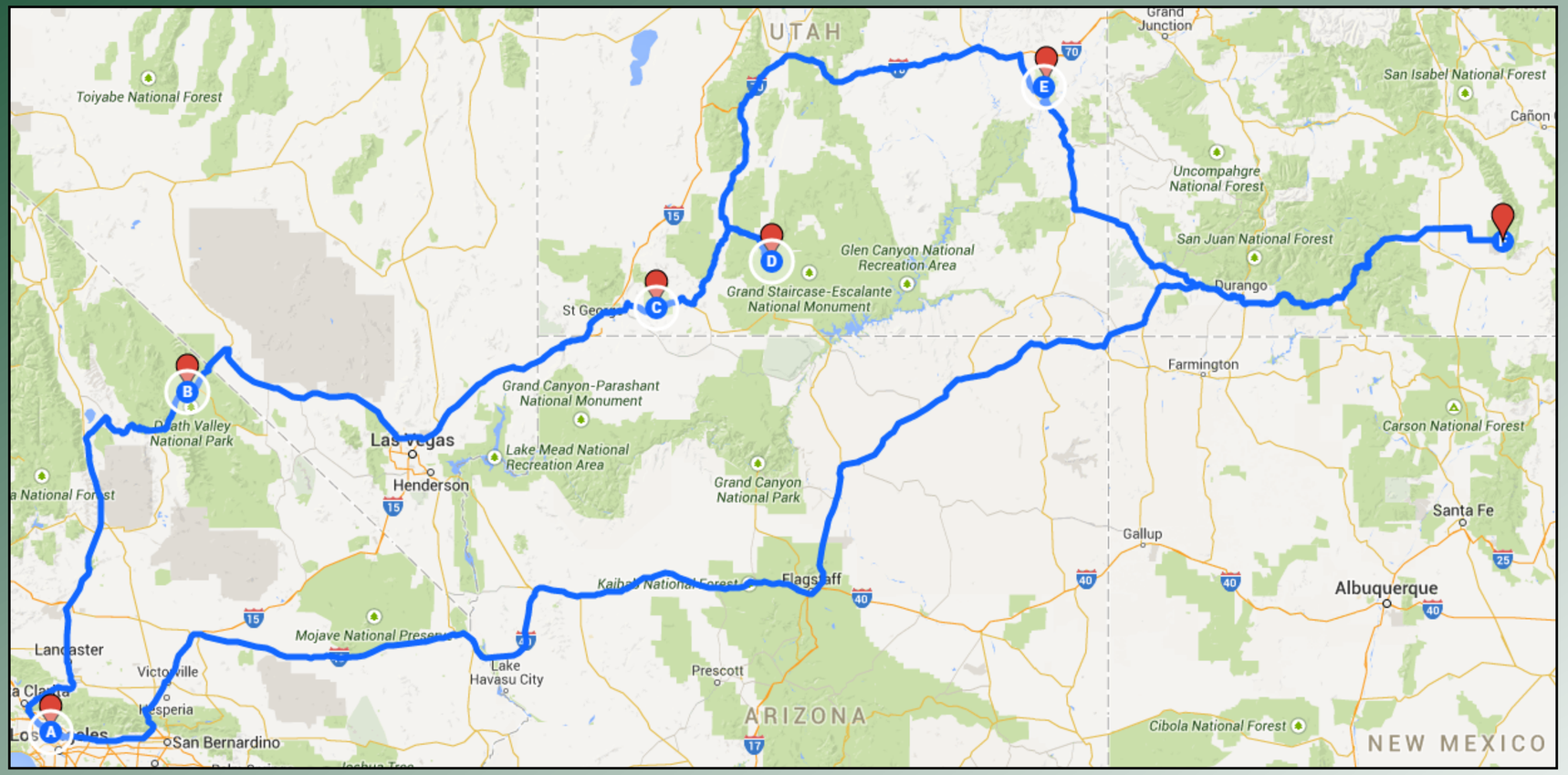


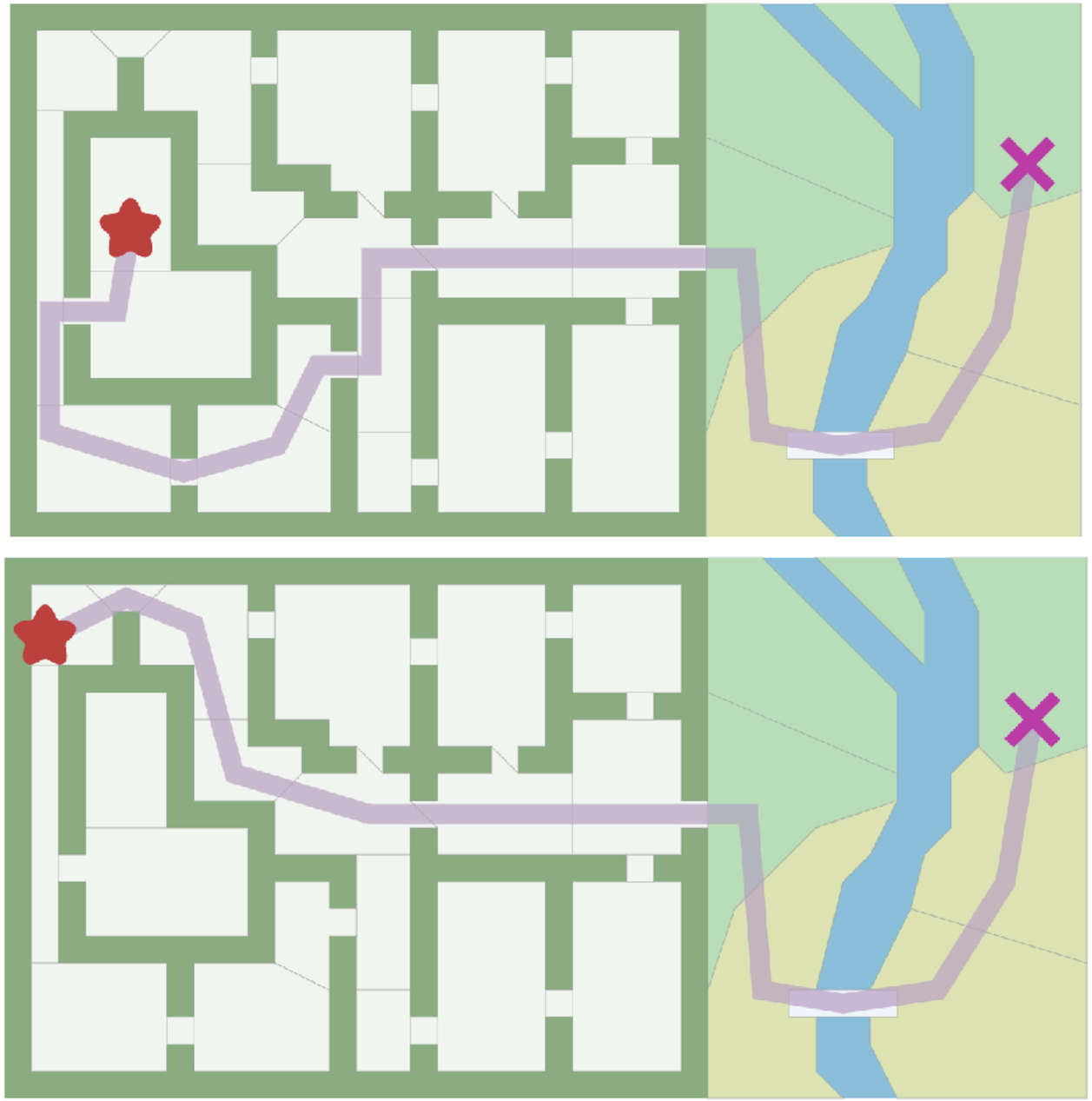
Planning 2

ENAE 380 Flight Software Systems

