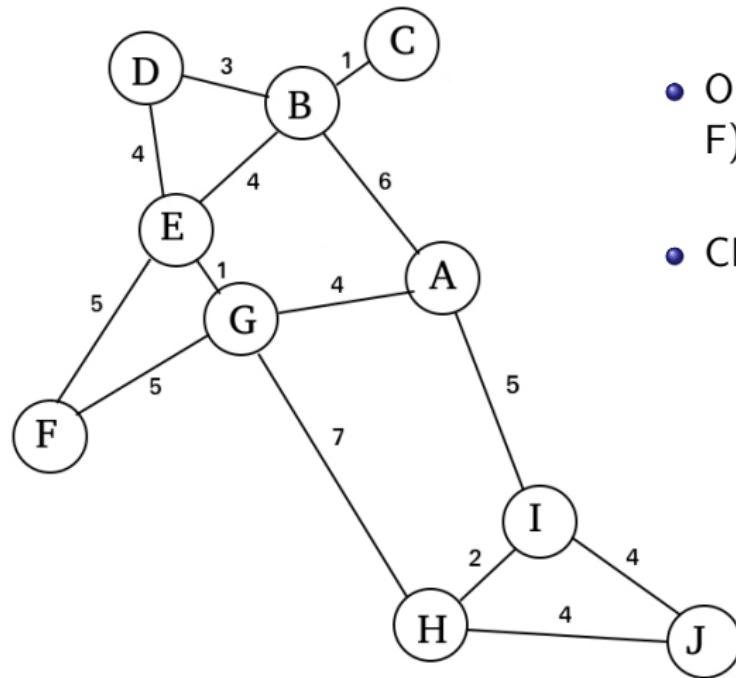
- Find a path from B to I
- OPEN: [(7, 5, G), (7, 6, A), (12, 9, F)]
- CLOSED: { B, C, D, E }

A* Search



- Find a path from B to I
- OPEN: [(7, 6, A), (12, 9, F), (13, 12, H)]
- CLOSED: { B, C, D, E, G }

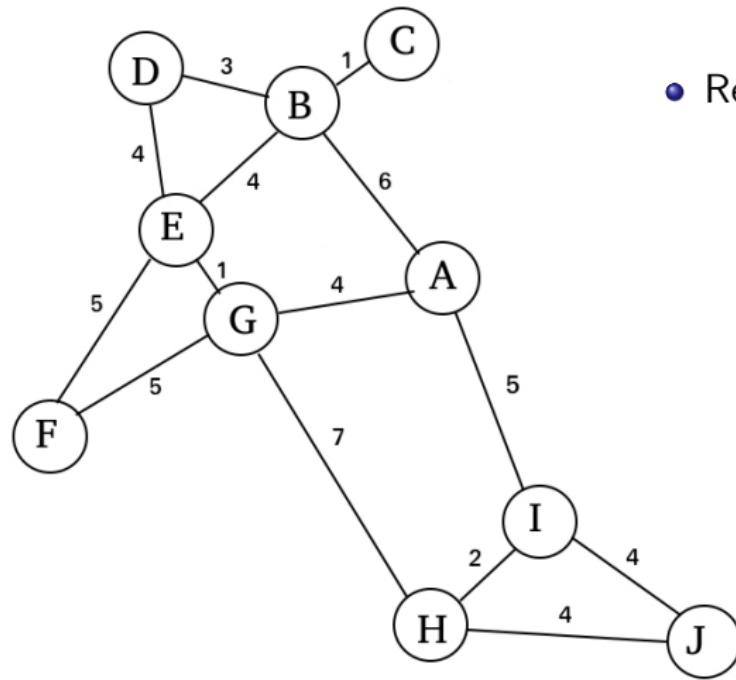
A* Search



- Find a path from B to I
- OPEN: [(11, 11, I), (12, 9, F), (13, 12, H)]
- CLOSED: { B, C, D, E, G, A }

A* Search

- Find a path from B to I
- Reached goal I



Greedy Search

- Always finds the solution
- If the given heuristic is **optimistic** the solution is optimal

Live demonstration

Is there anything better?

