ILP / 6

Cristina Adriana Alexandru, PhD

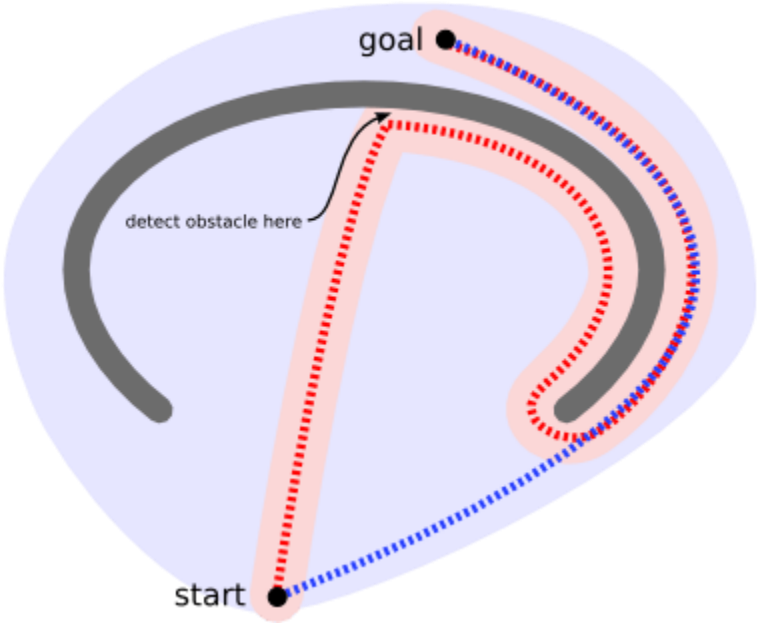
Slides adapted from: Changjian Li

This lecture

- Path finding (i.e. graph search) algorithms
  - What path finding means
  - Representing maps
  - Breadth first search algorithm
  - Dijkstra algorithm
  - A* algorithm

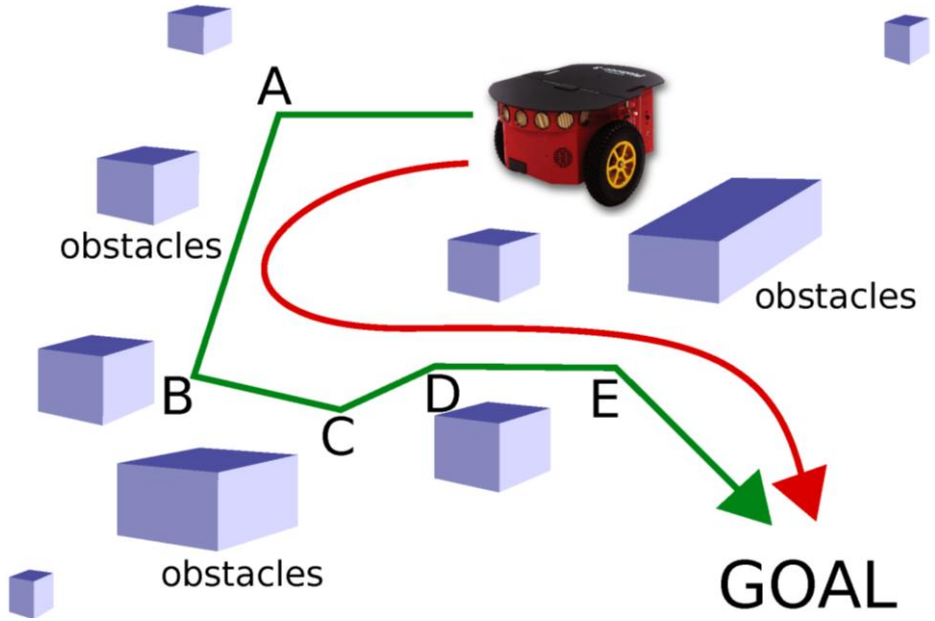
Pathfinding

- Find a path from one location (*[source]*) to (*[goal]*)



Pathfinding is Ubiquitous

- Robotics

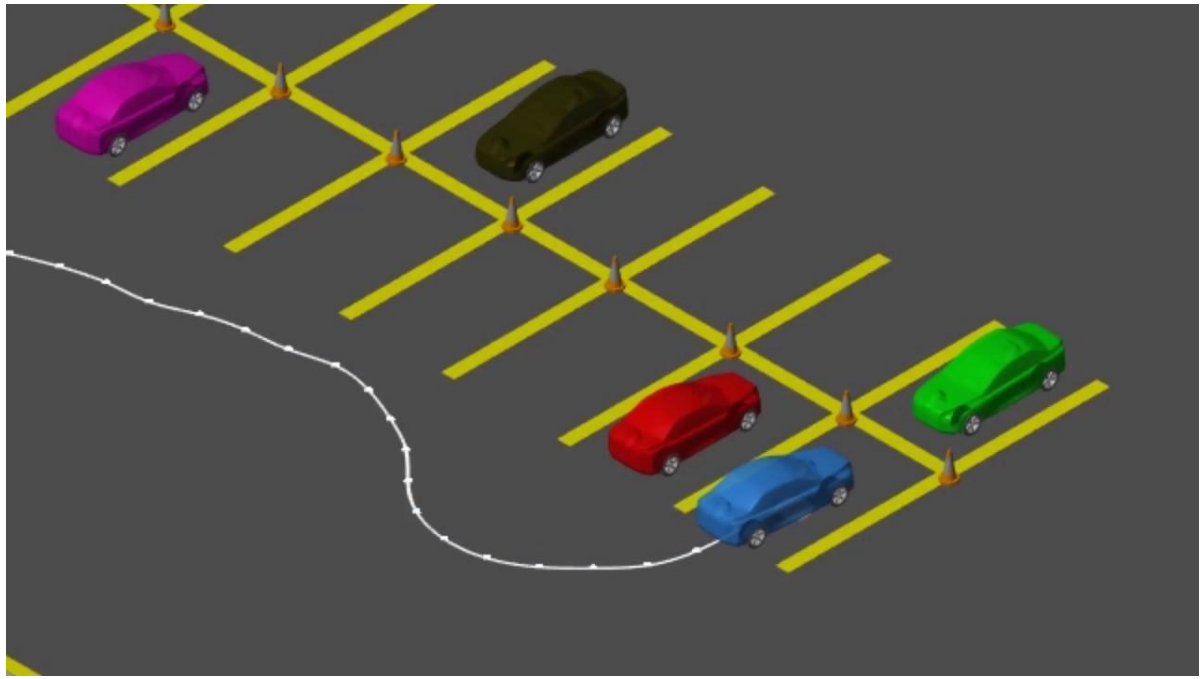


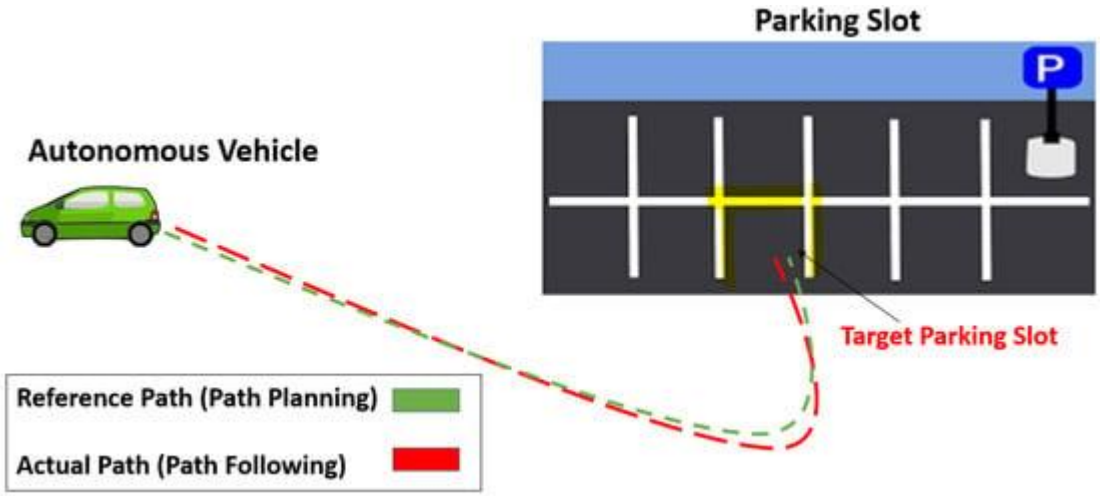
Pathfinding is Ubiquitous

- Warehouse

Pathfinding is Ubiquitous

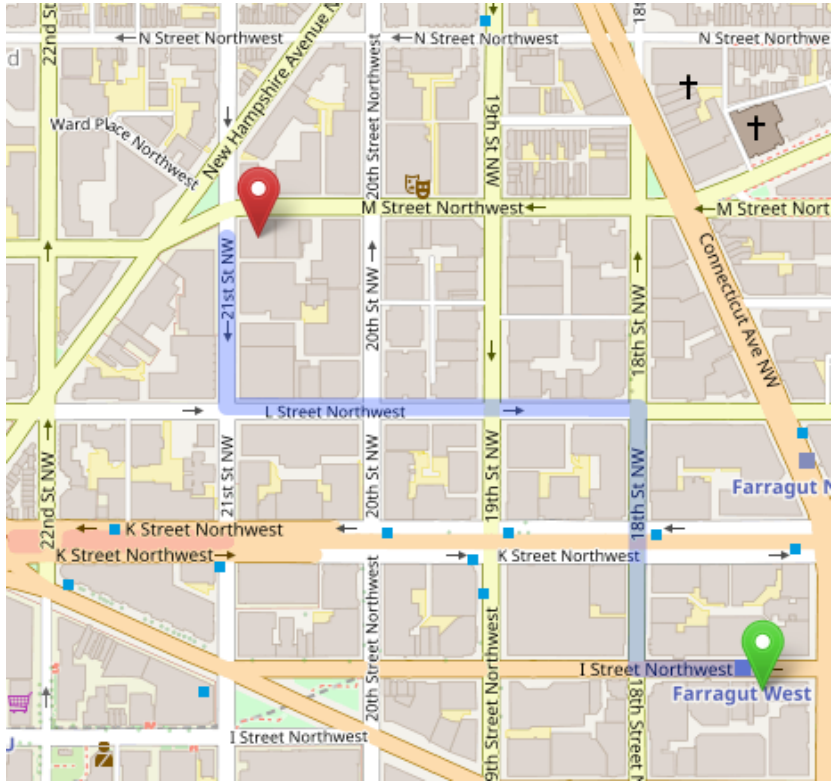
- Autonomous parking

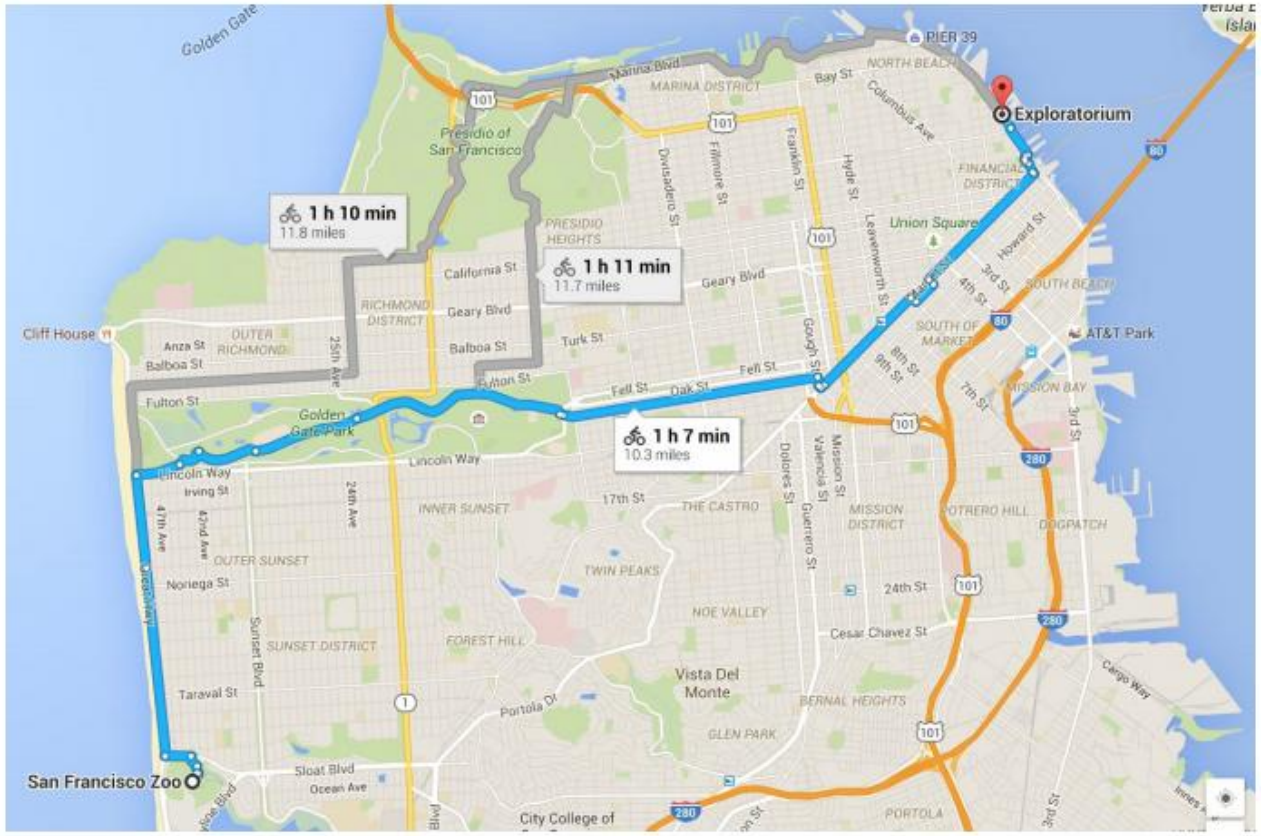




Pathfinding is Ubiquitous

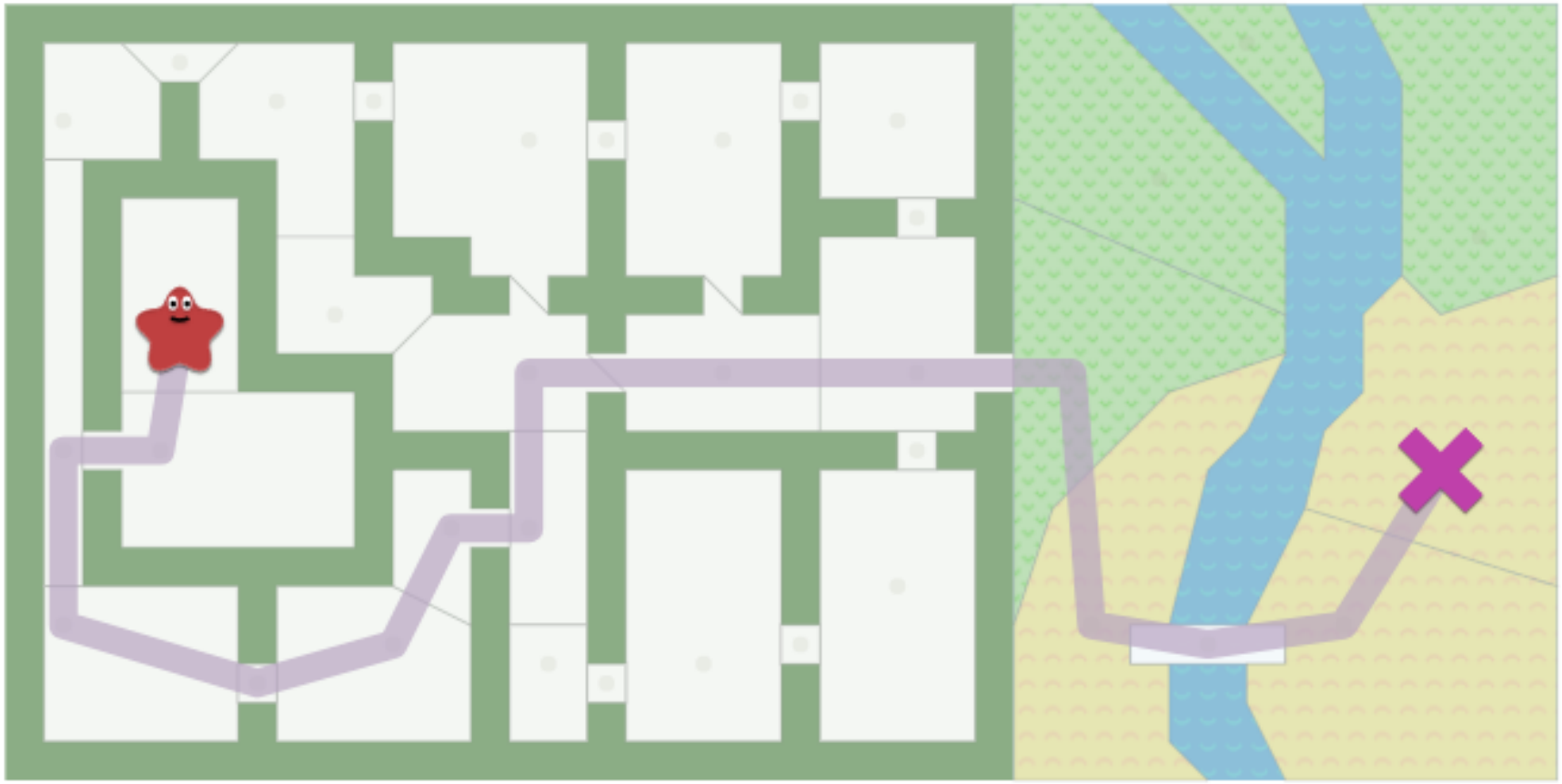
- Map route search





Pathfinding is Ubiquitous

- Games

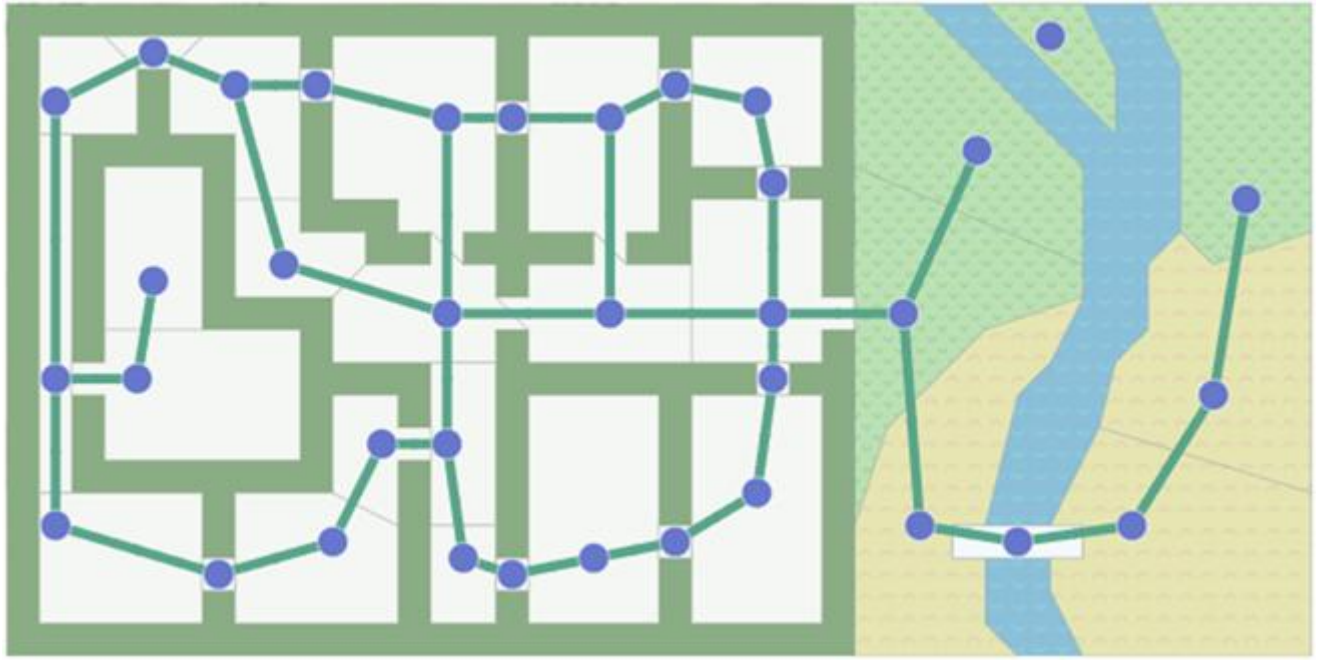


Pathfinding Goal(s)

- Shortest distance
- Least amount of travel time
- Lowest resource consumption
  - e.g., fuel, money