Lecture 15

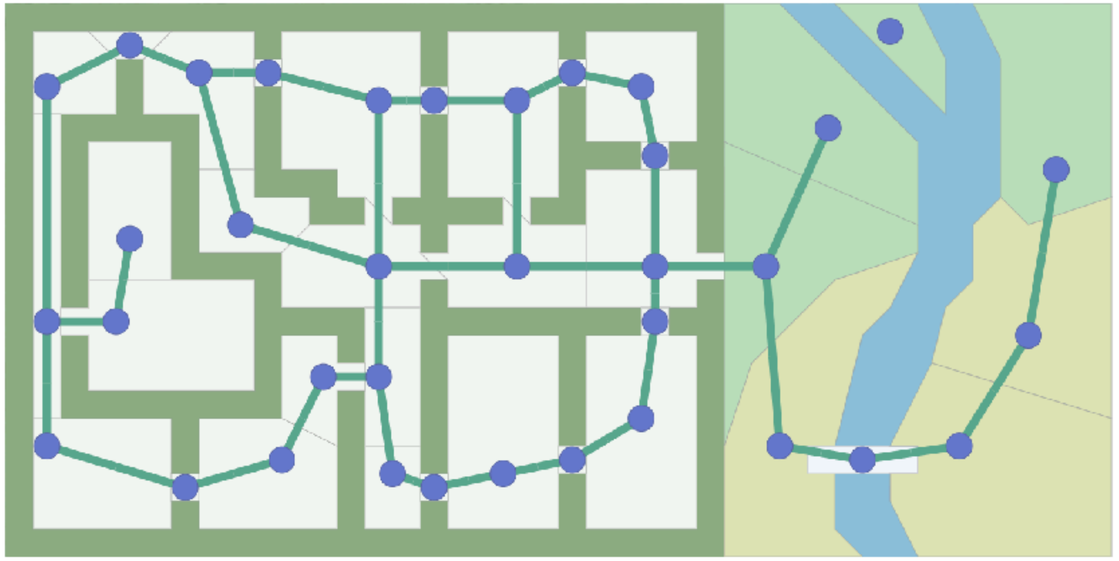
October 30, 2024

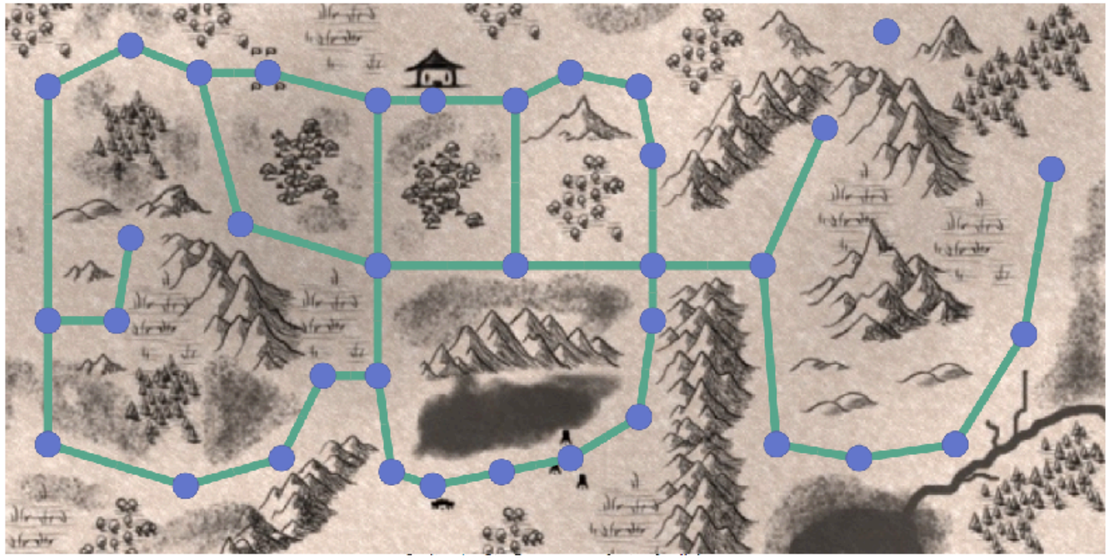
Searches



Representing the Map

Input: Graph, represented by a set of locations (nodes) and connections (edges) between them