Is there anything better?

Well...depends...

Is there anything better?

Well...depends...

- Dijkstra's algorithm (same as UCS but without an exact "goal") - mostly used in networking

Is there anything better?

Well...depends...

- Dijkstra's algorithm (same as UCS but without an exact "goal") - mostly used in networking
- D* search and variations - mostly used in robotics

Is there anything better?

Well...depends...

- Dijkstra's algorithm (same as UCS but without an exact "goal") - mostly used in networking
- D* search and variations - mostly used in robotics
- A* search and variations - most often the best solution for a video game pathfinding

Pathfinding in video games

Alright there's an
obvious grid...

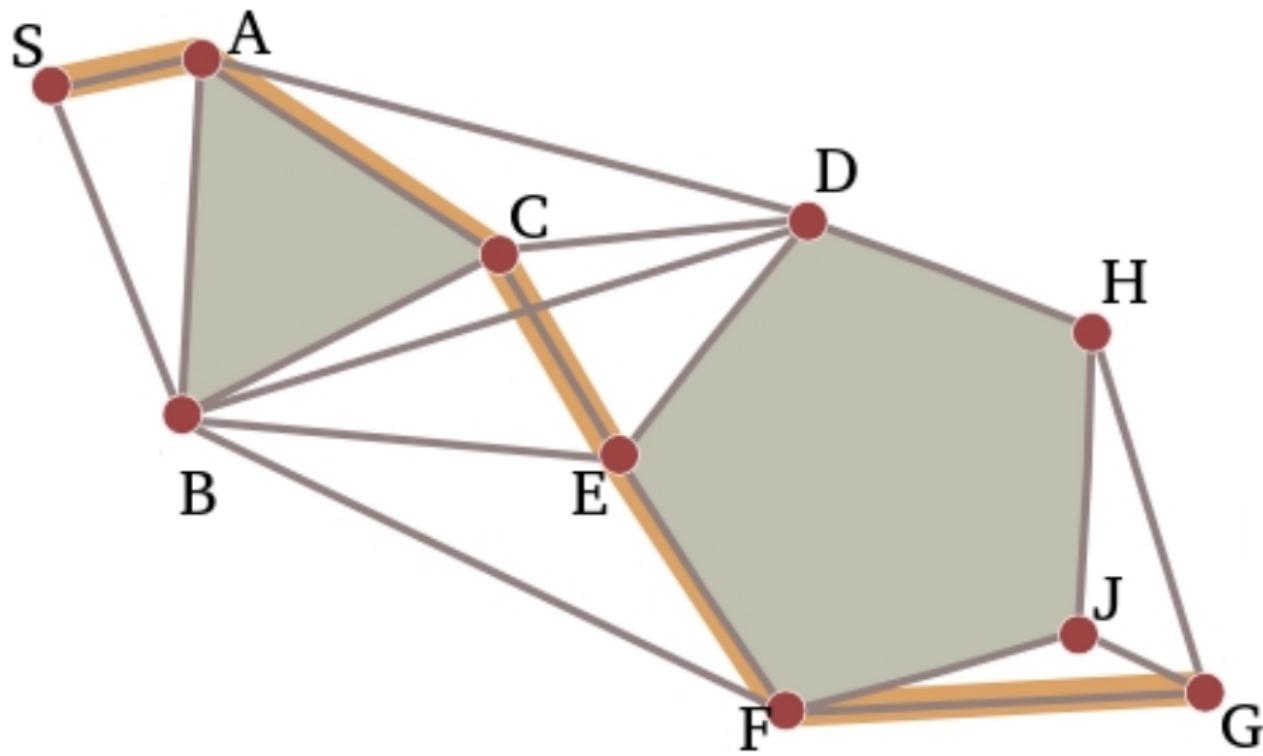


Pathfinding in video games

...now what?



Visibility graph



Divide and conquer

- Divide the whole grid into several smaller pieces
- Agents don't need to find a path on the whole grid at once
- Easier to modify the graph in case of frequent smaller changes in the terrain

Any questions?

Plz no difficult ones