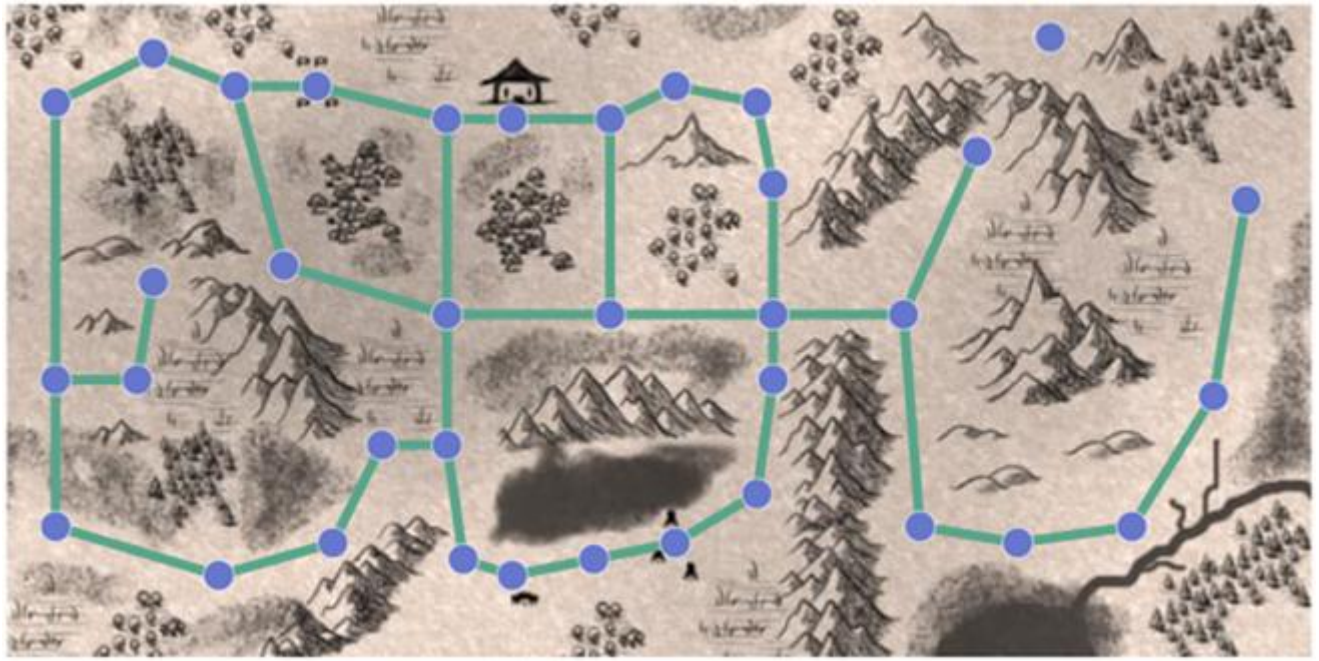
Pathfinding algorithms

- Graph search algorithms
  - Map is represented as a graph
  - Input: graph with nodes and edges



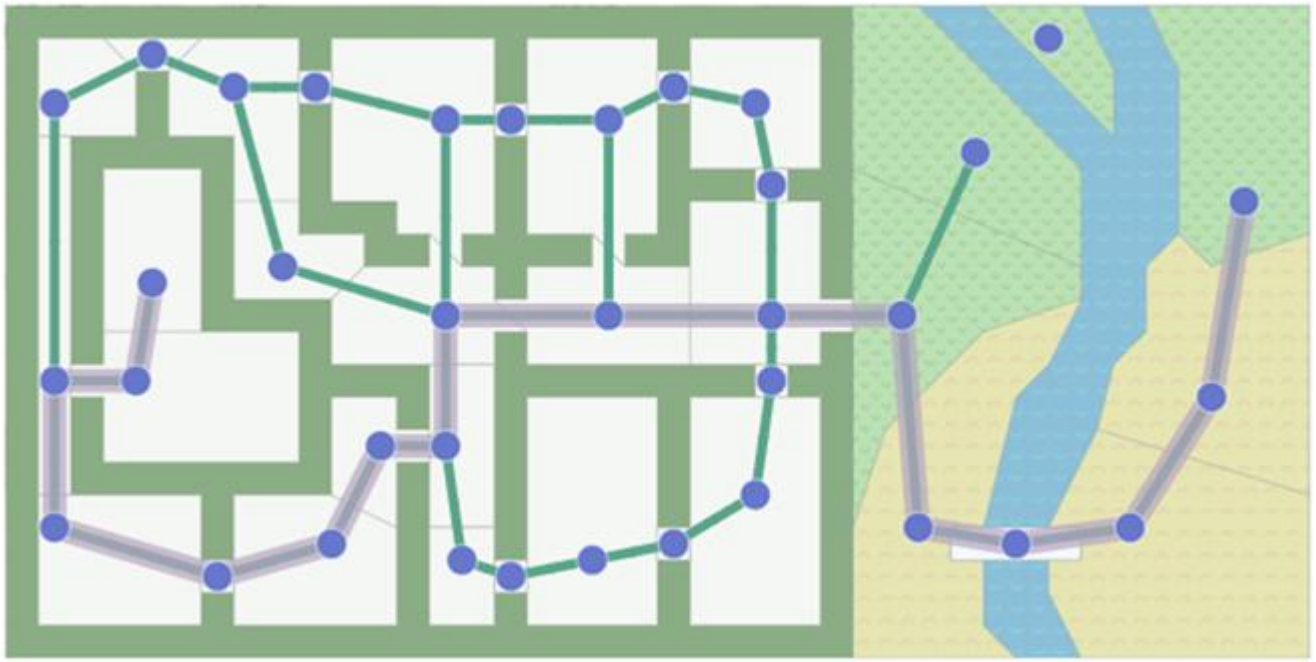
Pathfinding algorithms

- Graph search algorithms
  - Map is represented as a graph
  - Input: graph with nodes and edges



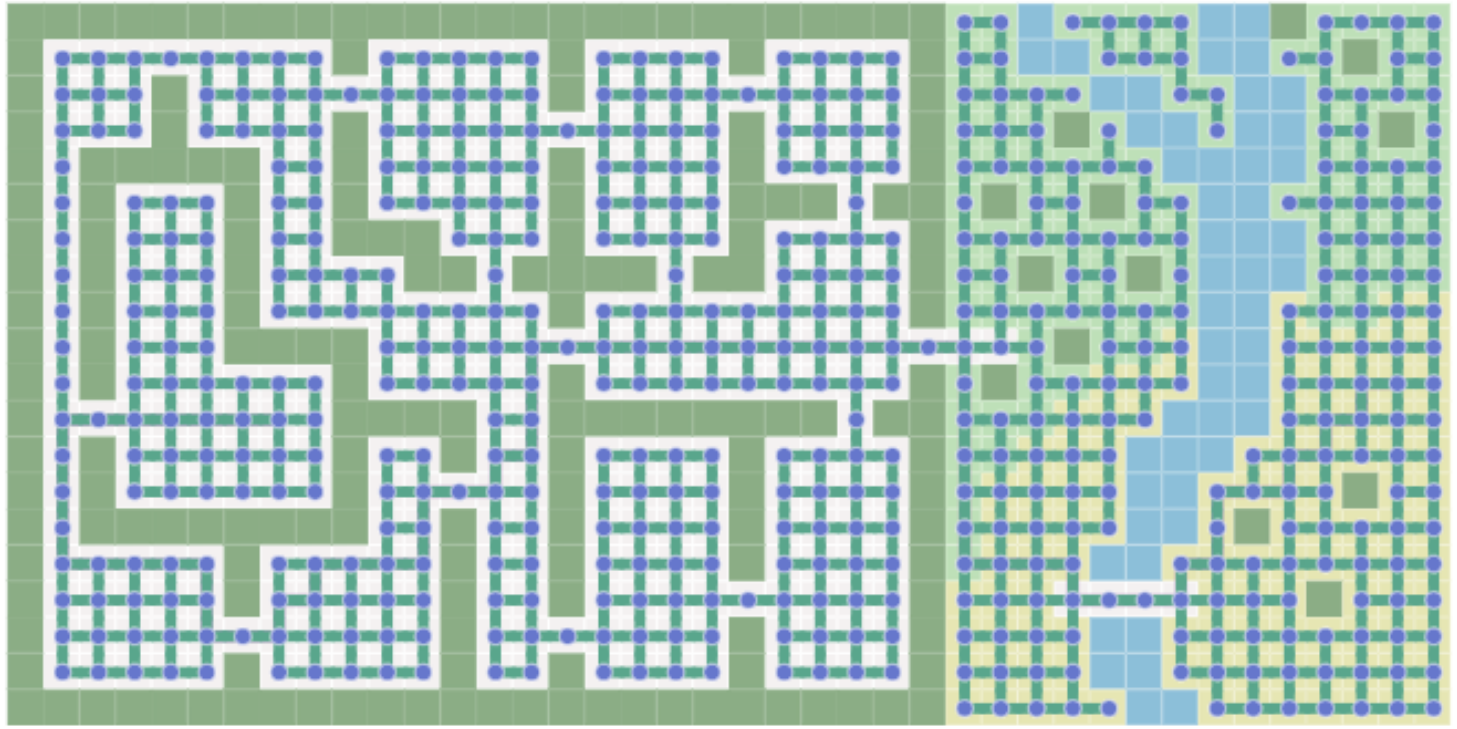
Pathfinding algorithms

- Graph search algorithms
  - Map is represented as a graph
  - Input: graph with nodes and edges
  - Output: nodes and edges



Alternative to representing the map

- Grid (still with graph correspondent)



Which representation?

- Graph search algorithm can accept any kind of graph

- In this lecture we use:
  - *grid map* with tiles as nodes and edges between adjacent nodes for easy visualisations
  - *graph* (undirected, for simplicity) for complex algorithm details
- Map is static
  - no change
  - all the obstacles are known

Algorithms

- Breadth First Search

- Dijkstra’s Algorithm

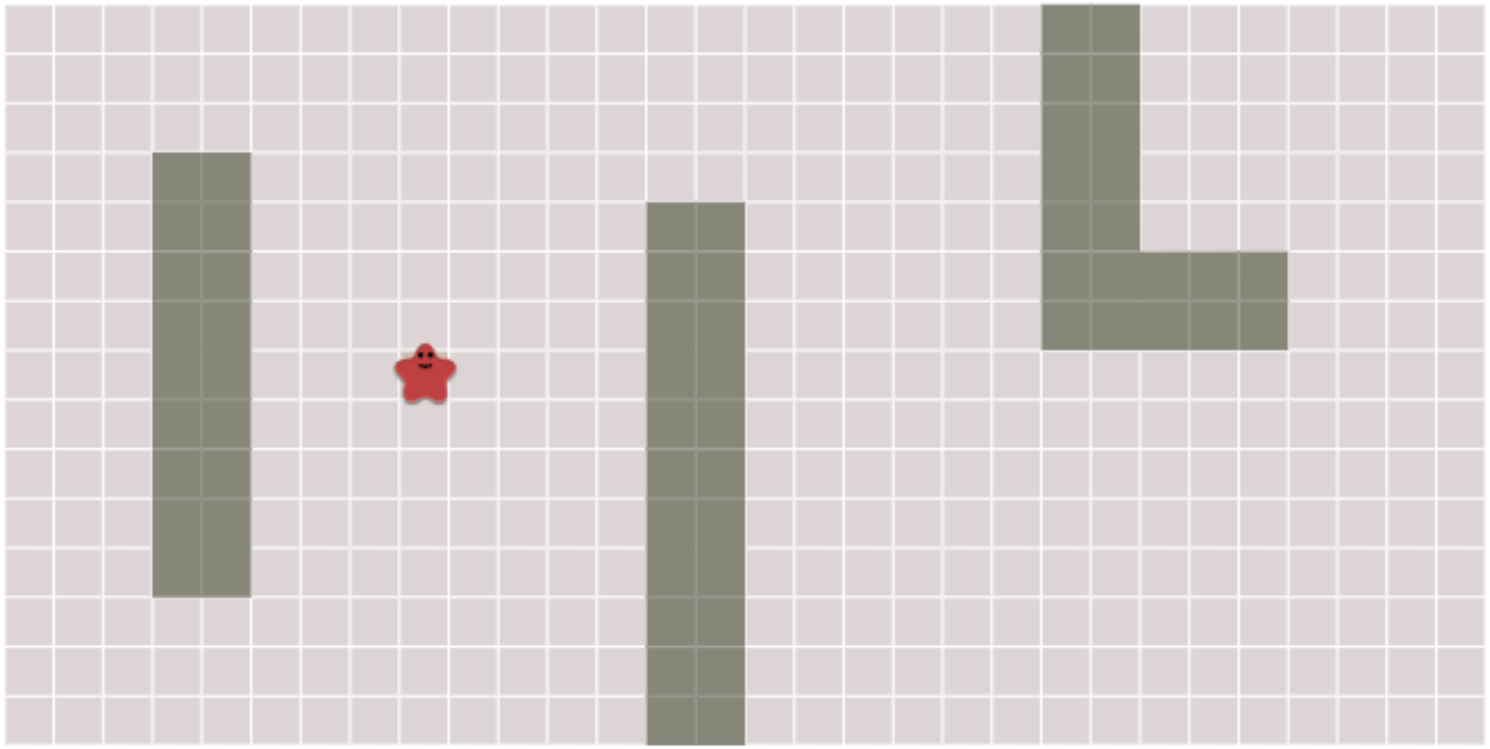
- A*

Breadth First Search: goal

- Find the shortest path from a source node to all other reachable nodes in terms of the number of edges

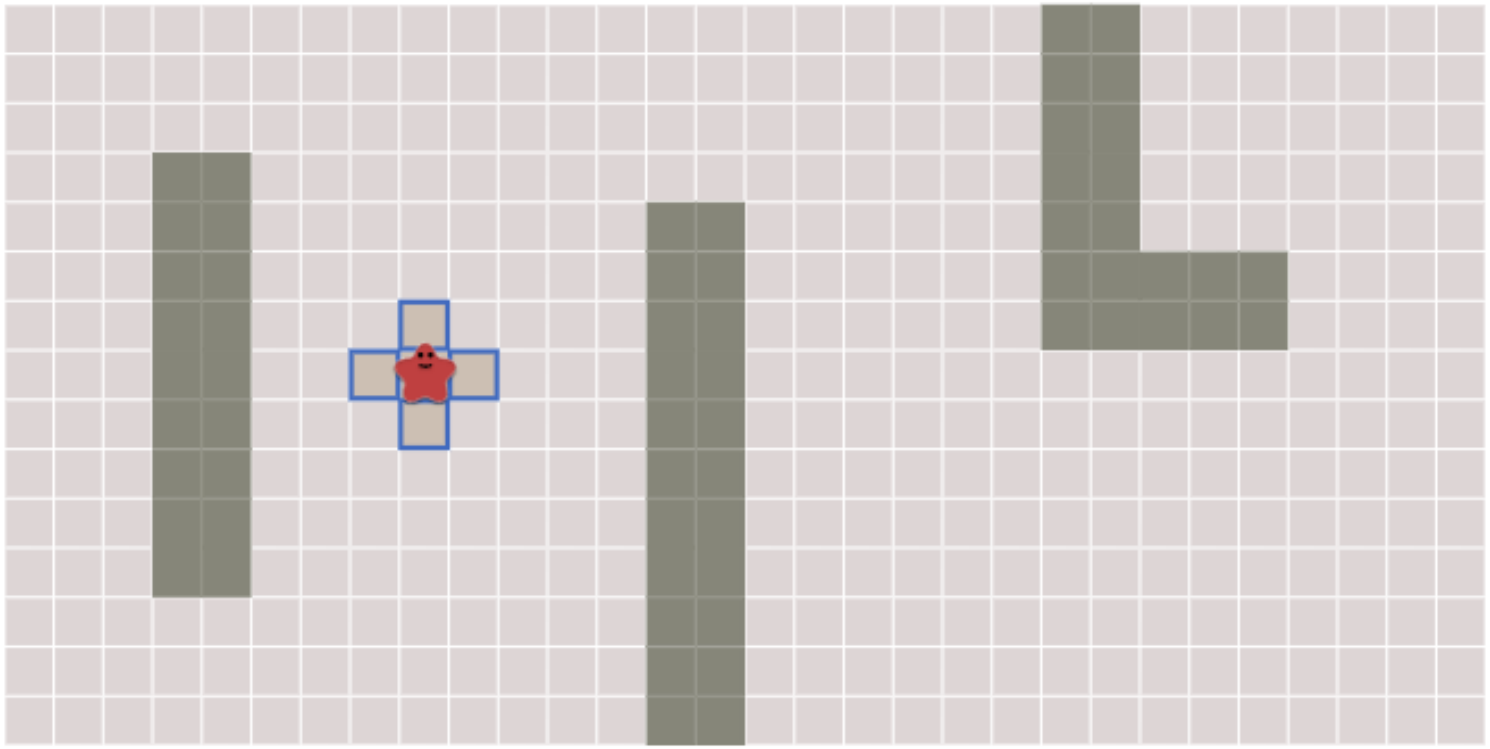
Breadth First Search: how it works

- Keep tracking of an expanding ring:



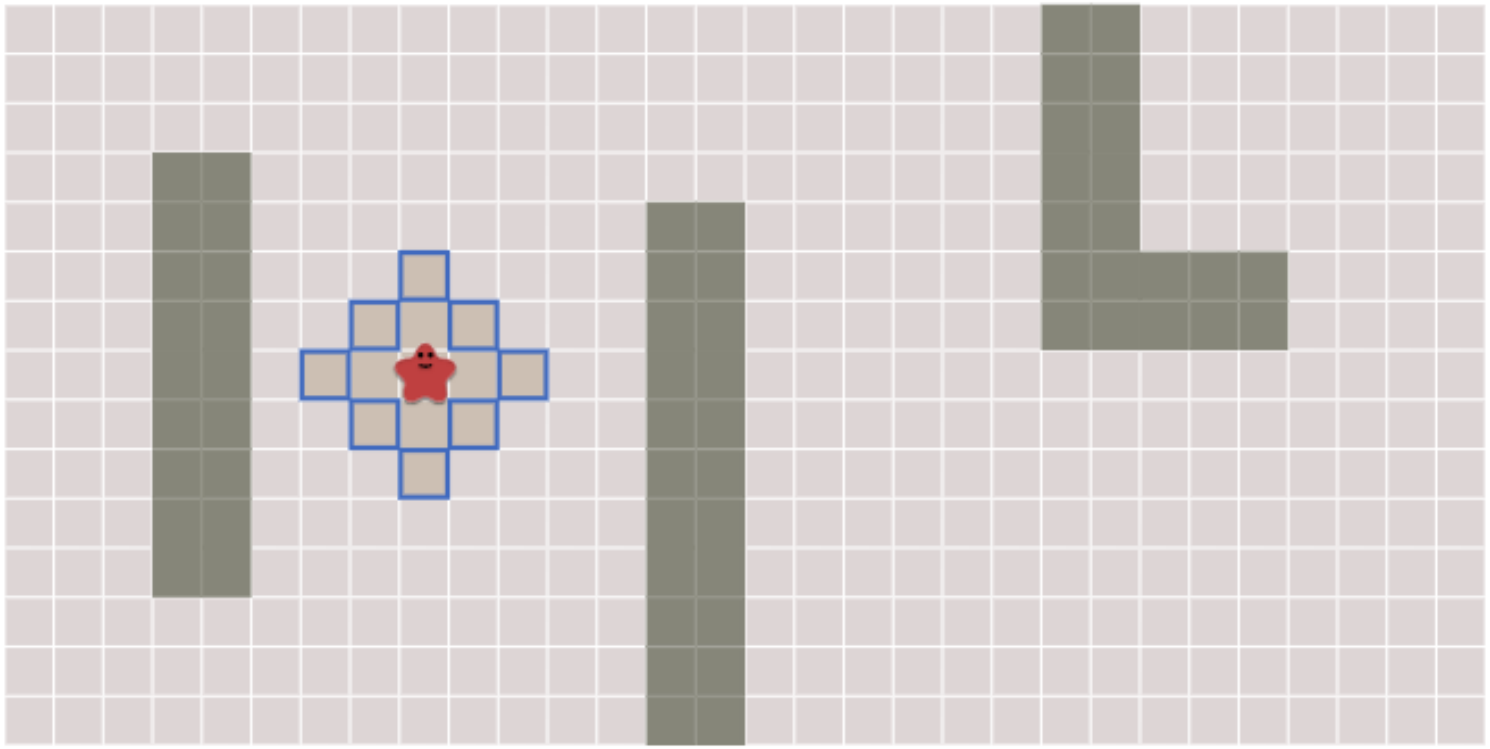
Breadth First Search: how it works

- Keep tracking of an expanding ring:



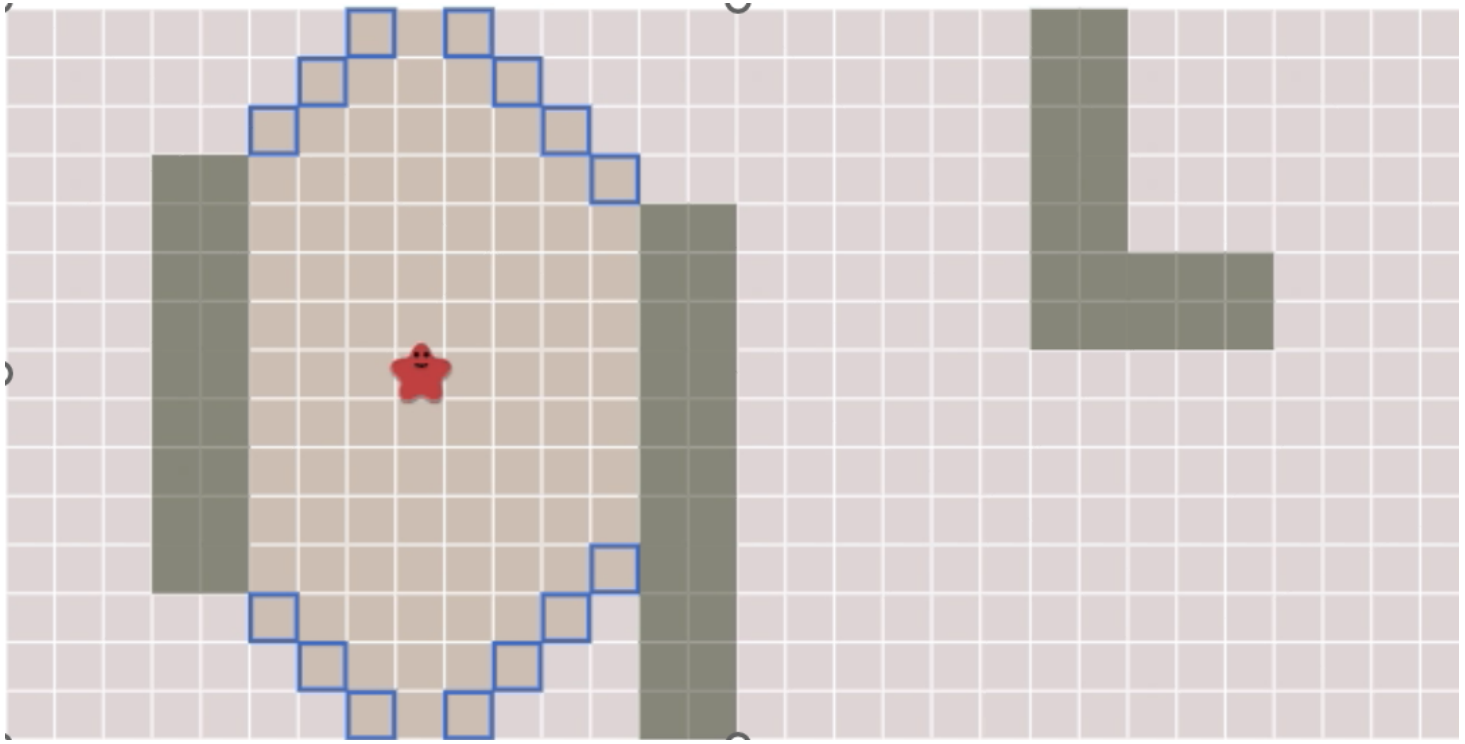
Breadth First Search: how it works

- Keep tracking of an expanding ring:



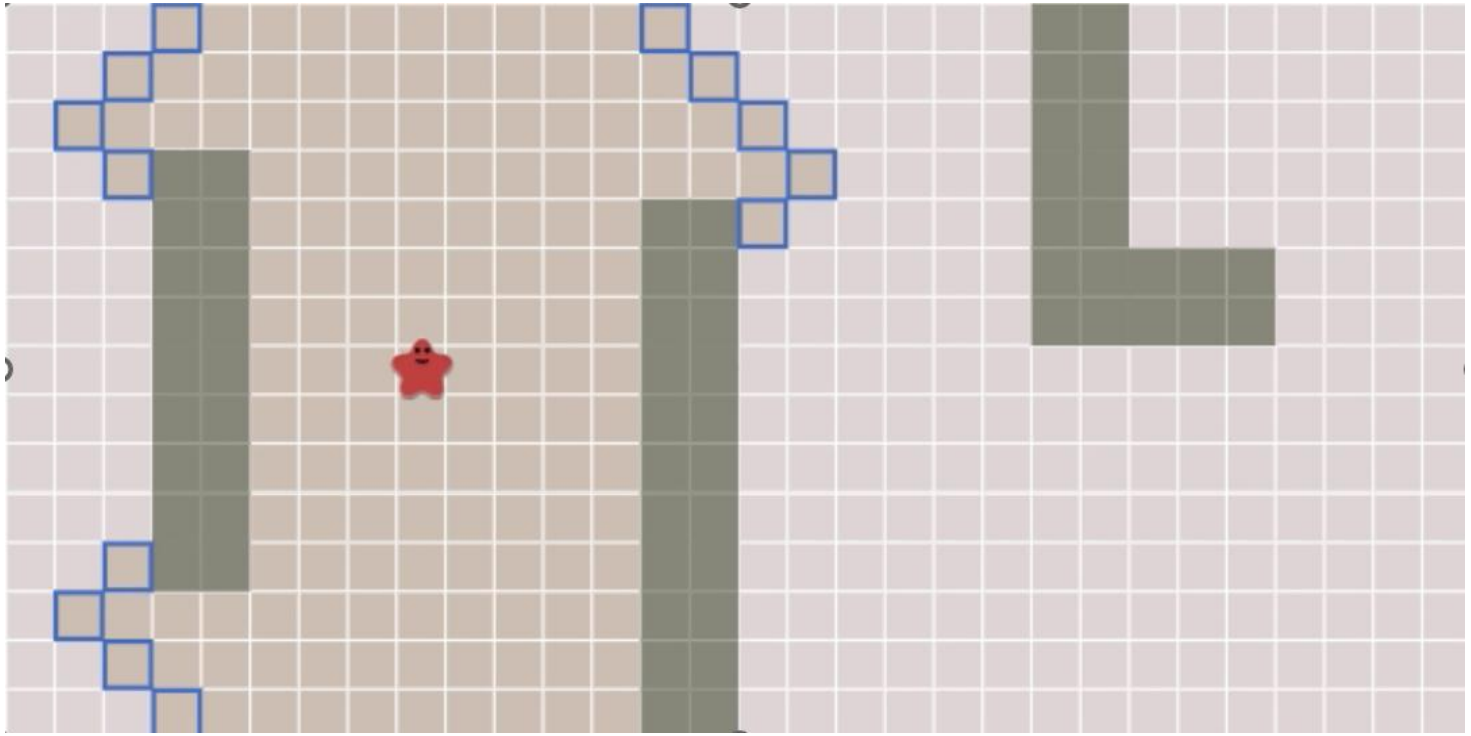
Breadth First Search: how it works

- Keep tracking of an expanding ring:



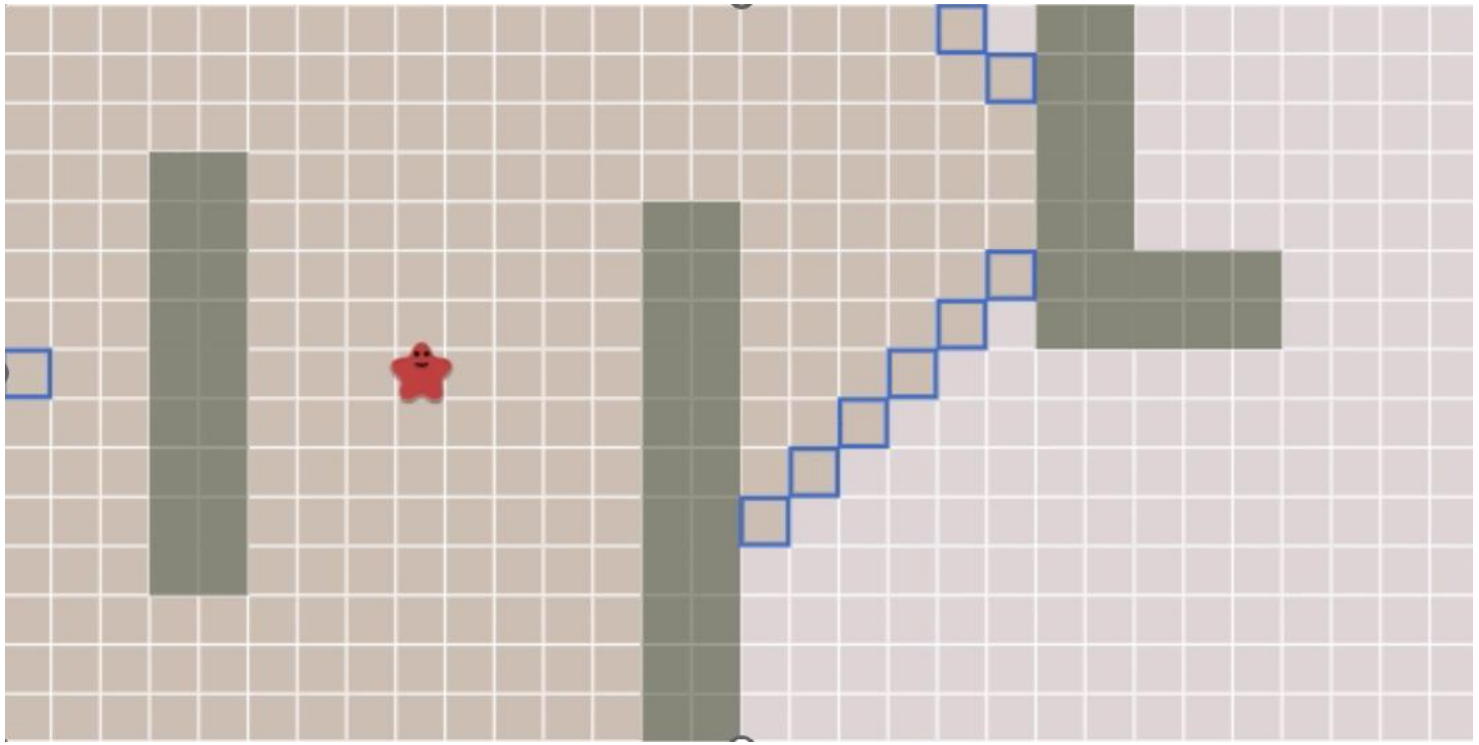
Breadth First Search: how it works

- Keep tracking of an expanding ring:



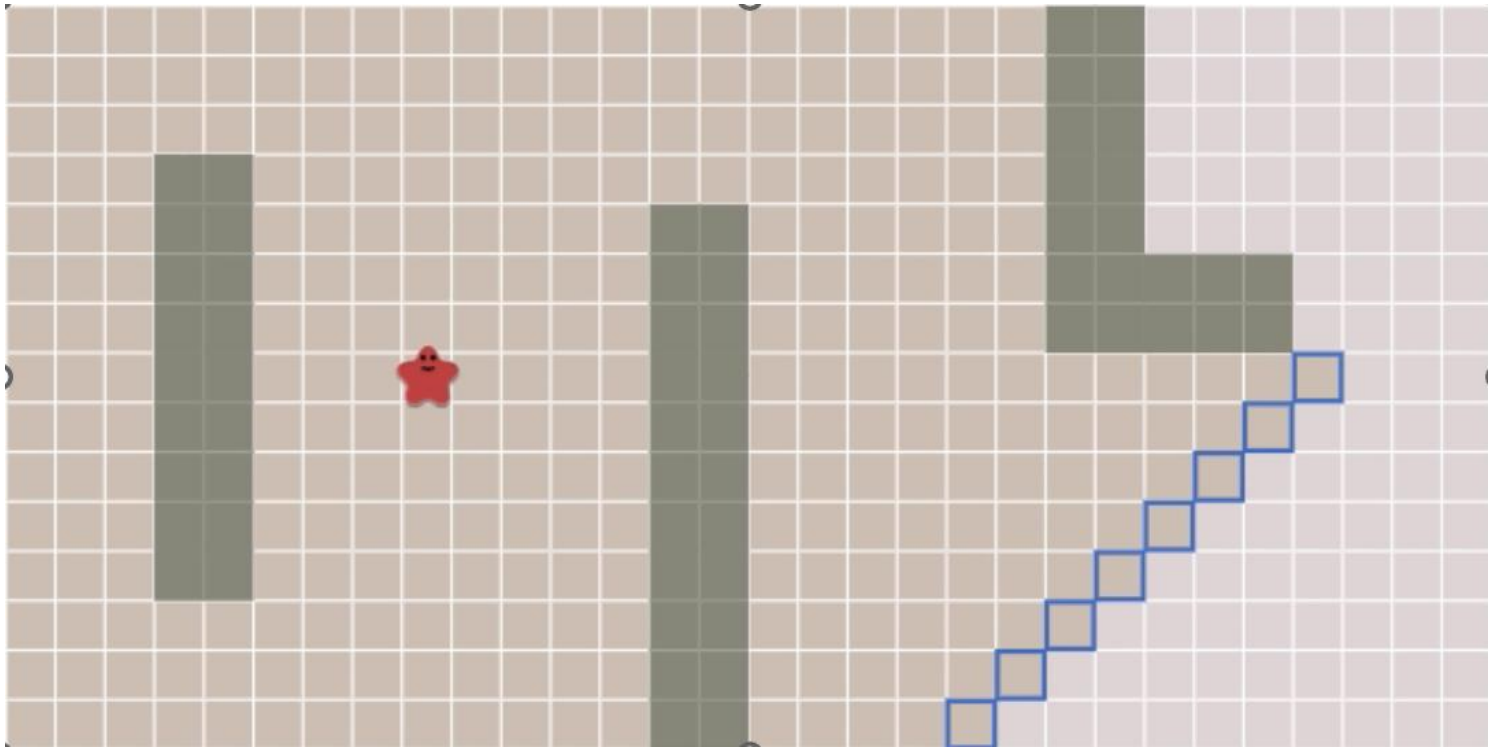
Breadth First Search: how it works

- Keep tracking of an expanding ring:



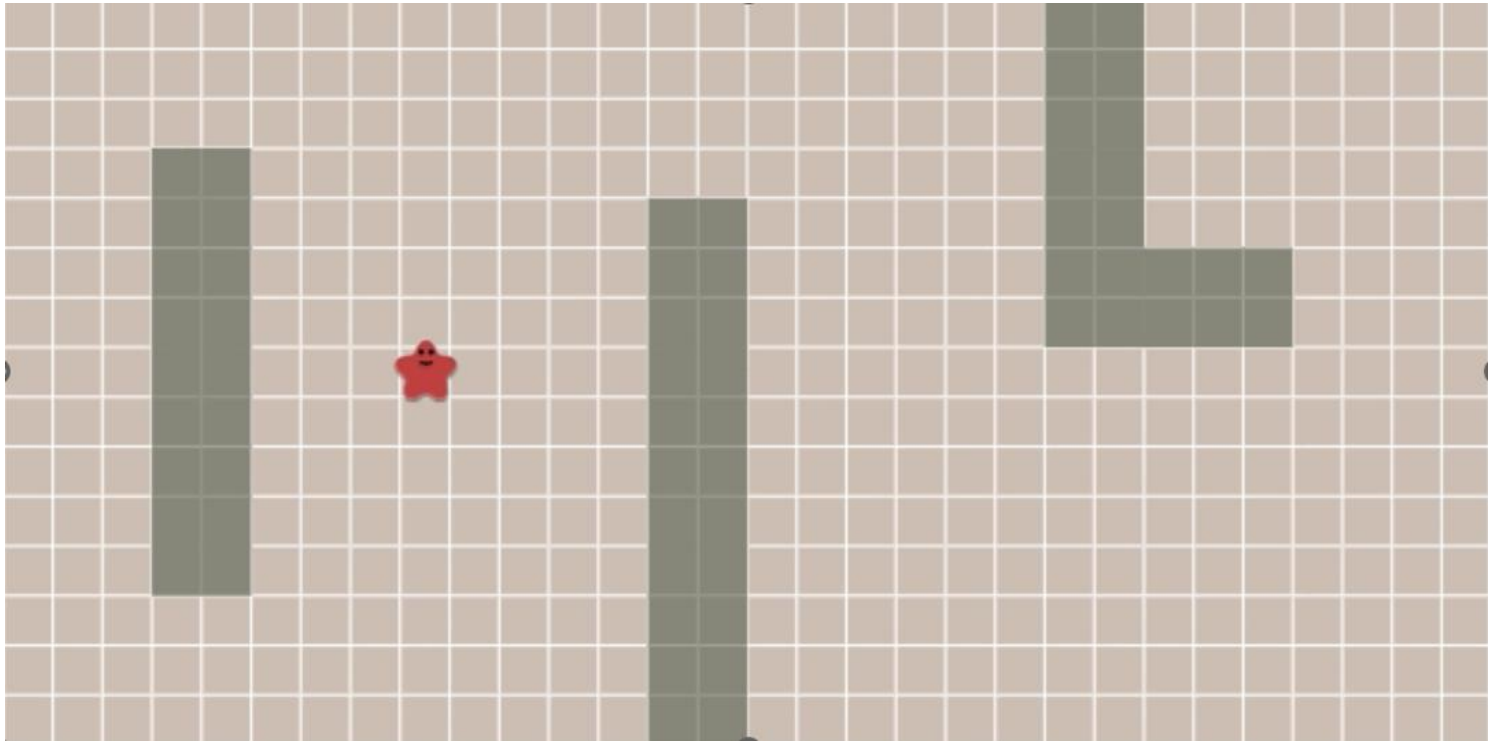
Breadth First Search: how it works

- Keep tracking of an expanding ring:

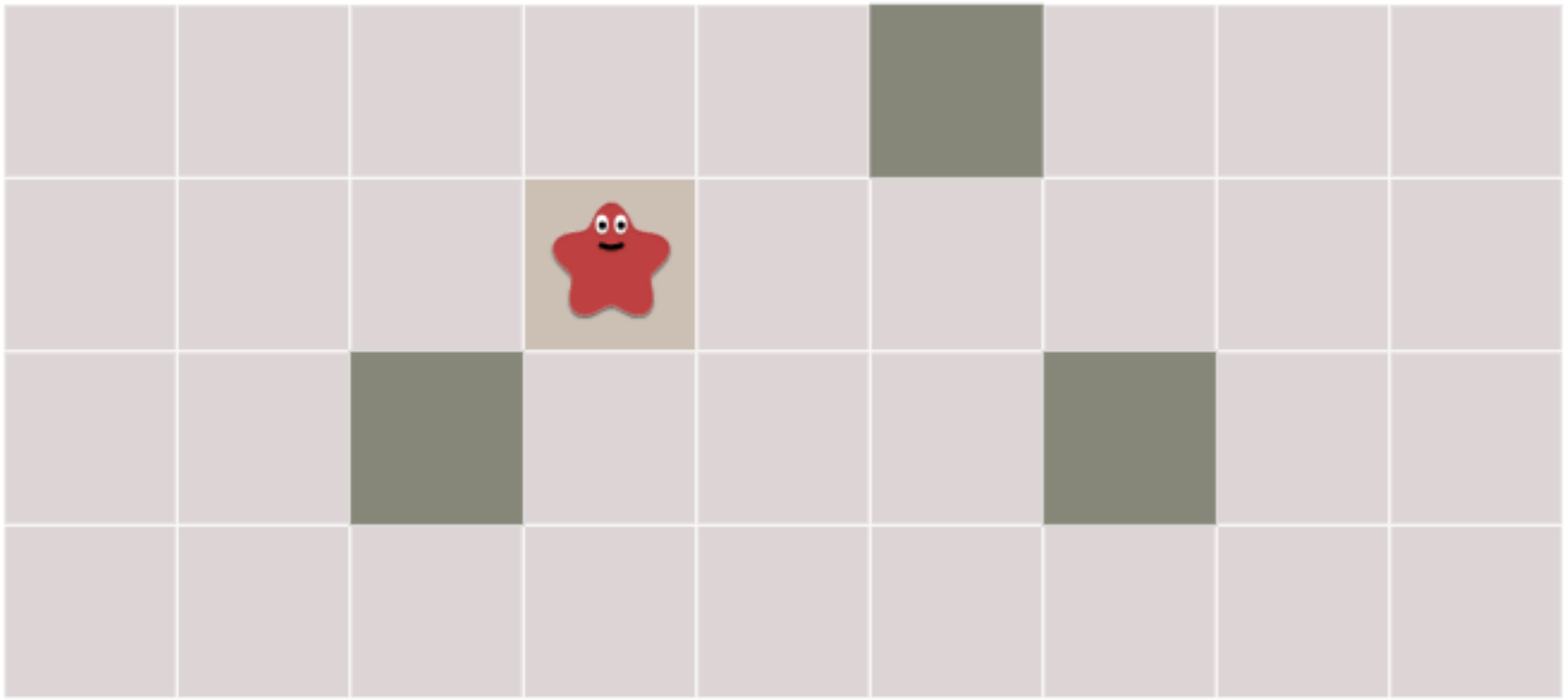


Breadth First Search: how it works

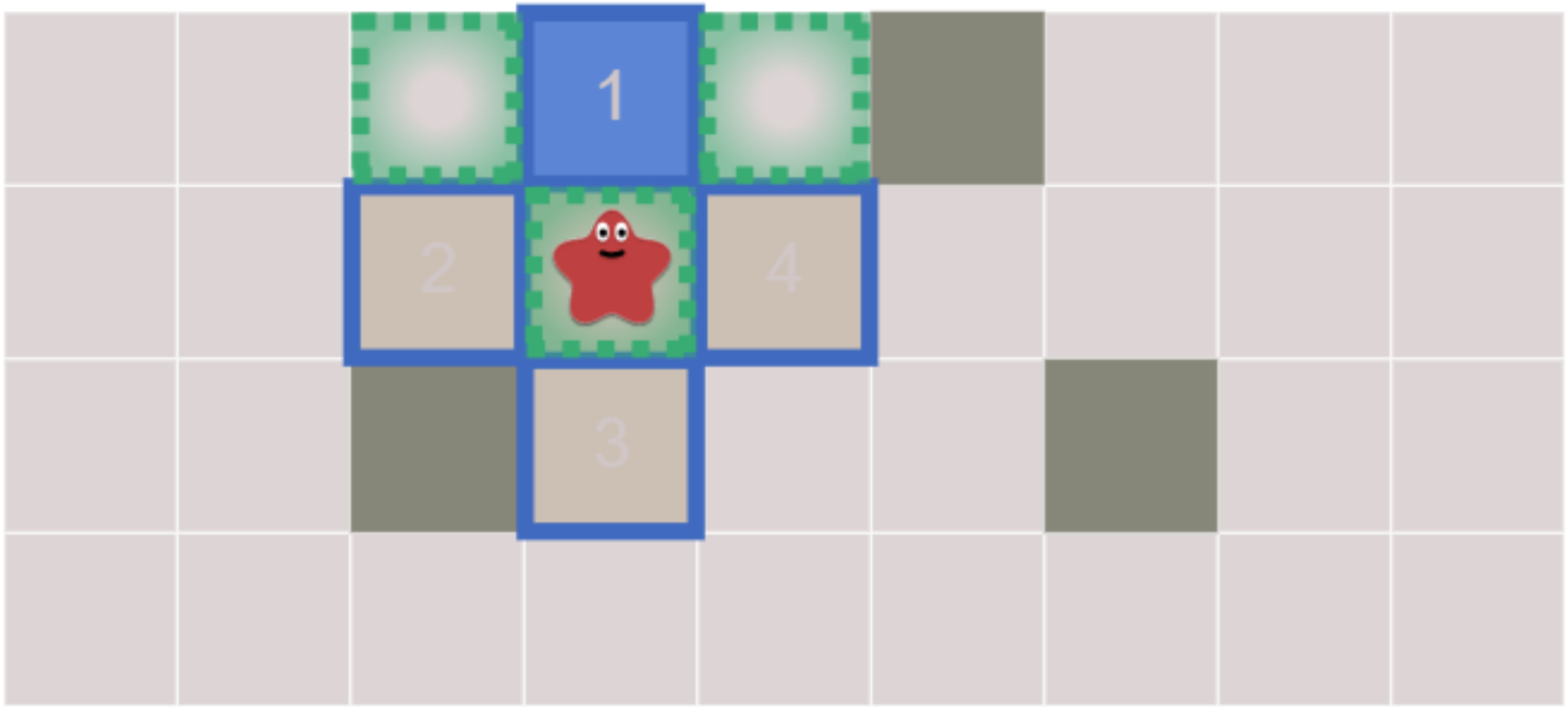
- Keep tracking of an expanding ring:



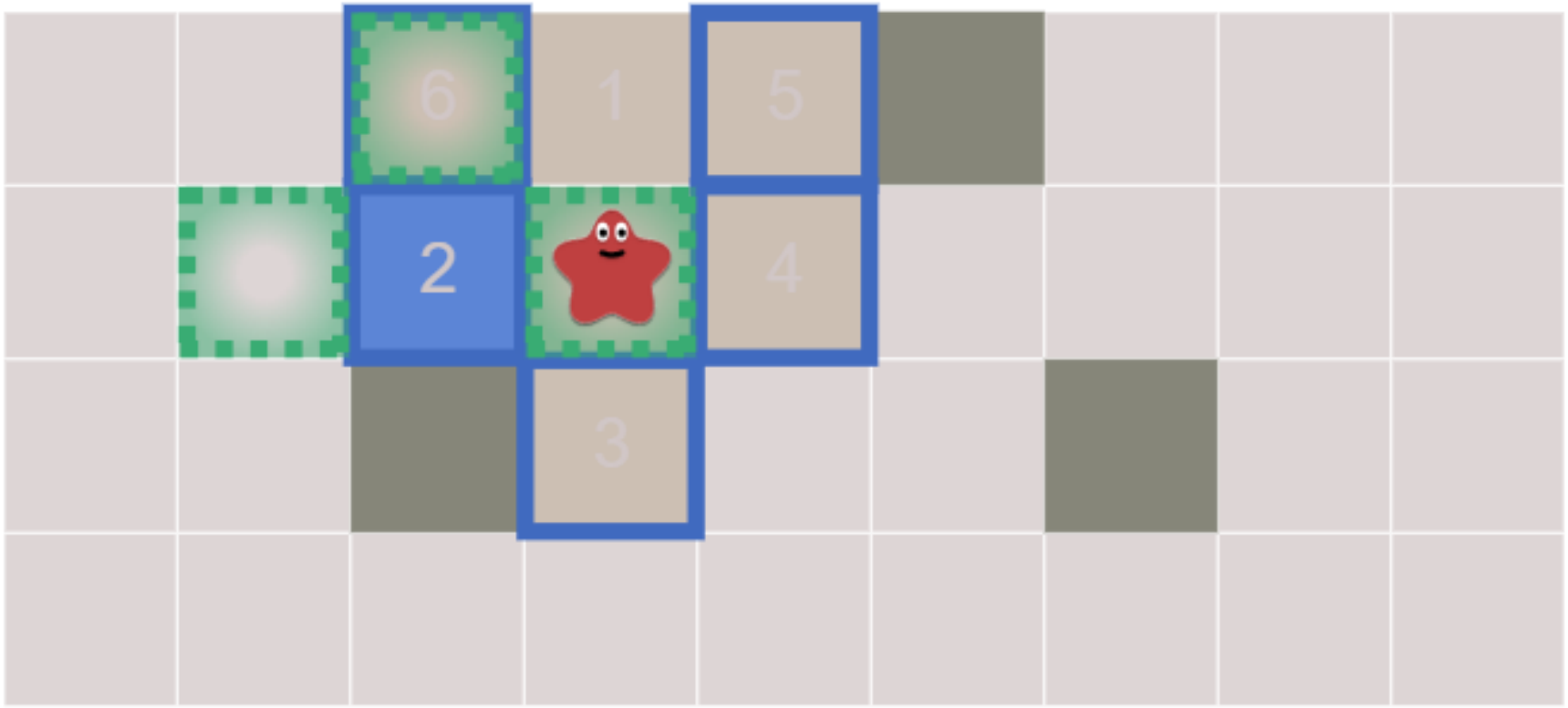
Breadth First Search: how it works



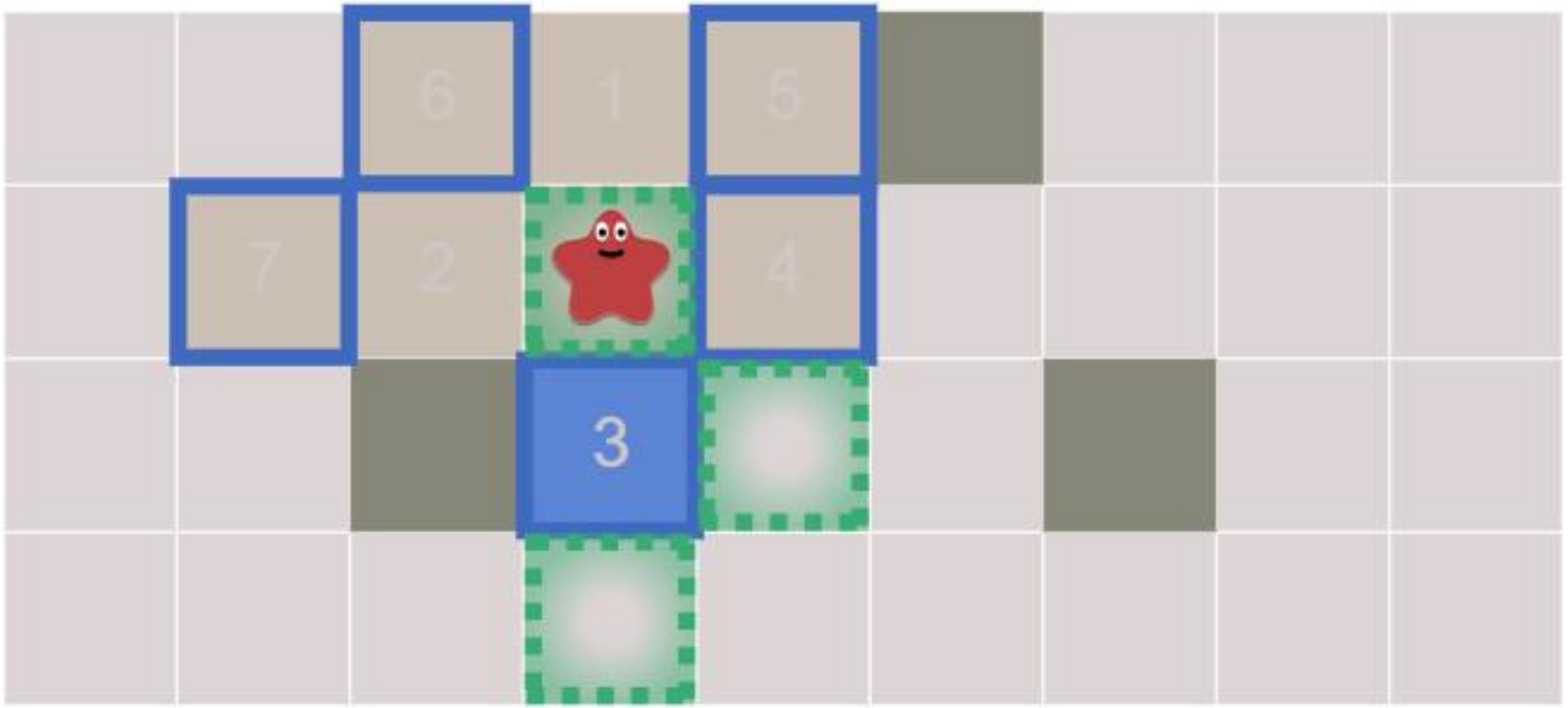
Breadth First Search: how it works



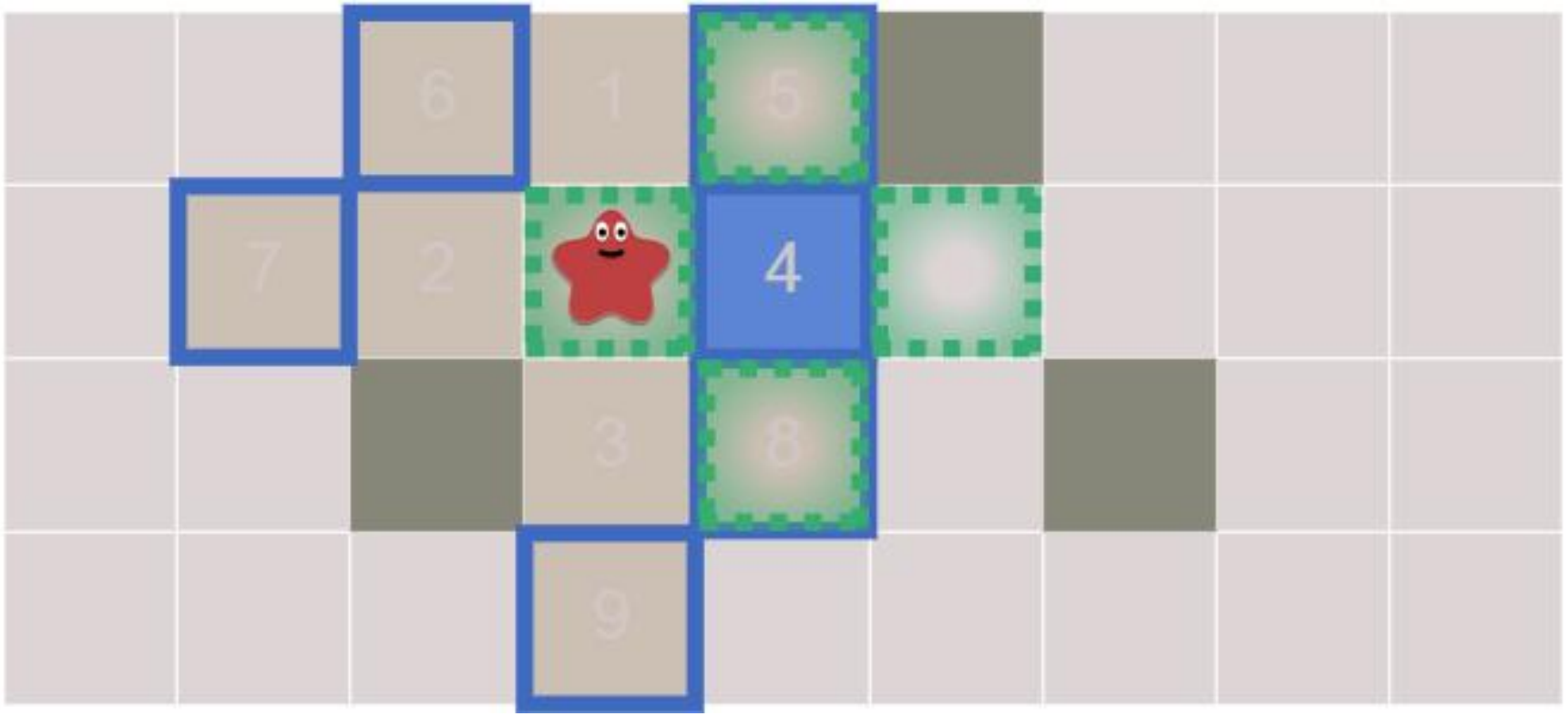
Breadth First Search: how it works



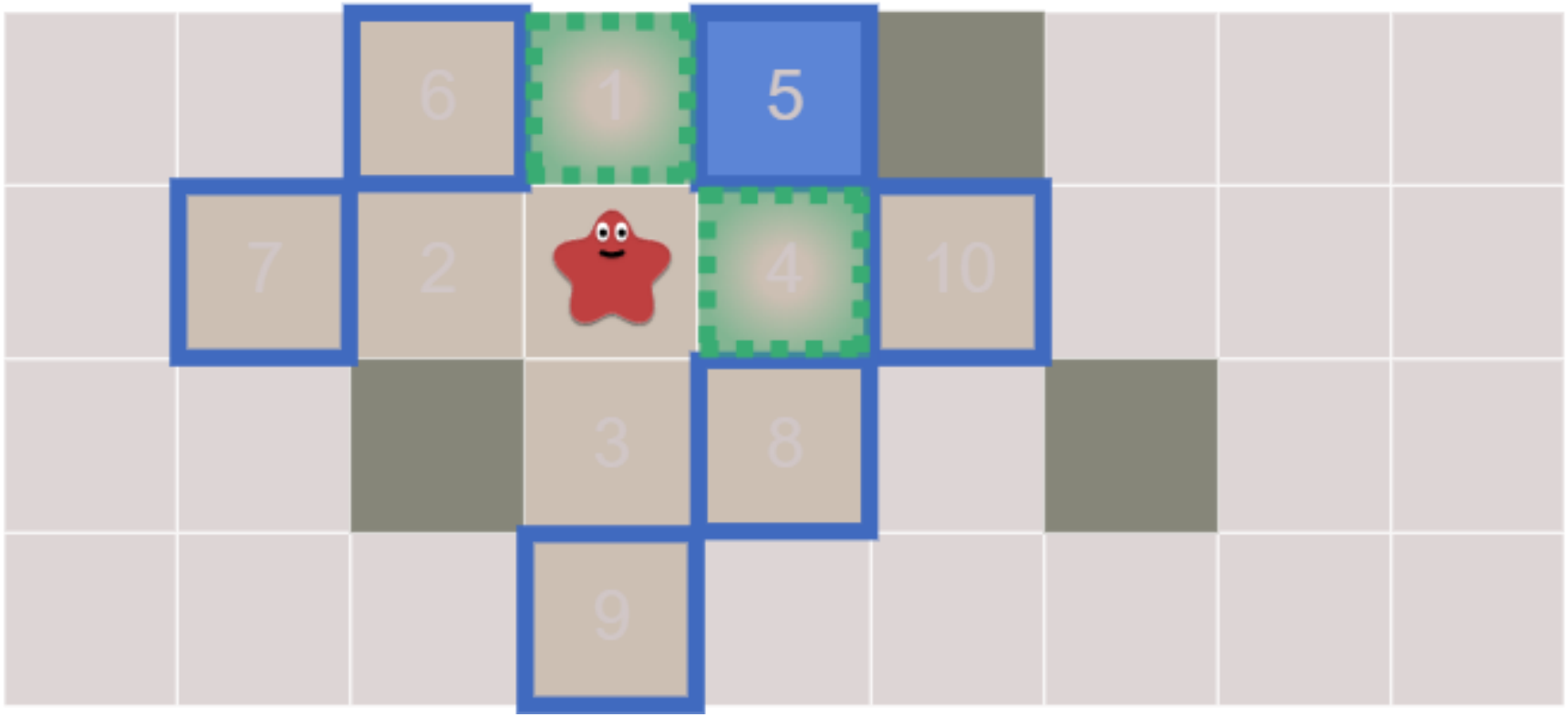
Breadth First Search: how it works



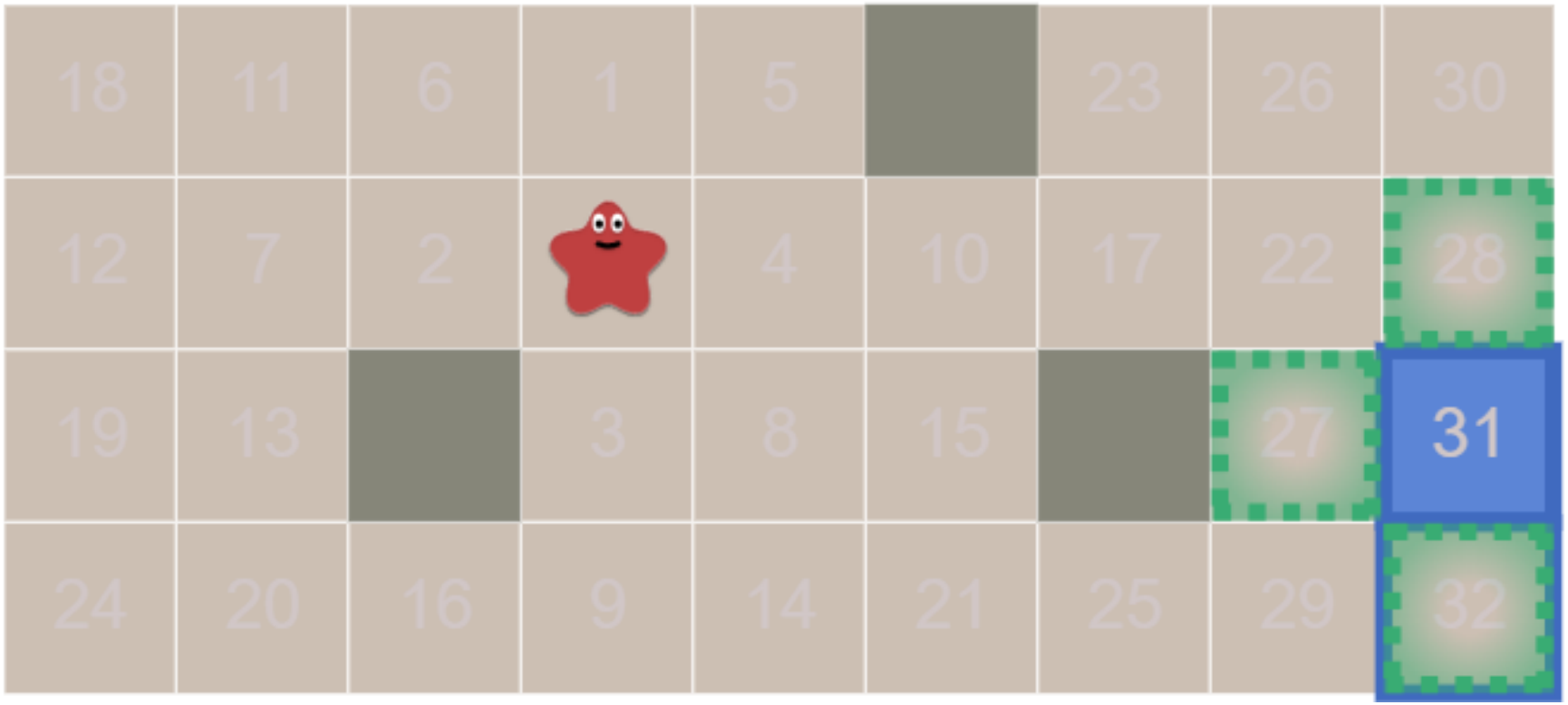
Breadth First Search: how it works



Breadth First Search: how it works



Breadth First Search: how it works



Breadth First Search: pseudocode

algorithm BFS **is**

Input: A graph *G* and a starting node *source* of *G*

Output: *parent* which traces the shortest path from each node back to *source*

let *Q* be a queue //first in first out
label *source* as explored
Q.enqueue(*source*)
while *Q* is not empty **do**
v := *Q*.dequeue()
for all edges from *v* to *w* **in** *G*.adjacentEdges(*v*) **do**
if *w* is not labelled as explored **then**
label *w* as explored
w.parent := *v*
Q.enqueue(*w*)