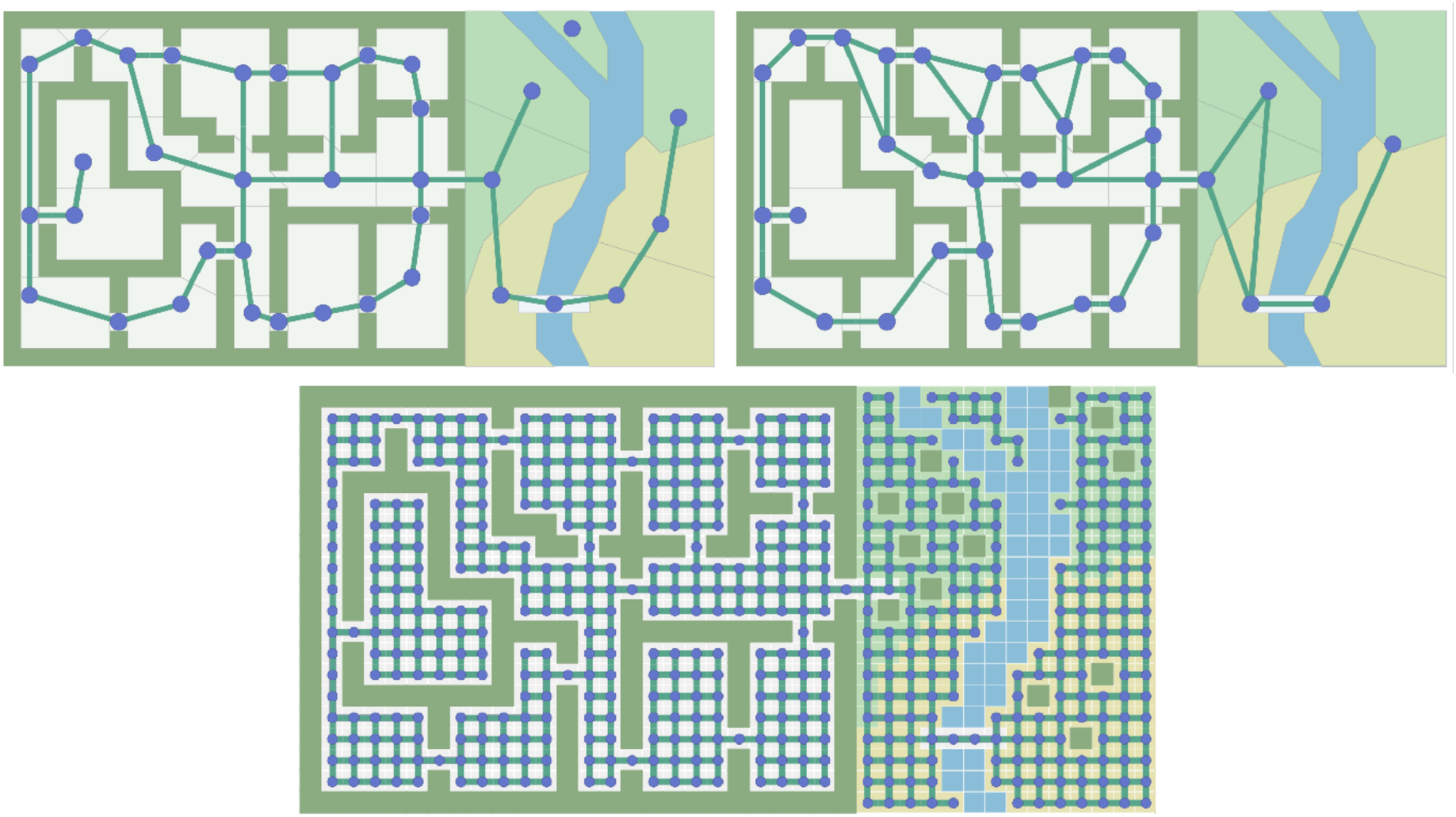




Representing the Map

Output: Path made of nodes and edges. A* will tell you to move from one node to another, but it won't tell you how.

Tradeoffs: Doorways can be nodes or edges.



Algorithms

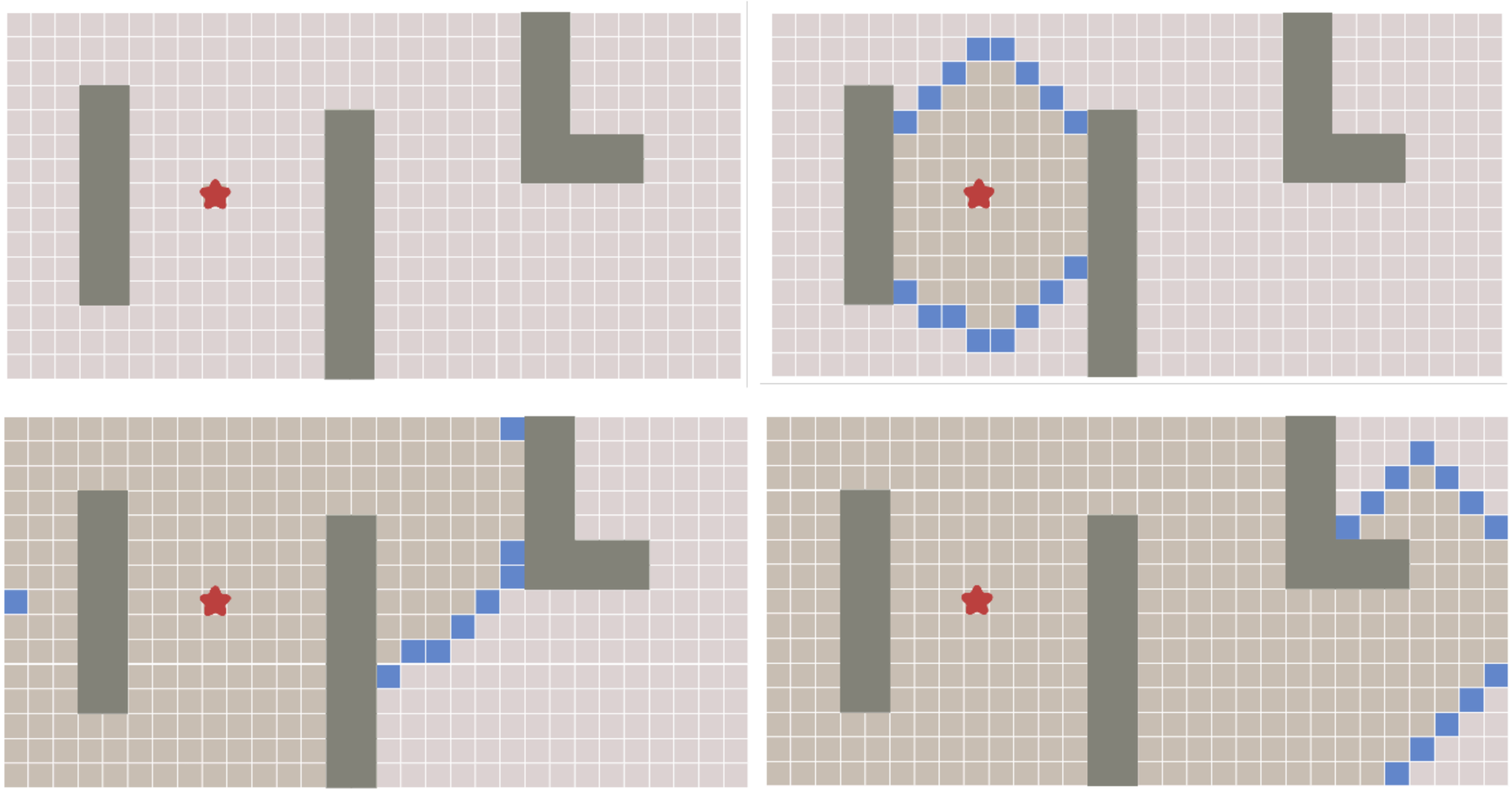
Breadth First Search: Explores equally in all directions. Useful for a number of applications.