Getting path to goal

algorithm calculatePath **is**

Input: *parent* and *goal*

Output: *S* as a sequence of nodes between *goal* and *source*

let *S* be a sequence

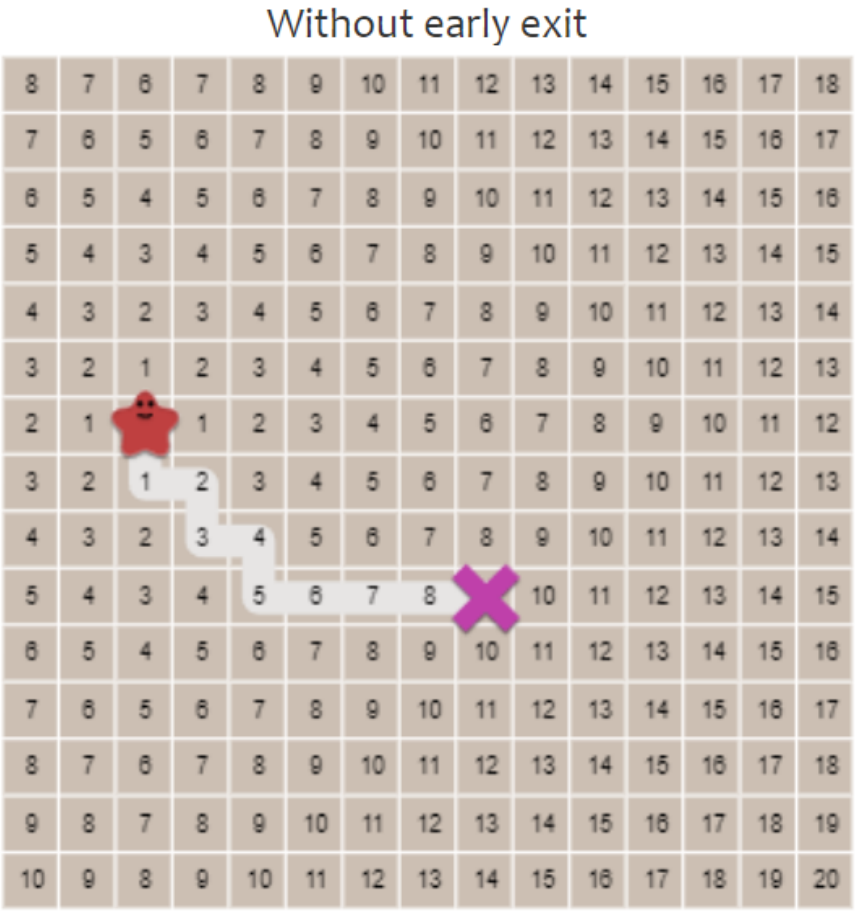
u := *goal*

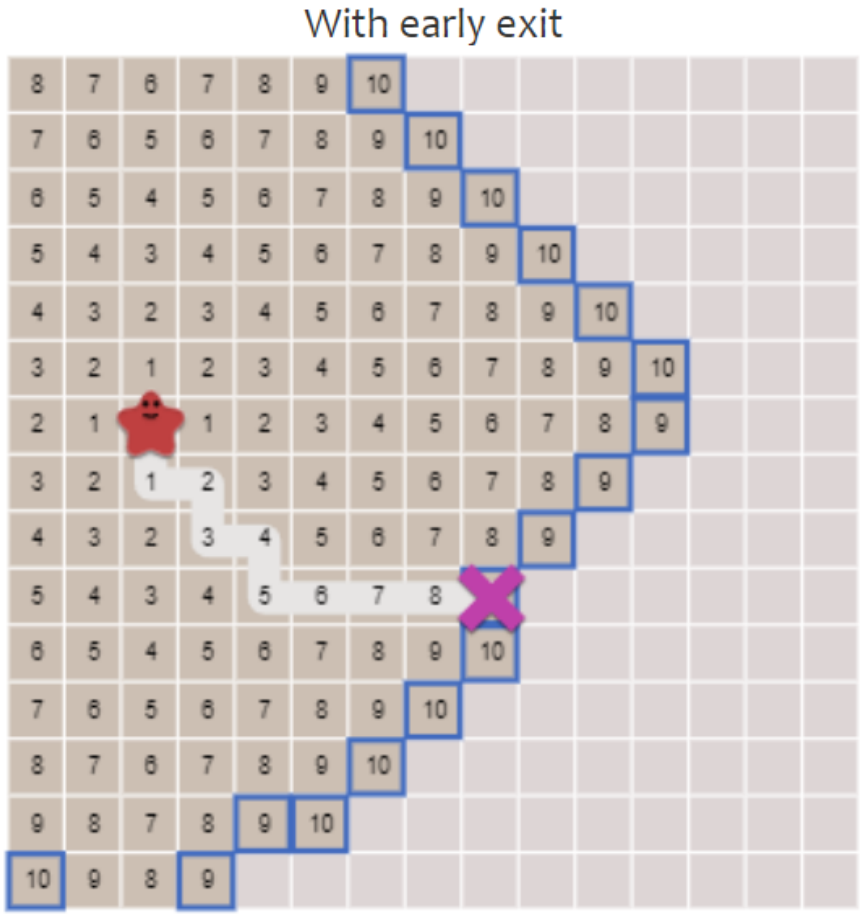
while *parent[u]* is defined **do**

*S.*push(*u*)

u := *parent[u]*

Breadth First Search: early exit





Breadth First Search: pseudocode

algorithm BFS **is**

Input: A graph *G,* a starting node *source* of *G* and a goal node *goal*

Output: *parent* which traces the shortest path from each node back to *source*

```
let Q be a queue
label source as explored
Q.enqueue(source)
while Q is not empty do
        v := Q.dequeue()
        if v is goal then
                break
        for all edges from v to w in G.adjacentEdges(v) do
                if w is not labelled as explored then
                        label w as explored
                        w.parent := v
                        Q.enqueue(w)
```

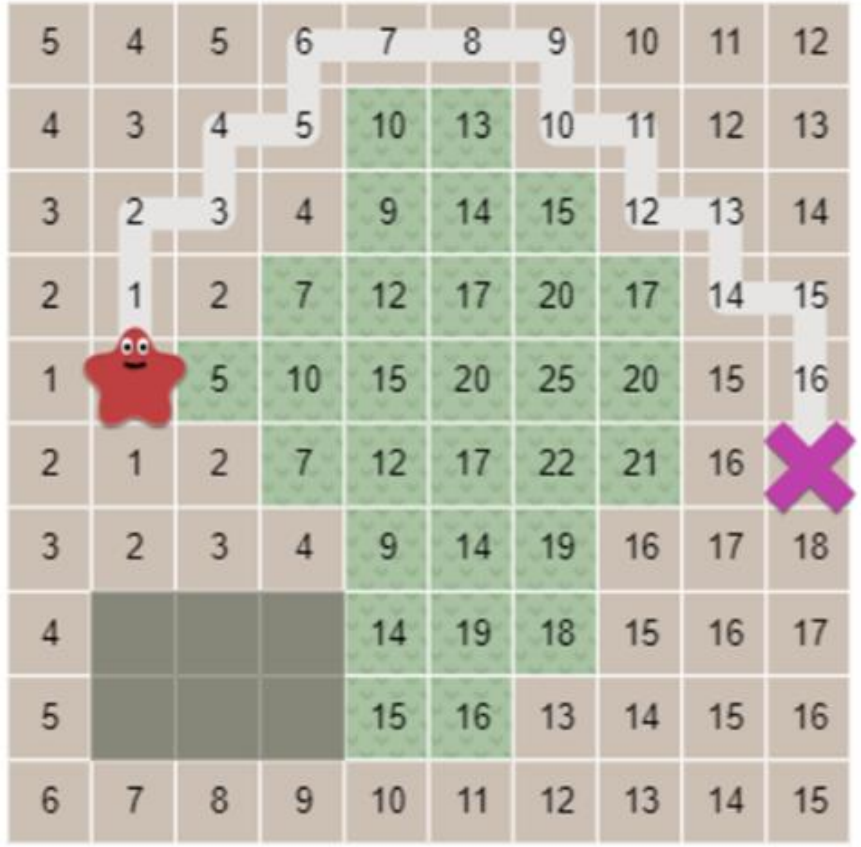
Dijkstra's algorithm: goal

- Find the shortest path from a source node to all other reachable nodes, ***while also considering movement cost*** (difference to BFS)
  - *Movement cost can be distance, travel time, resource consumption (e.g. fuel, money, etc.)*
- Dijkstra cannot work with negative movement cost
- It ***prioritises smaller movement cost***

Dijkstra's algorithm: goal

Movement costs 1

Movement costs 5 on grass



Dijkstra's algorithm: how it works

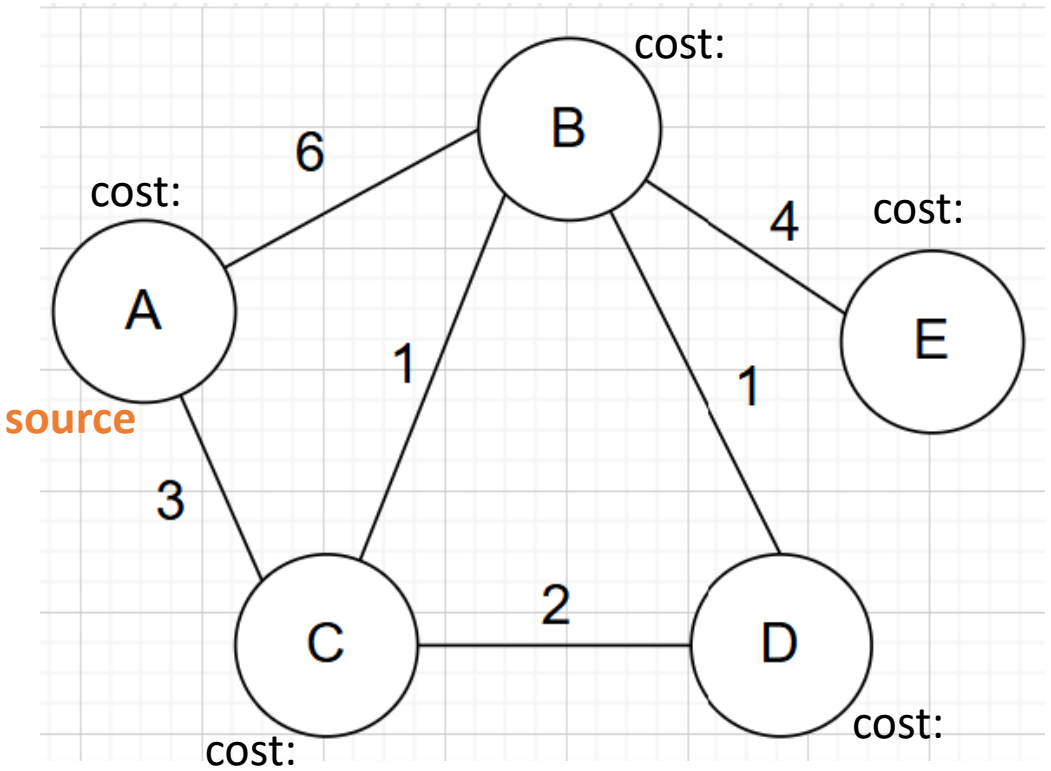
1. Mark the non-visited node with smallest cost (initially source) as visited
2. Find all its unvisited neighbors
3. Calculate the cost of reaching them
4. Sort based on cost
5. Repeat 1-4 until all nodes visited

Dijkstra's algorithm: practicalities

- Movement costs can be considered as (positive) weights on ***weighted graph***
- ***Min priority queue*** makes algorithm more efficient
  - Stores nodes-cost tuples with their cost priority
  - Node-cost tuple with min cost always head of queue
  - In Java, dedicated PriorityQueue type

  https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html

Dijkstra's algorithm: how it works

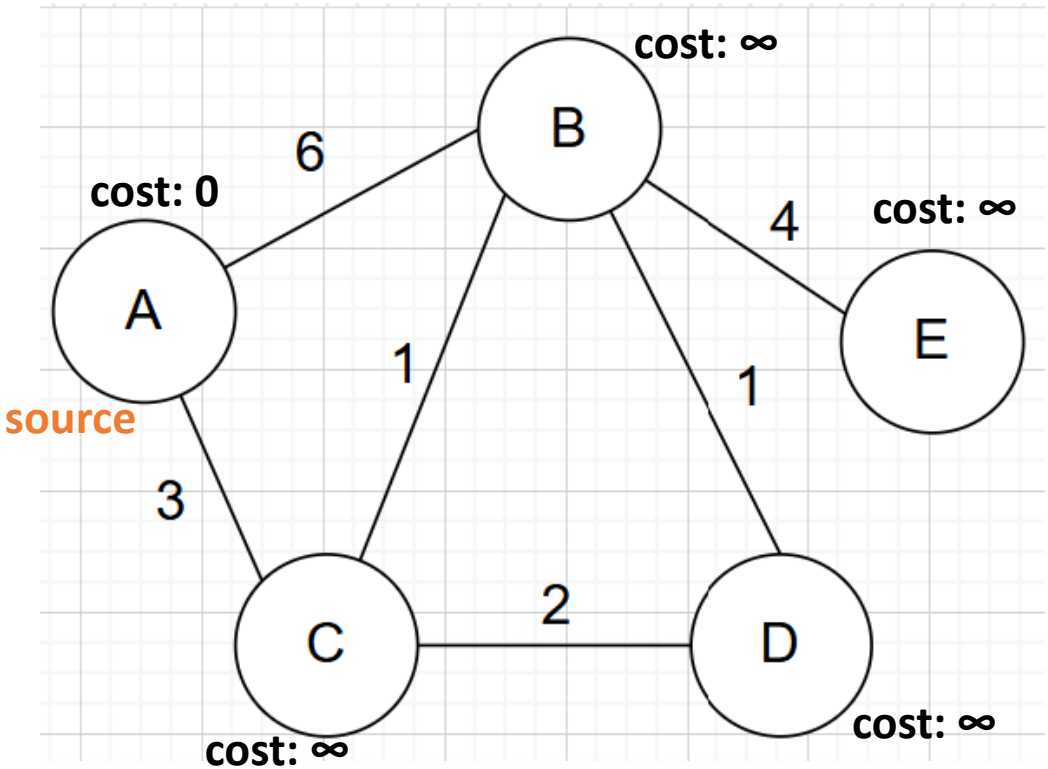


A B C D E

parent:

Q (min priority queue):

Dijkstra's algorithm: how it works

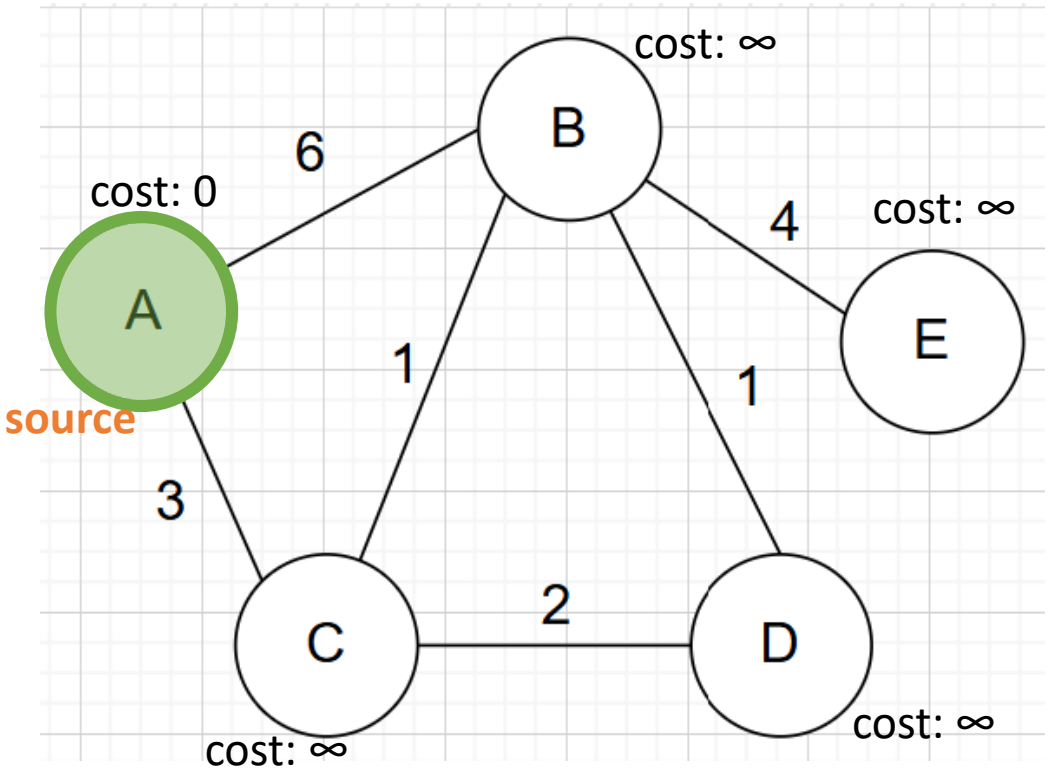


A B C D E

parent:

Q (min priority queue):
(A, 0)

Dijkstra's algorithm: how it works



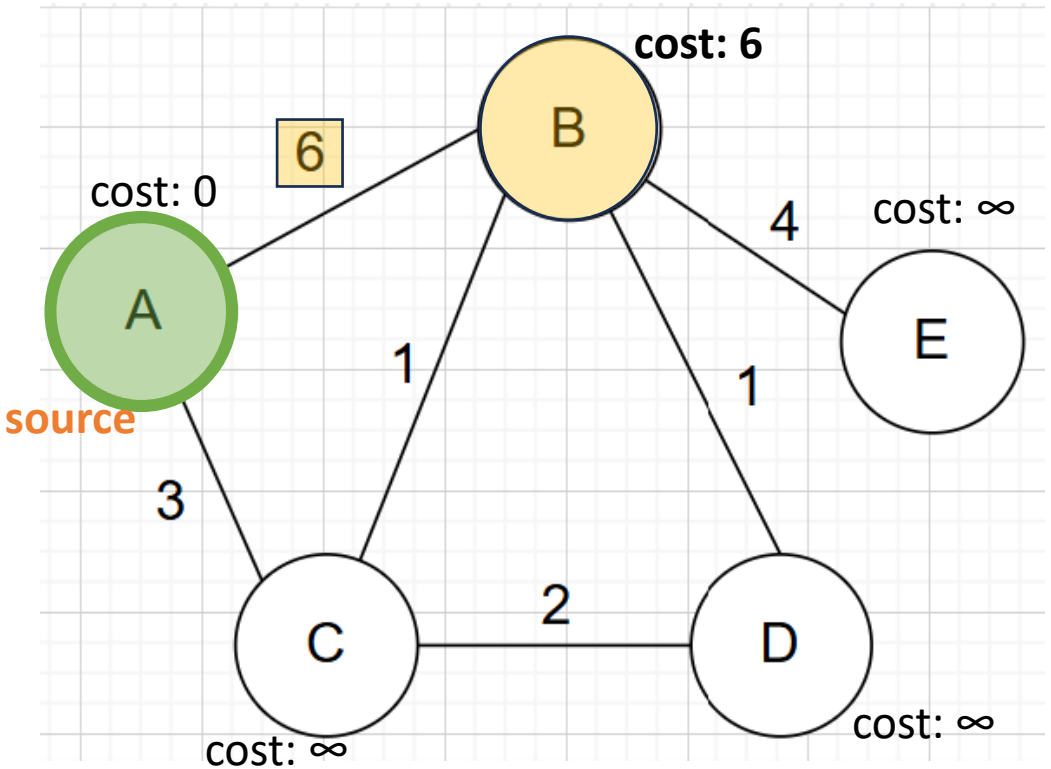
A B C D E

parent:

Q (min priority queue):

~~**(A, 0)**~~

Dijkstra's algorithm: how it works



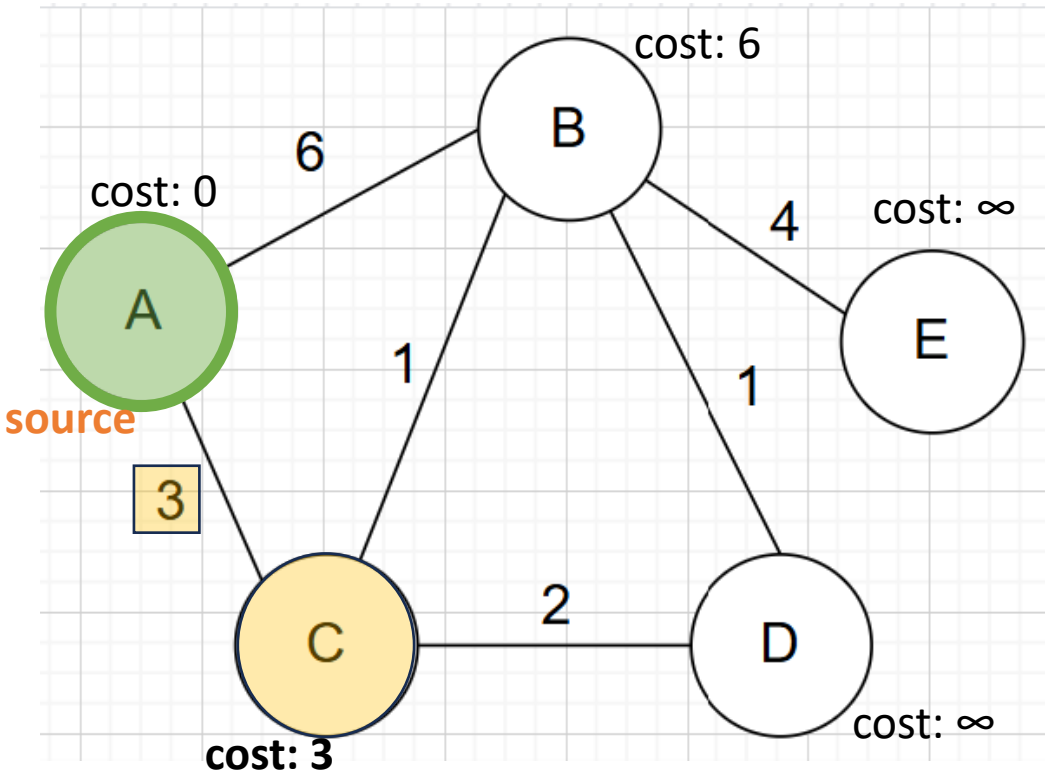
	A	B	C	D	E
parent:		**A**			

Q (min priority queue):

~~(A, 0)~~

(B, 6)

Dijkstra's algorithm: how it works



	A	B	C	D	E
parent:		A	**A**		

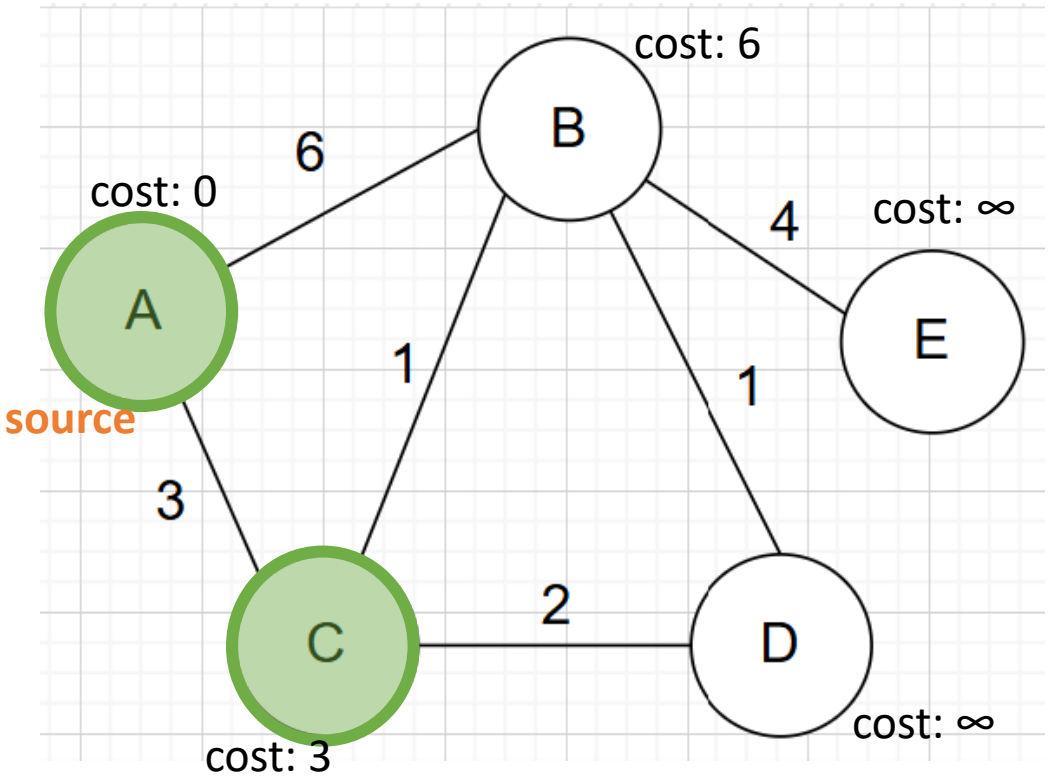
Q (min priority queue):