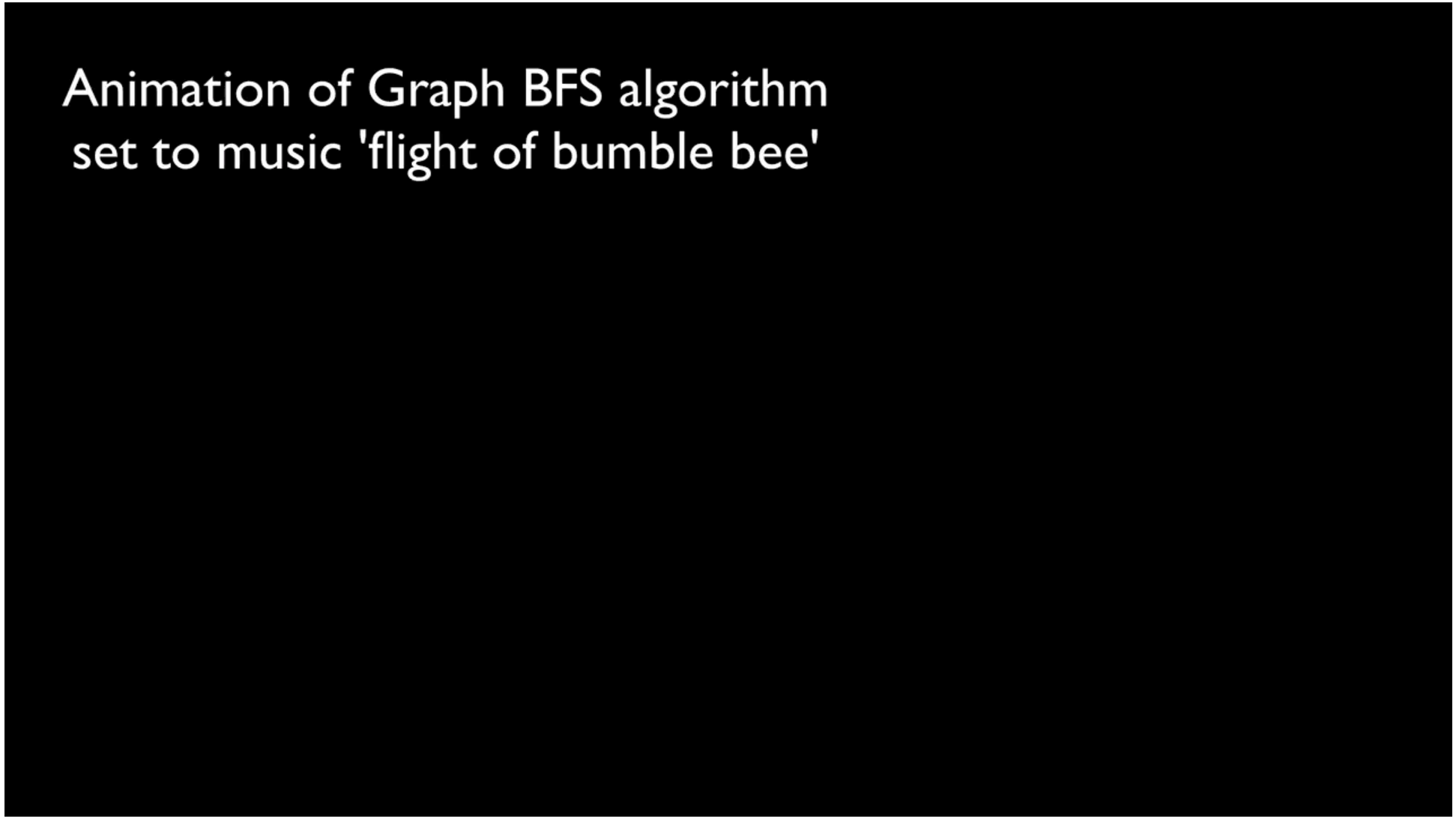
Dijkstra's Algorithm: Prioritize which paths to explore, favoring lower cost paths.

A*: Modification of Dijkstra's Algorithm that is optimized for a single destination. Prioritizes paths leading closer to a goal.

Breadth First Search

Keep track of an expanding ring called the **frontier** (aka flood fill on a grid)

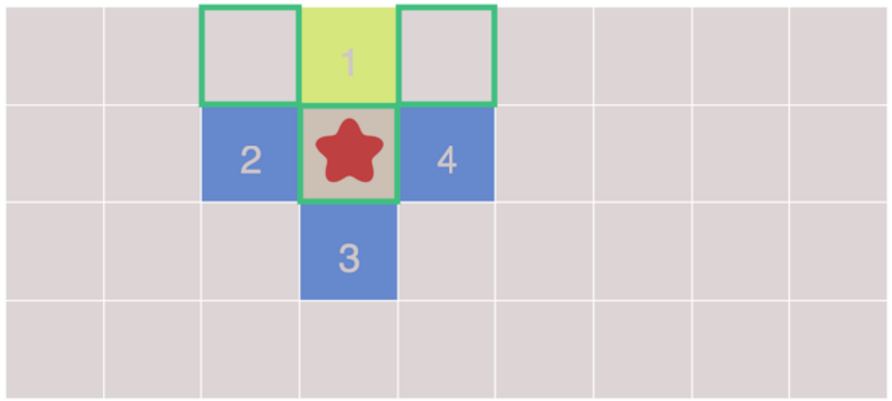


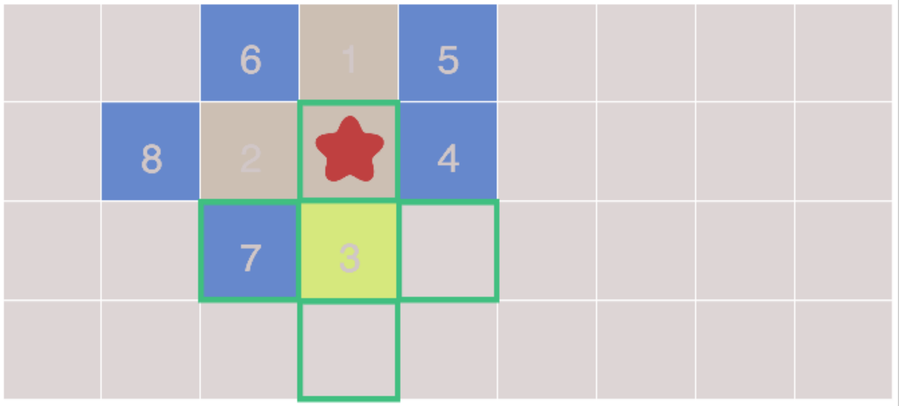

Animation of Graph BFS algorithm
set to music 'flight of bumble bee'

Breadth First Search

Repeat these steps until frontier is empty

1. Pick and remove a location from the frontier
2. Expand it by looking at its neighbors. Any neighbors we haven't visited yet we add to the frontier, and also to the visited set





Breadth First Search

Repeat these steps until frontier is empty

1. Pick and remove a location from the frontier
2. Expand it by looking at its neighbors. Any neighbors we haven't visited yet we add to the frontier, and also to the visited set

```
frontier = Queue()
frontier.put(start )
visited = {}
visited[start] = True

while not frontier.empty():
   current = frontier.get()
   for next in graph.neighbors(current):
      if next not in visited:
         frontier.put(next)
         visited[next] = True
```

Breadth First Search

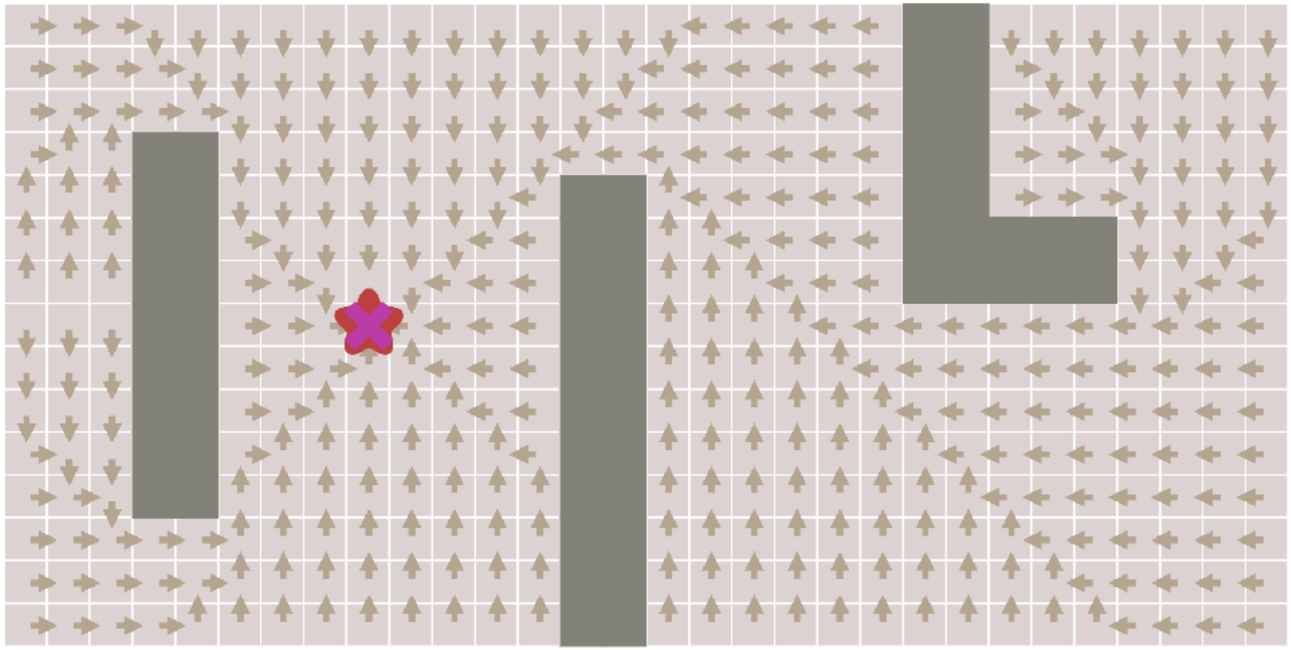
Repeat these steps until frontier is empty