~~(A, 0)~~

(C, 3)

(B, 6)

Dijkstra's algorithm: how it works



	A	B	C	D	E
parent:		A	A		

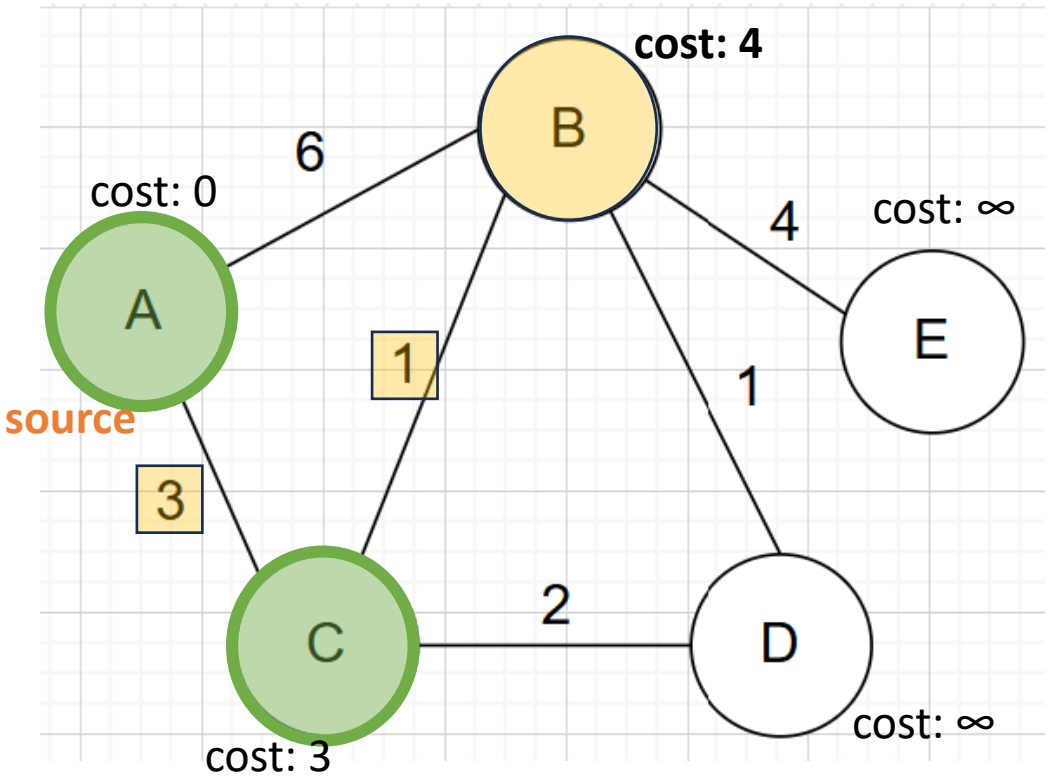
Q (min priority queue):

~~(A, 0)~~

~~**(C, 3)**~~

(B, 6)

Dijkstra's algorithm: how it works



	A	B	C	D	E
parent:		**C**	A		

Q (min priority queue):

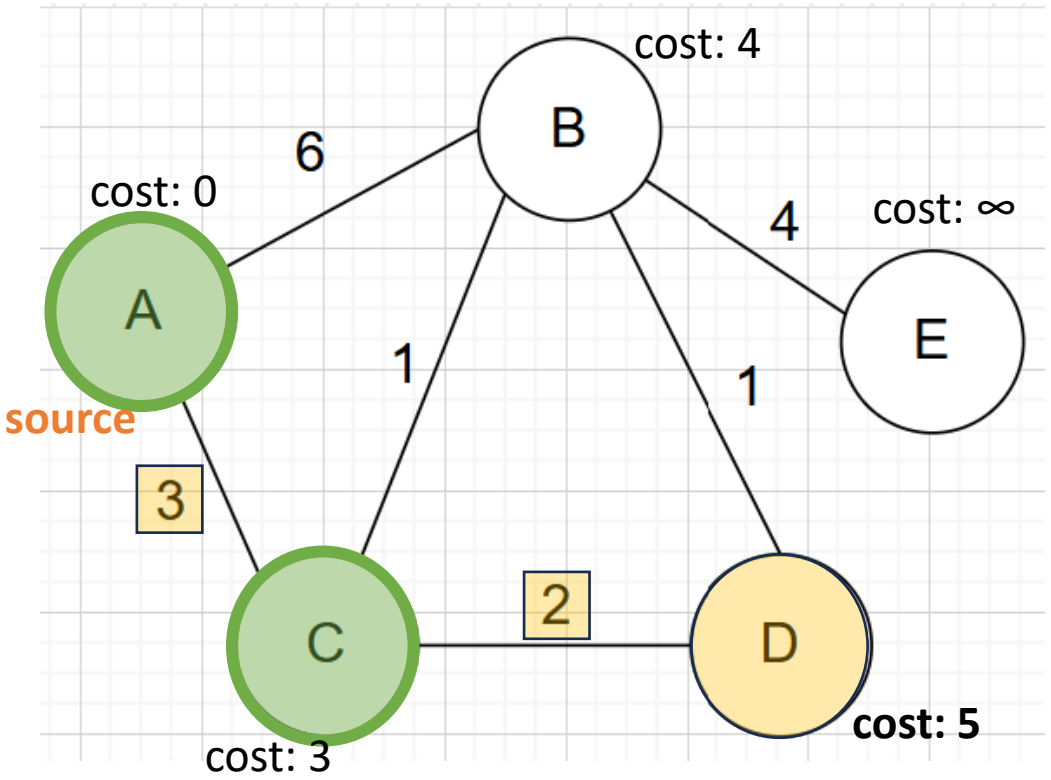
~~(A, 0)~~

~~(C, 3)~~

(B, 4)

(B, 6)

Dijkstra's algorithm: how it works



	A	B	C	D	E
parent:		C	A	**C**	

Q (min priority queue):

~~(A, 0)~~

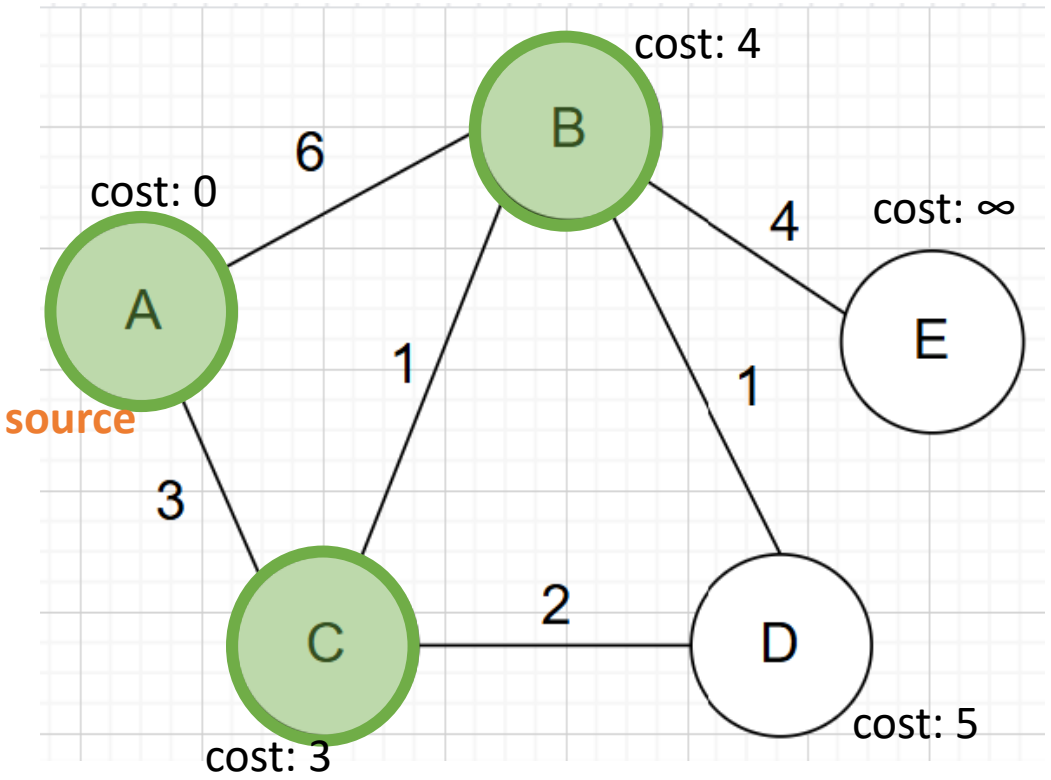
~~(C, 3)~~

(B, 4)

(D, 5)

(B, 6)

Dijkstra's algorithm: how it works



	A	B	C	D	E
parent:		C	A	C	

Q (min priority queue):

~~(A, 0)~~

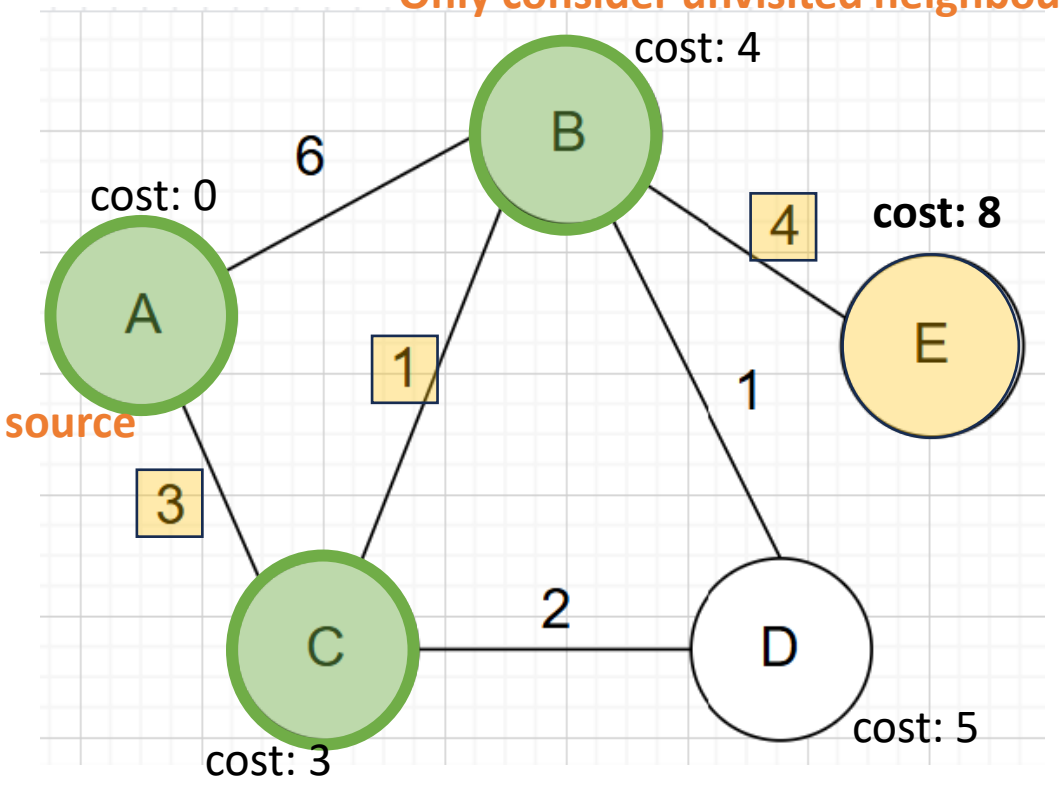
~~(C, 3)~~

~~**(B, 4)**~~

(D, 5)

(B, 6)

Dijkstra's algorithm: how it works

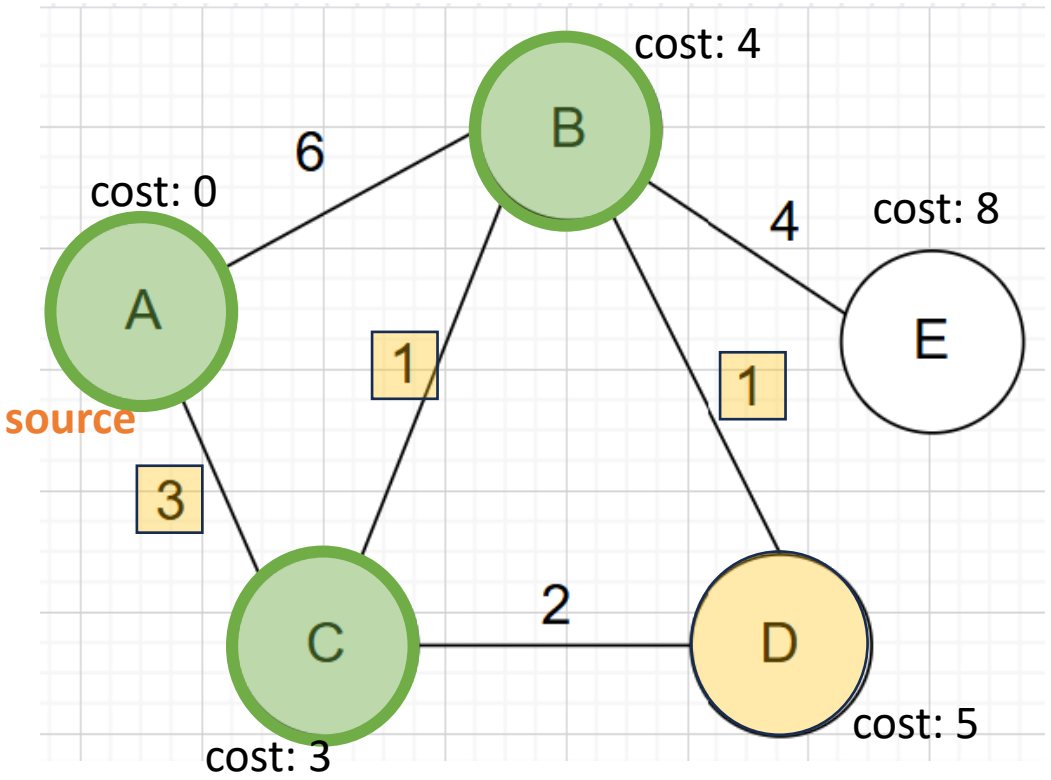


	A	B	C	D	E
parent:		C	A	C	**B**

Q (min priority queue):

~~(A, 0)~~
~~(C, 3)~~
~~(B, 4)~~
(D, 5)
(B, 6)
(E, 8)

Dijkstra's algorithm: how it works

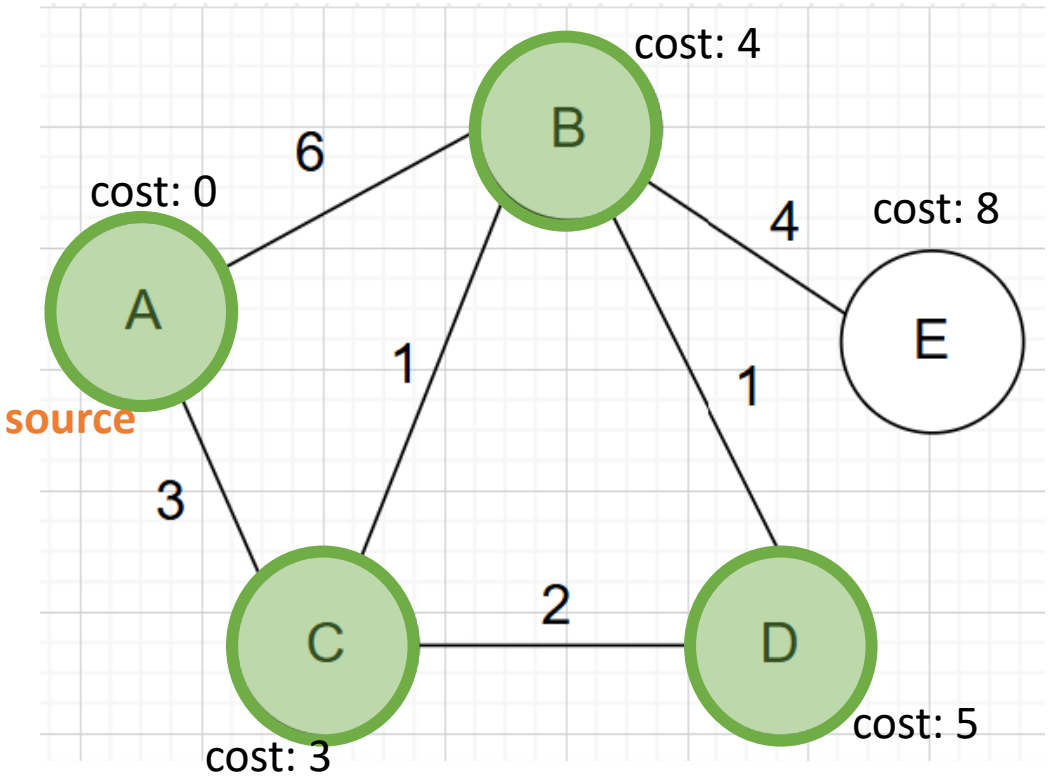


	A	B	C	D	E
parent:		C	A	C	B

Q (min priority queue):

~~(A, 0)~~
~~(C, 3)~~
~~(B, 4)~~
(D, 5)
(B, 6)
(E, 8)

Dijkstra's algorithm: how it works



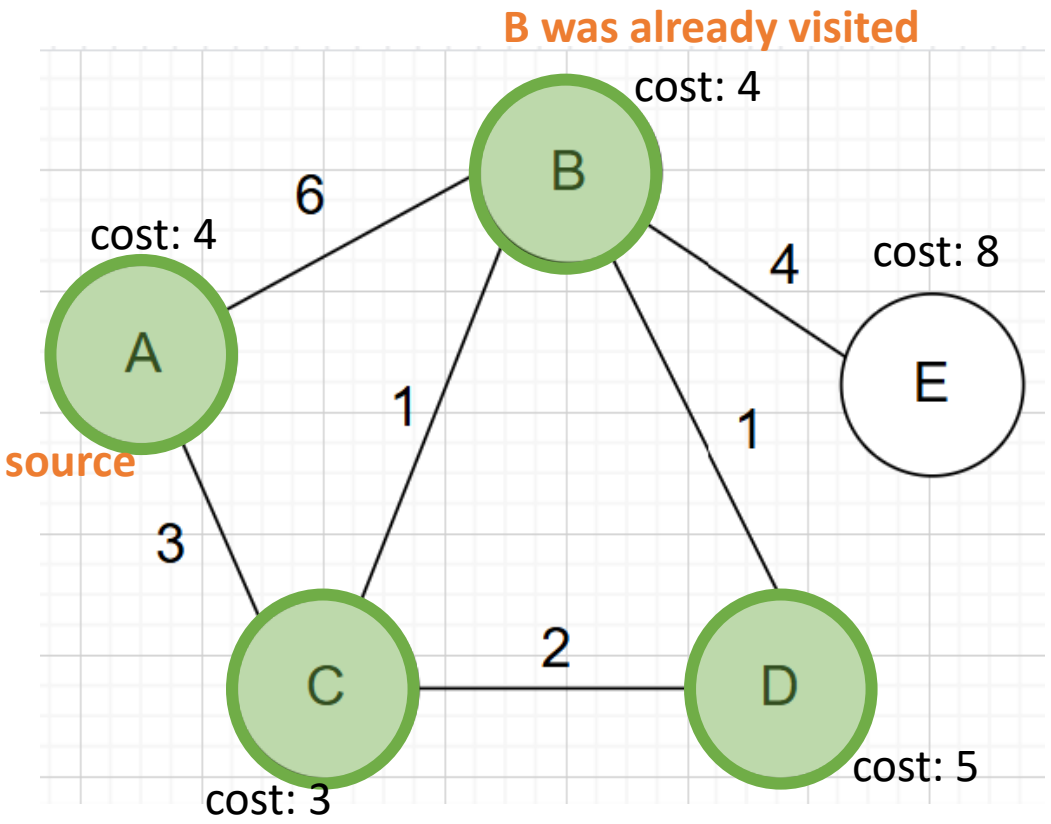
D has no unvisited neighbours

	A	B	C	D	E
parent:		C	A	C	B

Q (min priority queue):

~~(A, 0)~~
~~(C, 3)~~
~~(B, 4)~~
~~**(D, 5)**~~
(B, 6)
(E, 8)

Dijkstra's algorithm: how it works

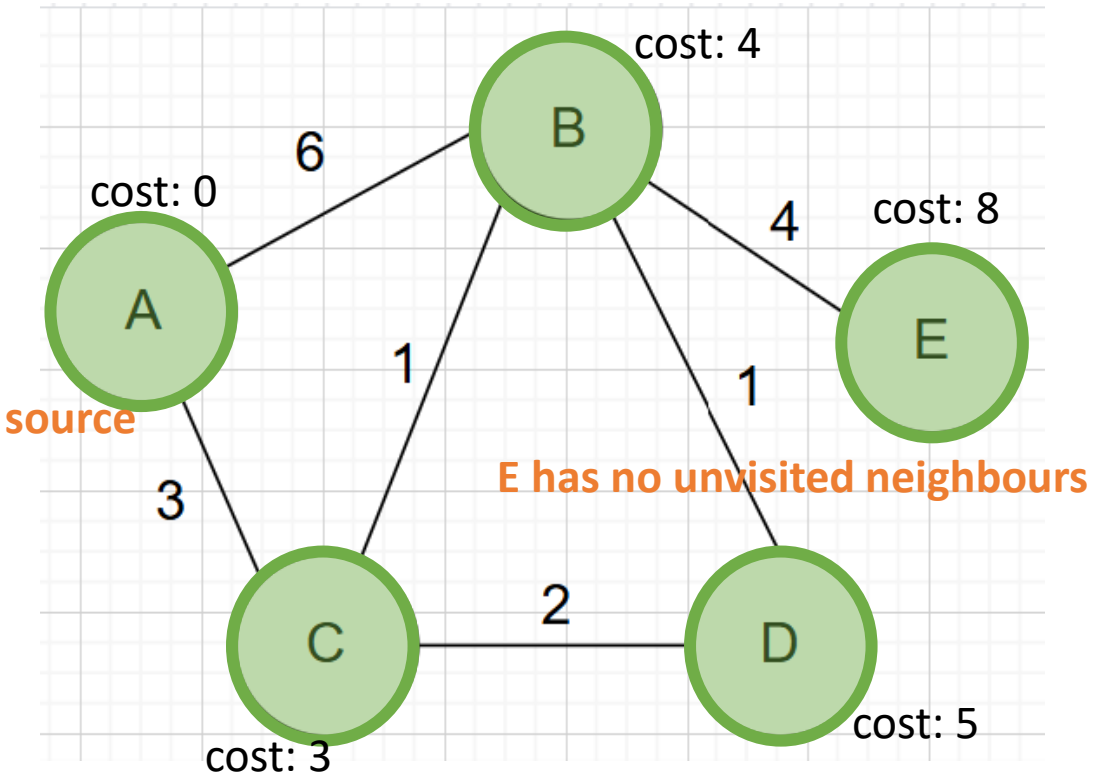


	A	B	C	D	E
parent:		C	A	C	B

Q (min priority queue):

~~(A, 0)~~
~~(C, 3)~~
~~(B, 4)~~
~~(D, 5)~~
~~**(B, 6)**~~
(E, 8)

Dijkstra's algorithm: how it works

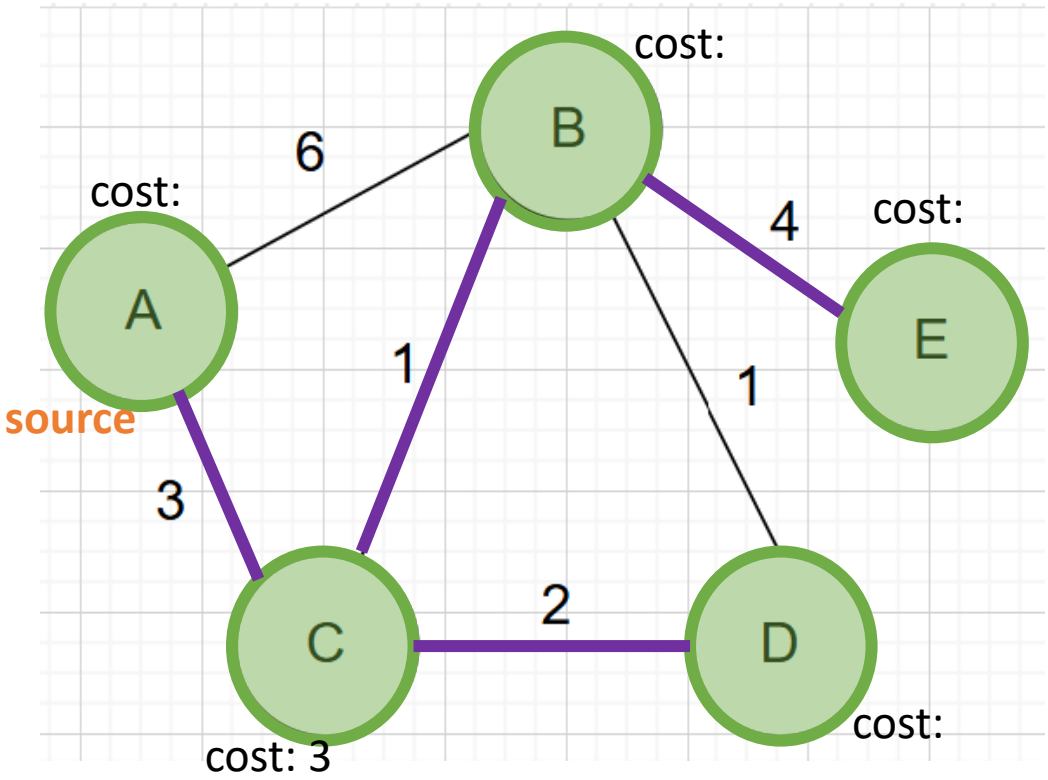


	A	B	C	D	E
parent:		C	A	C	B

Q (min priority queue):

~~(A, 0)~~
~~(C, 3)~~
~~(B, 4)~~
~~(D, 5)~~
~~(B, 6)~~
~~**(E, 8)**~~

Dijkstra's algorithm: how it works



	A	B	C	D	E
parent:		C	A	C	B

Q (min priority queue):
empty

Shortest paths from source A to each goal:

- **to B: A- C- B**
- **to C: A- C**
- **to D: A- C- D**
- **to E: A- C- B- E**

Dijkstra's algorithm: pseudocode (1)

algorithm Dijkstra **is**

Input: A graph *G,* a starting node *source* of *G*

Output: *parent* which traces the shortest path from each node back to *source*

let *Q* be a priority queue
cost[*source*] := 0
Q.add_with_priority(*source*, 0)

for each node *v* in *Graph.Nodes* ***do***
if *v* ≠ *source* **then**
parent[*v*] := UNDEFINED
cost[*v*] := INFINITY

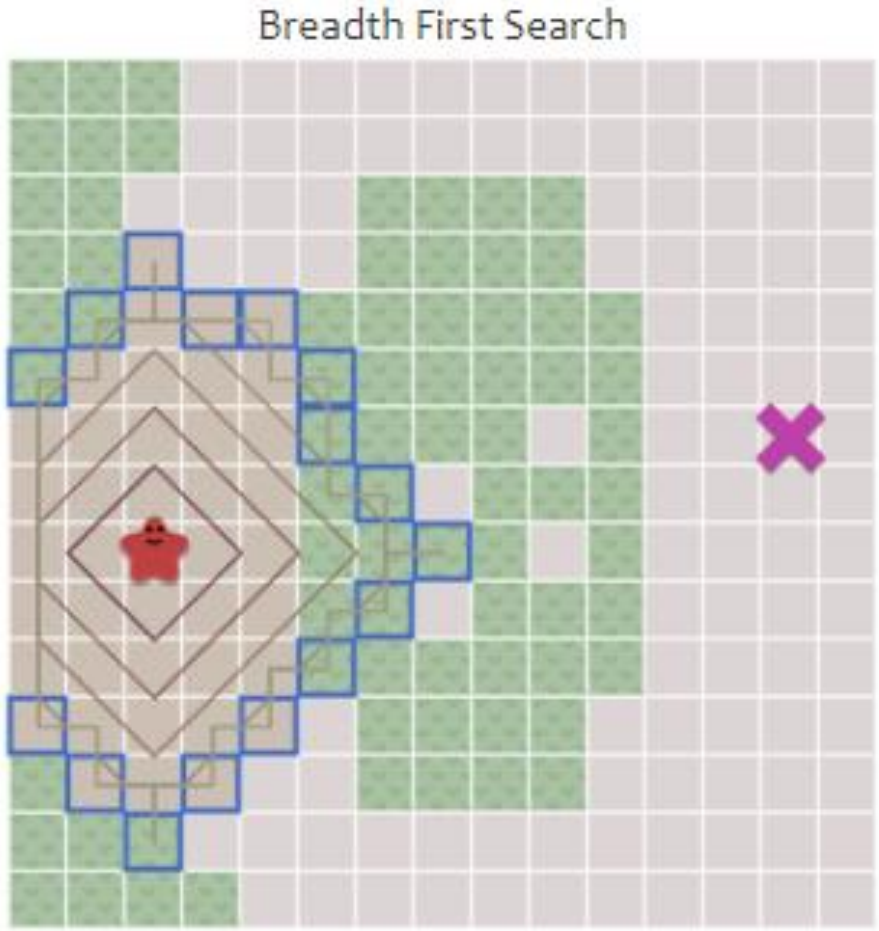
Dijkstra's algorithm: pseudocode (2)

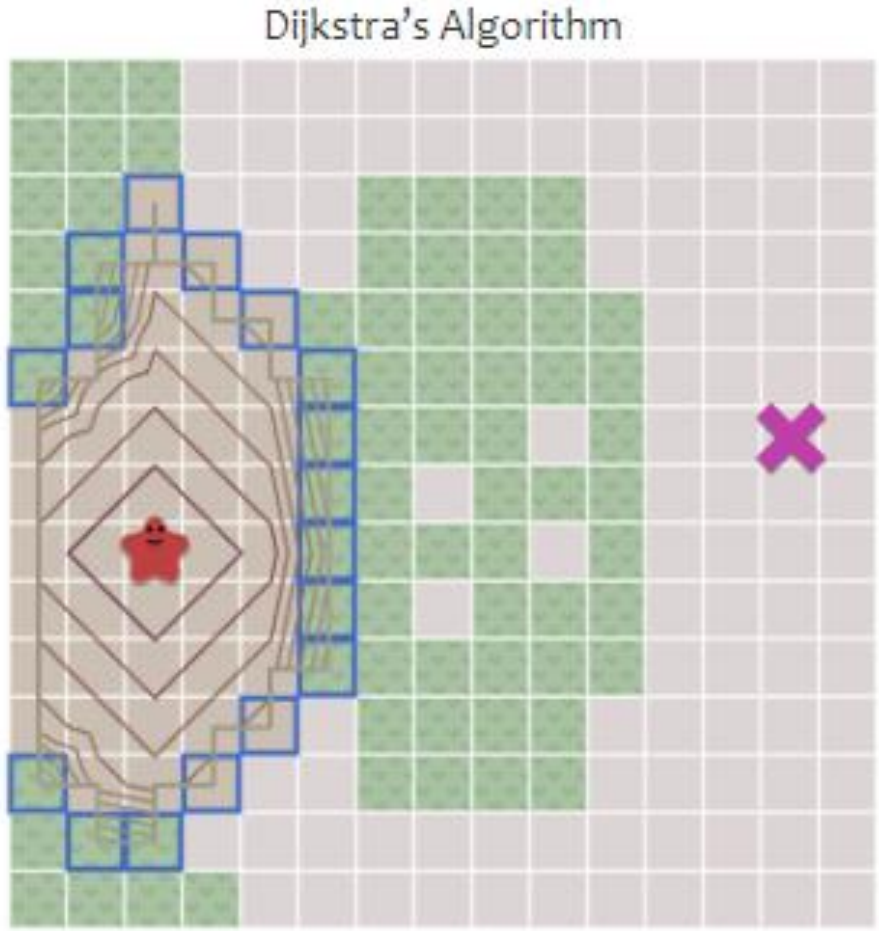
```
while Q is not empty do
        u := Q.extract_min()
        if u is not labelled as explored then          //optimization
                label u as explored                    //optimization
                for all edges from u to v in G.adjacentEdges(u) do
                        if v is not labelled as explored then          //optimization
                                new_cost := cost[u] + Graph.Edges(u, v)
                                if new_cost < cost[v] then
                                        parent[v] := u
                                        cost[v] := new_cost
                                        Q. add_with_priority (v, new_cost)
```

Dijkstra's algorithm: early exit, path to goal

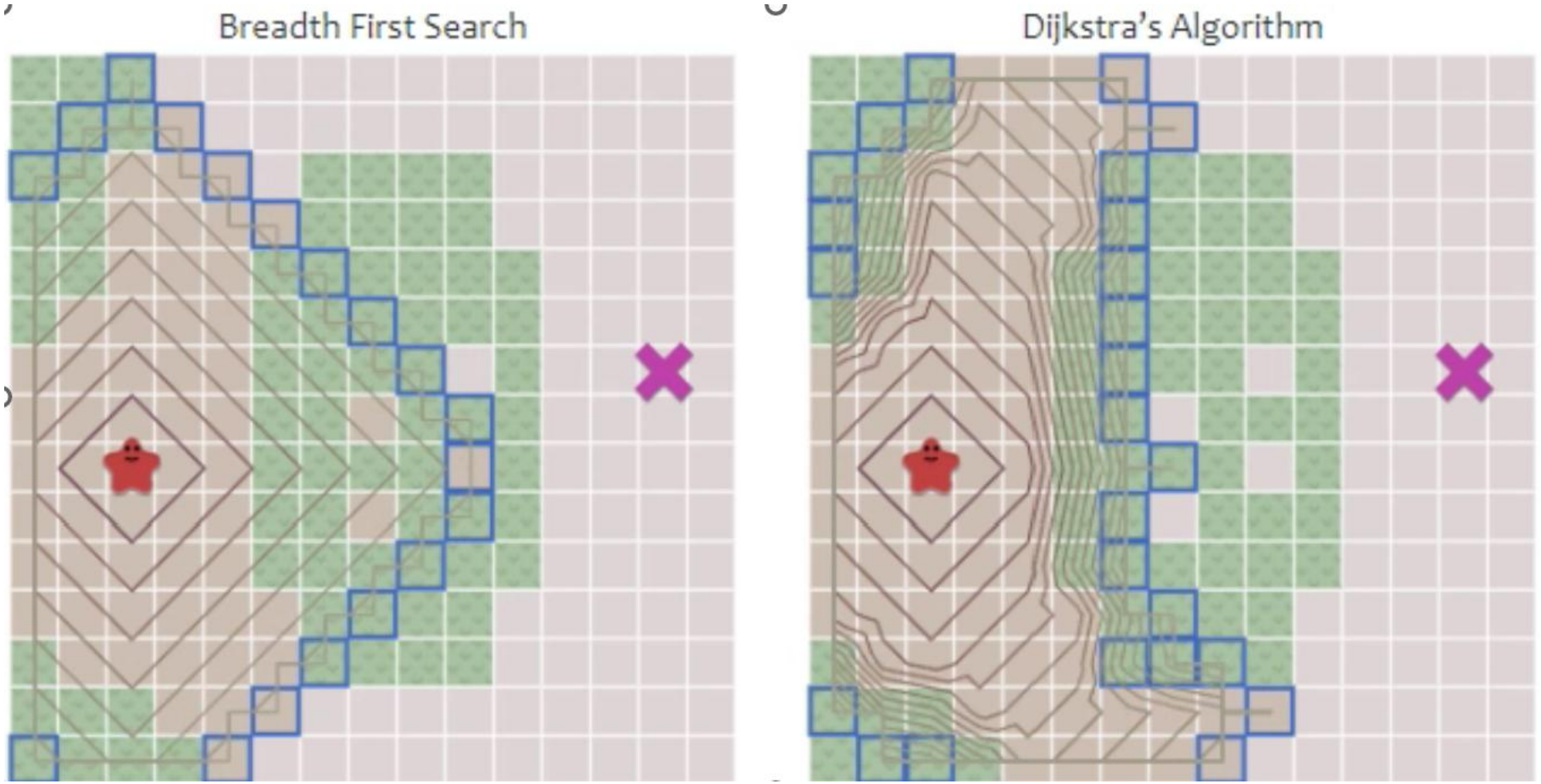
- Like BFS, Dijkstra calculates shortest path from source to all nodes, using up entire map
  - To more efficiently calculate path to a goal, can apply early exit like in BFS
- Can use same additional algorithm presented for BFS to get path to goal

Dijkstra vs BFS

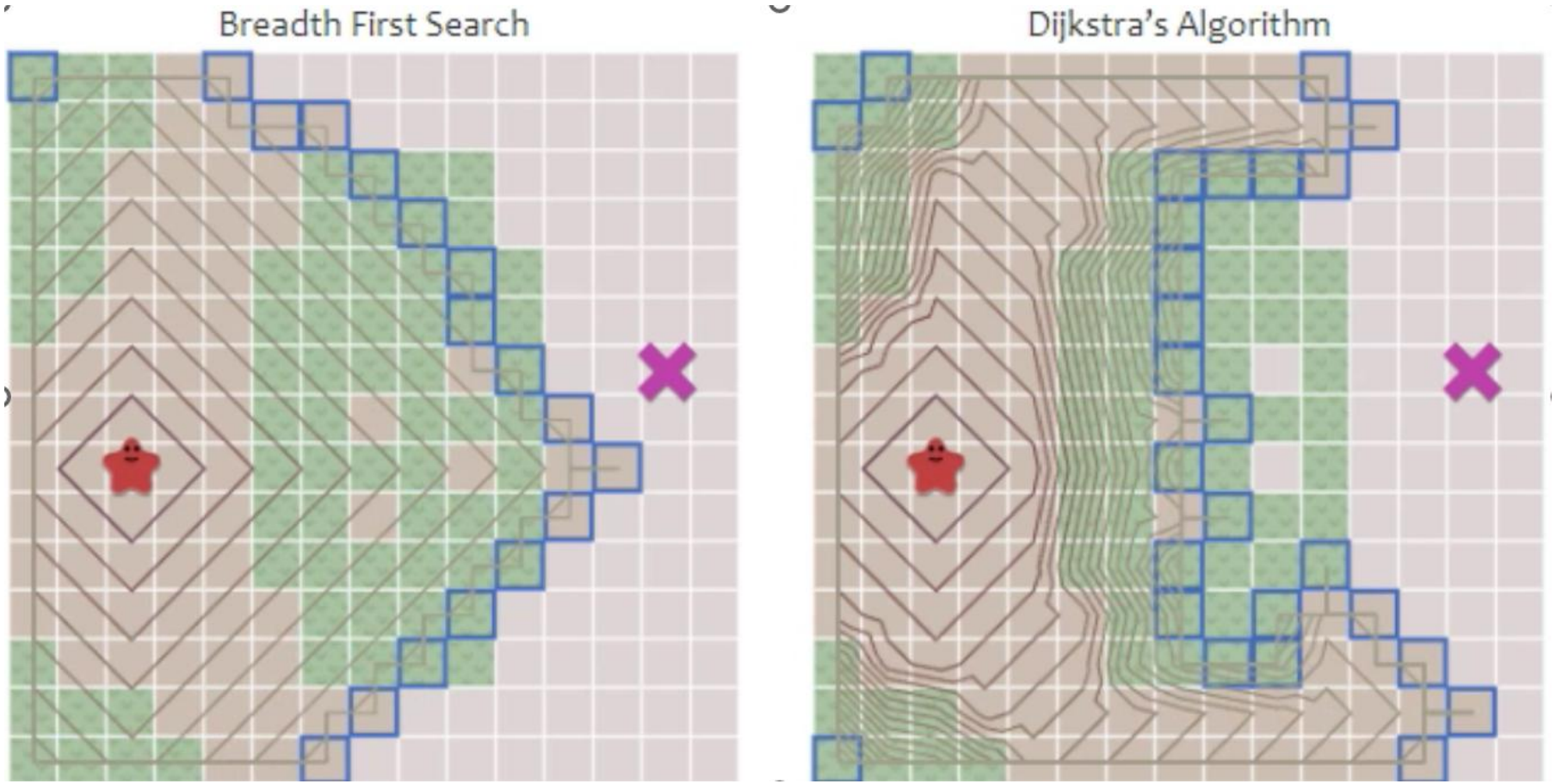




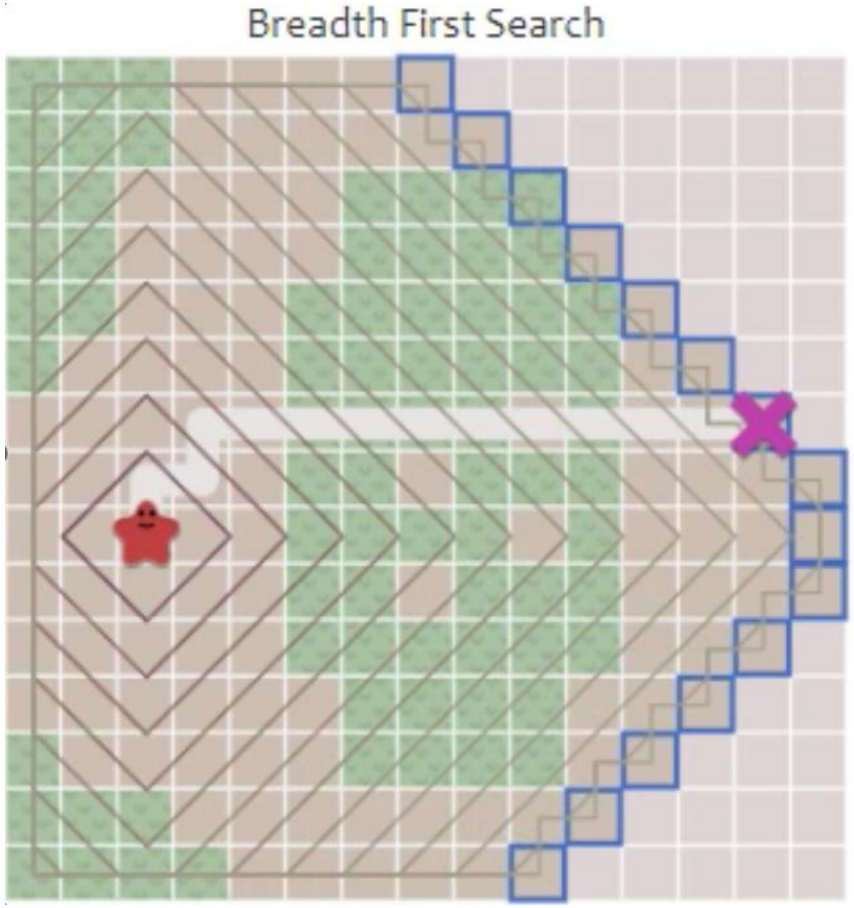
Dijkstra vs BFS

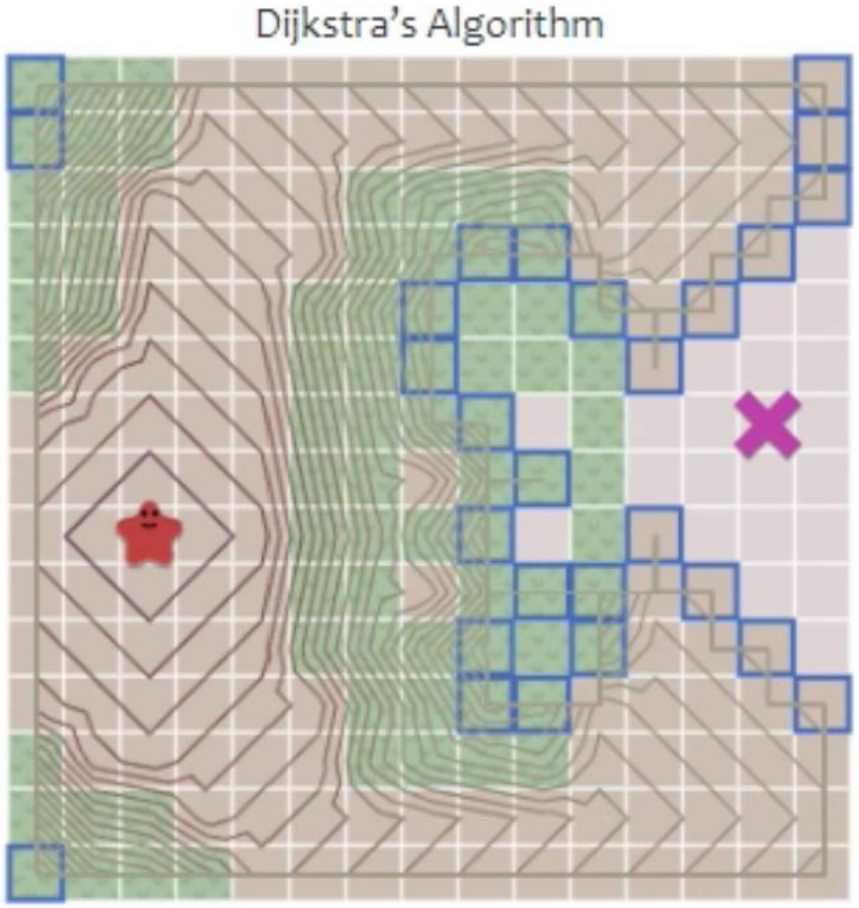


Dijkstra vs BFS



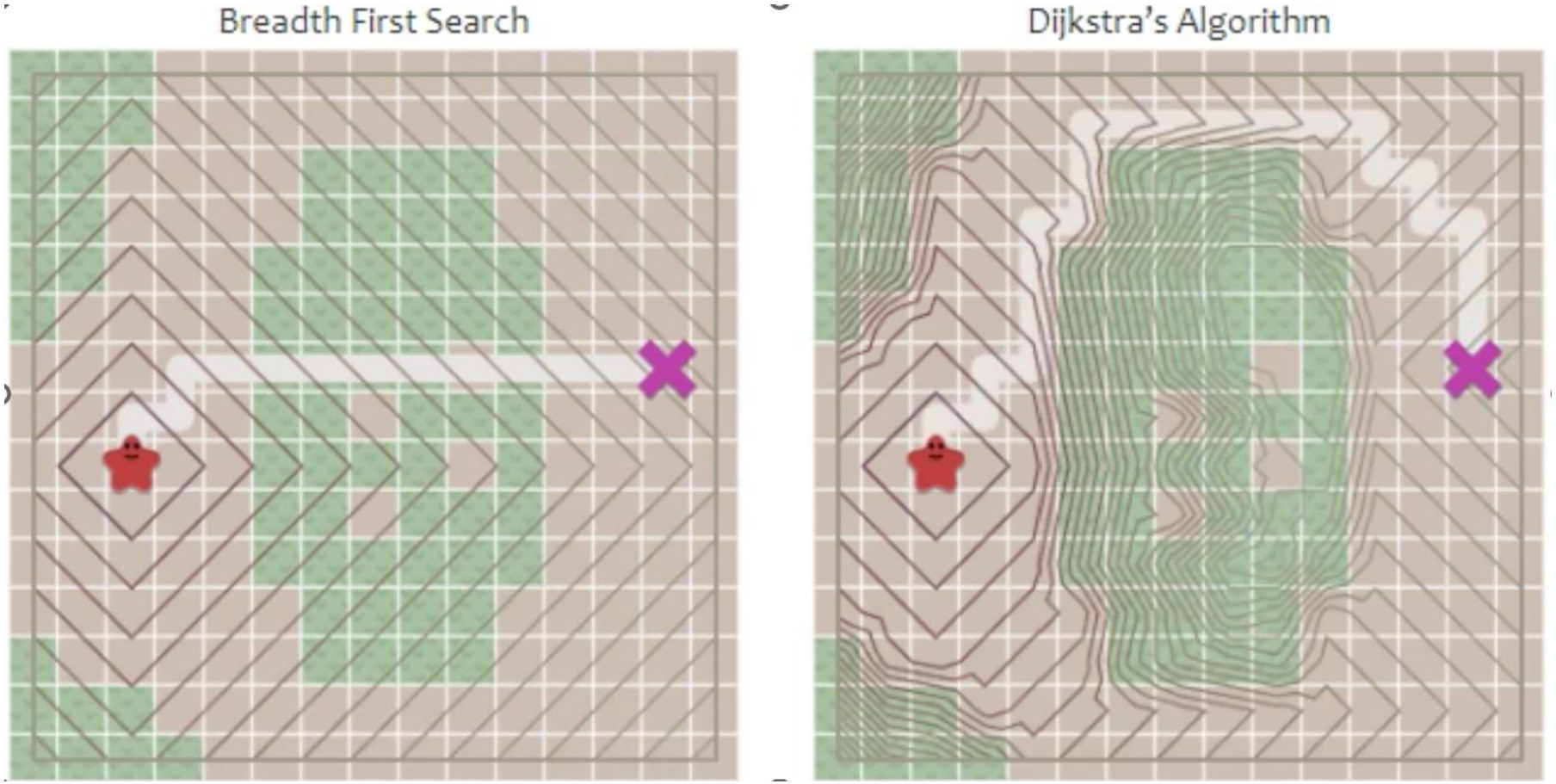
Dijkstra vs BFS





Dijkstra vs BFS

Dijkstra vs BFS

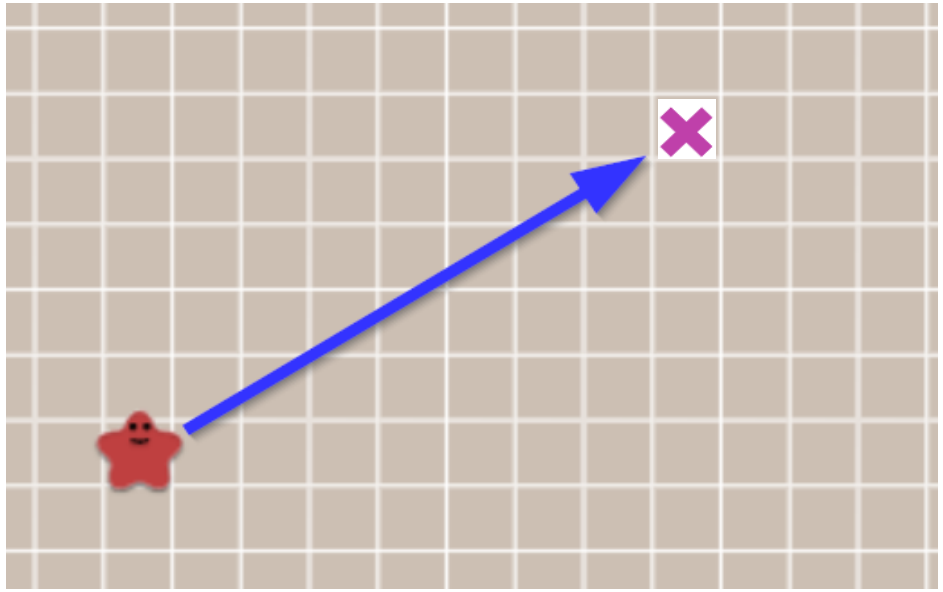


Dijkstra vs BFS

- Dijkstra
  - Excellent: finding the shortest path given varying cost
- BFS
  - Better: finding the shortest path given equal cost (faster)
  - Bad: finding the shortest path when varying cost (does not consider cost)
- Both
  - Good: [source] to all other locations
  - Bad: [source] to [goal], even with early exit

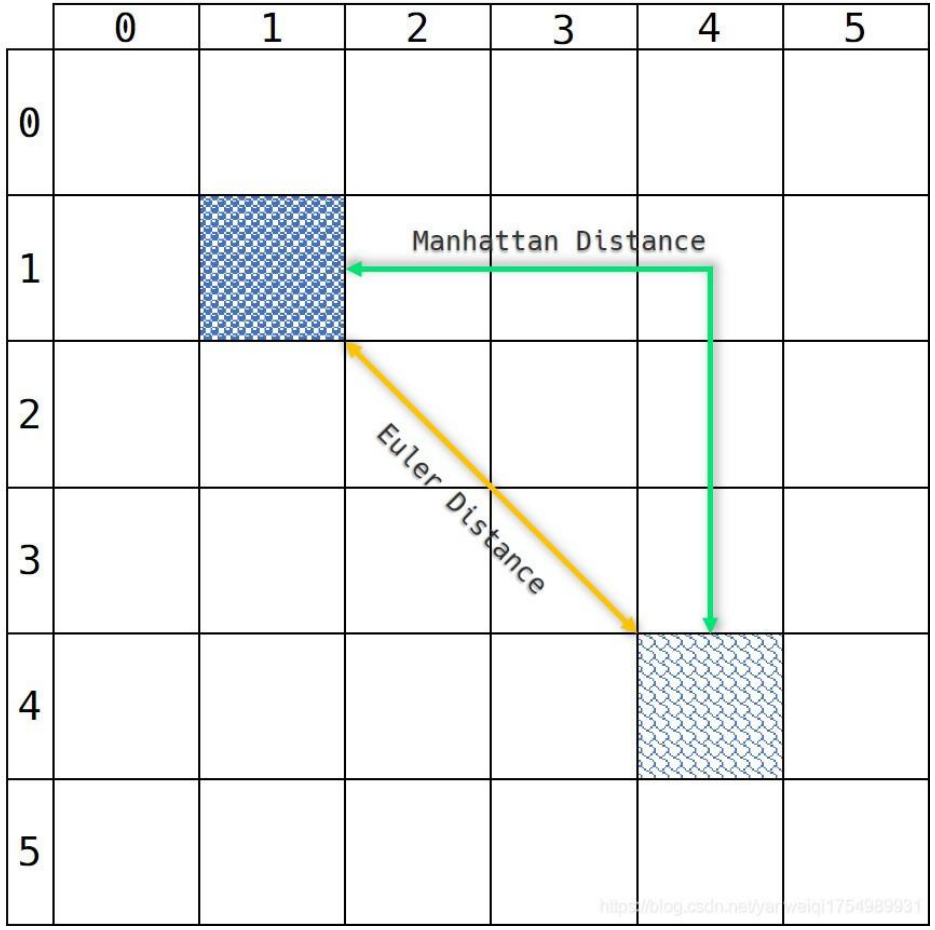
Heuristic Search

- Expand towards the goal



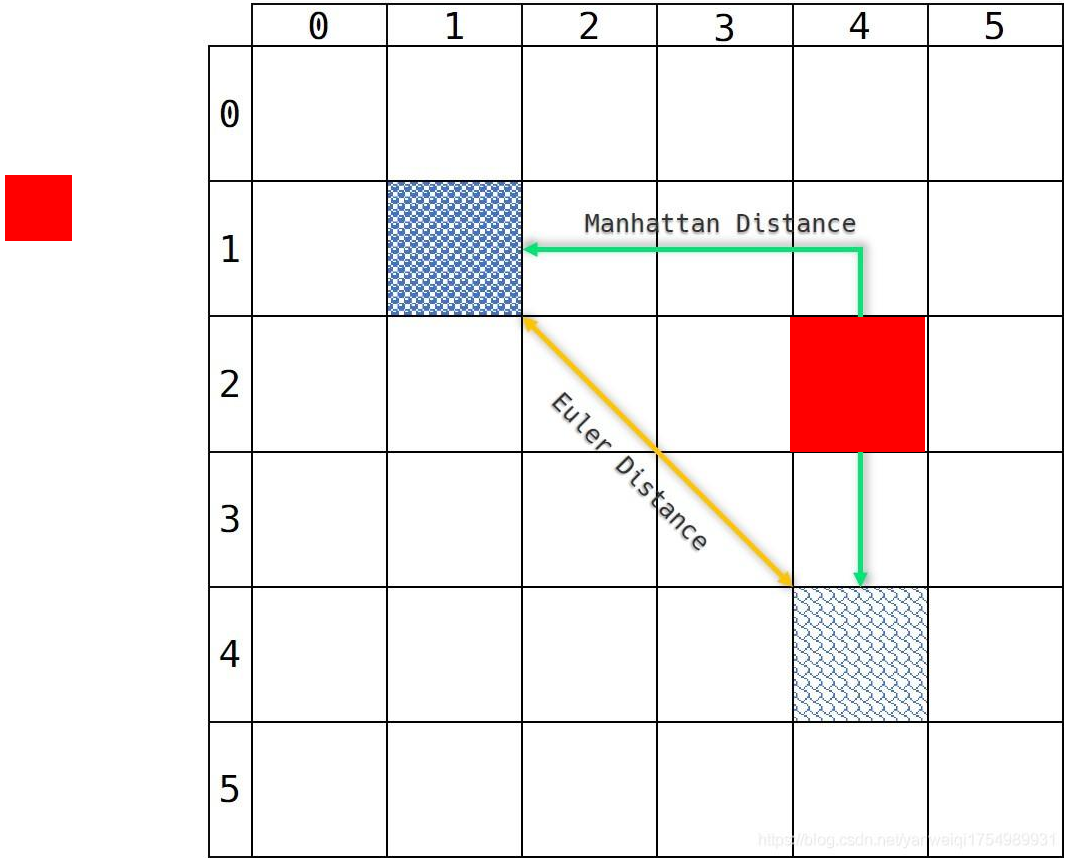
Heuristics for Grid Maps

- How close to the goal
  - Manhattan distance
  - Euler Distance
  - …



Heuristics for Grid Maps

- How close to the goal
  - Estimation
  - There may be obstacles



A* algorithm: goal

- Find the shortest path from a source node ***to goal node***, ***while also considering movement cost***
  - Unlike BFS and Dijkstra, focusing on path to goal
  - Unlike BFS, considering movement cost
- It prioritises smaller movement cost (like Dijkstra) but also ***using heuristics to guide search***

  => better performance if heuristic chosen well

A* algorithm: main concepts

- Very similar to Dijkstra, but:
  - **F = G + H used to prioritise which neighbour to choose next**
    - G = exact cost (like in Dijkstra) from the source to a node
    - H = heuristic estimated cost (must be implemented separately) from a node to the goal
  - **Prioritising lowest F when choosing next node to explore**

- Dijkstra is special case of A* where heuristic returns 0 for all nodes

A* algorithm: pseudocode

algorithm A-Star **is**

Input: A graph *G,* a starting node *source* of *G*, an ending node *goal* of G

Output*: parent* which traces the shortest path from *goal* to *source*

let *Q* be a priority queue

cost[*source*] := 0 //gscore

fscore[*source*] := heuristic(*source)*

Q.add_with_priority(*source*, 0)

for each node *v* in *Graph.Nodes* ***do***

if *v* ≠ *source* **then**

parent[*v*] := UNDEFINED

cost[*v*] := INFINITY

fscore[*v*] := INFINITY

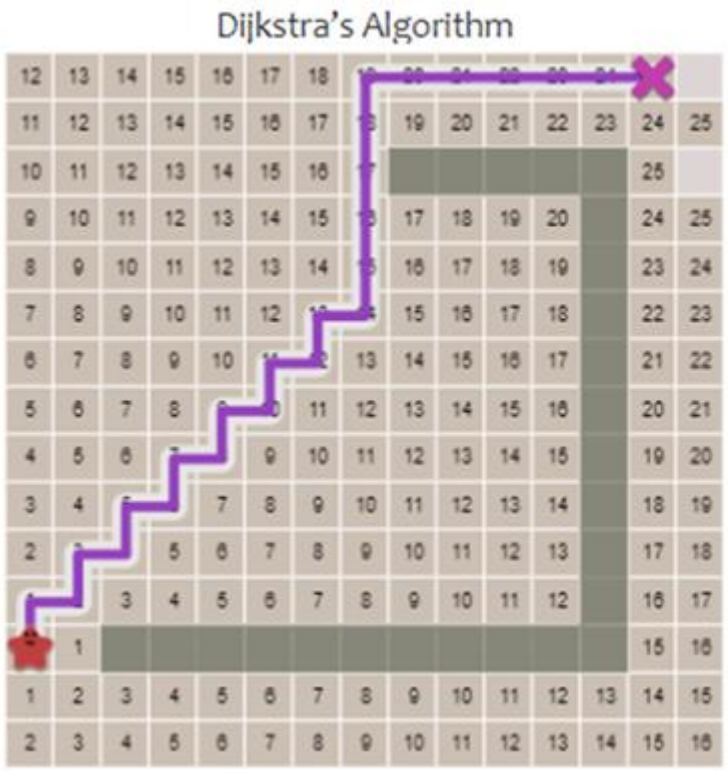
Differences to Dijkstra are highlighted

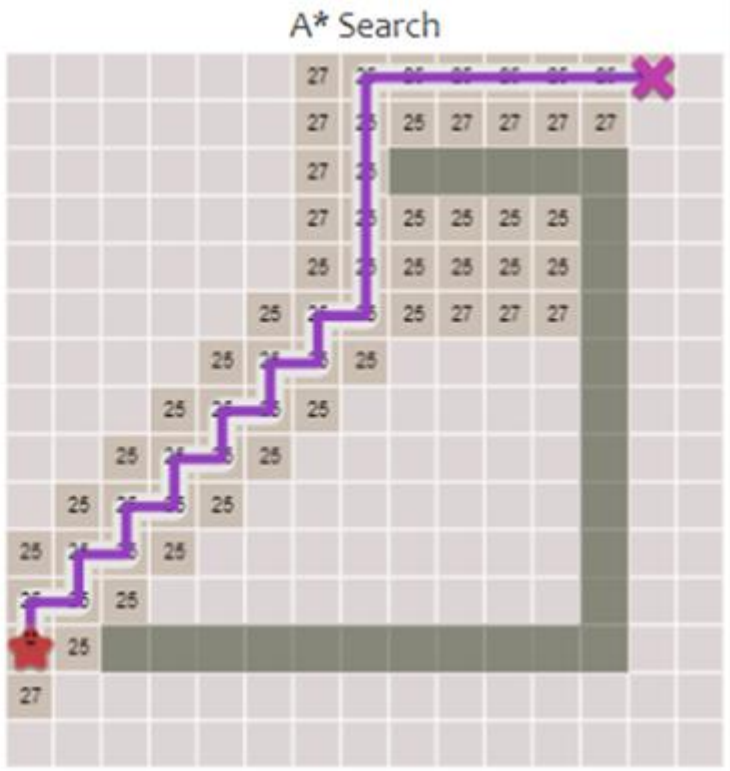
A* algorithm: pseudocode (2)

```
while Q is not empty do
        u := Q.extract_min()                    // will extract tuple with min fscore
        if u is goal then
                return calculatePath(parent, goal)        //same shown earlier
        if u is not labelled as explored then       //optimization
                label u as explored                 //optimization
        for all edges from u to v in G.adjacentEdges(u) do
                if v is not labelled as explored then       //optimization
                        new_cost := cost[u] + Graph.Edges(u, v)
                        if new_cost < cost[v] then
                                parent[v] := u
                                cost[v] := new_cost
                                fscore[v] := new_cost + heuristic(v)
                                Q. add_with_priority (v, fscore[v])
return failure          //Q  empty but goal not reached
```

Differences to Dijkstra are highlighted

A* vs Dijkstra





A* vs Dijkstra

- A*
  - Excellent: Guarantees shortest path [source] to [goal] (when varying or fixed cost) ***when heuristic does not overestimate cost***
  - Good: More efficient than Dijkstra if heuristic chosen well
  - Good: Complexity typically better than Dijkstra
  - Bad: performance and efficiency depends on quality of heuristic

- Dijkstra:
  - Excellent: Guarantees shortest path for all nodes when varying cost
  - Bad: Can be slow and less efficient for large graphs
  - Bad: [source] to [goal], even with early exit

Conclusion

- Breadth First Search: explores equally in all directions

- Dijkstra's Algorithm: prioritizes paths with lowest cost to explore

- A*: prioritizes paths that are lower cost as well as seem to be leading closer to a goal