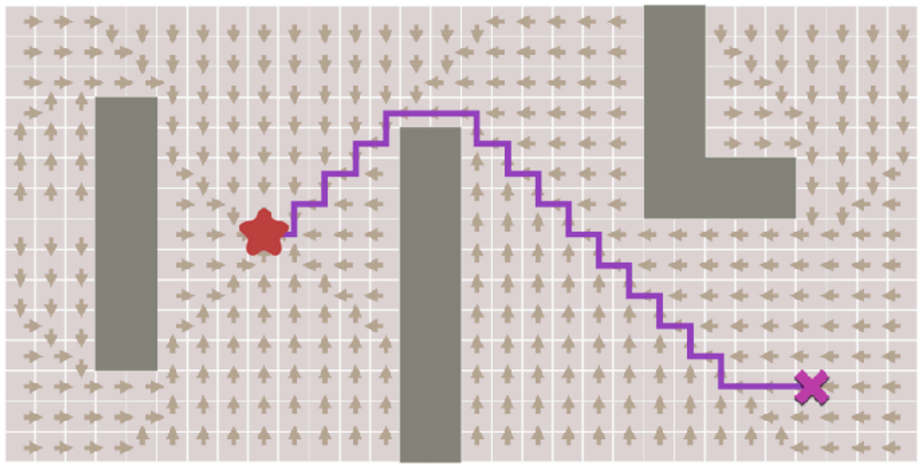
1. Pick and remove a location from the frontier
2. Expand it by looking at its neighbors. Any neighbors we haven't visited yet we add to the frontier, and also to the visited set

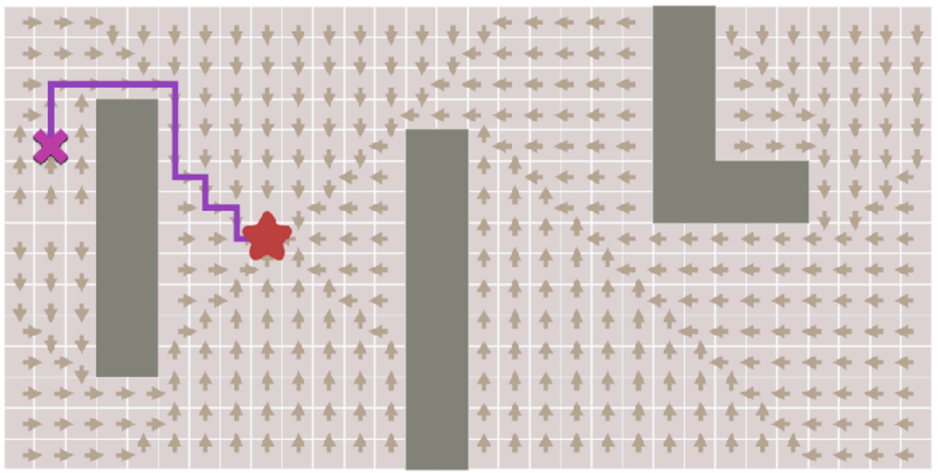
```
frontier = Queue()
frontier.put(start )
visited = {}
visited[start] = True

while not frontier.empty():
   current = frontier.get()
   for next in graph.neighbors(current):
      if next not in visited:
         frontier.put(next)
         visited[next] = True
```

```
frontier = Queue()
frontier.put(start )
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()
   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
         came_from[next] = current
```

Breadth First Search



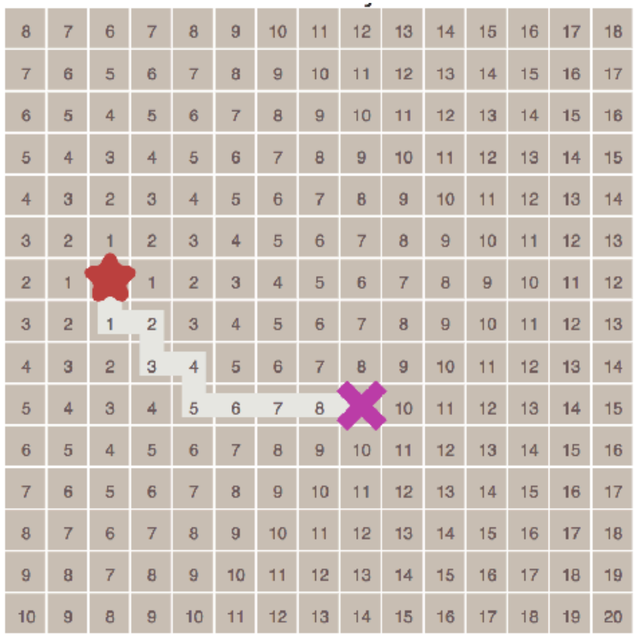


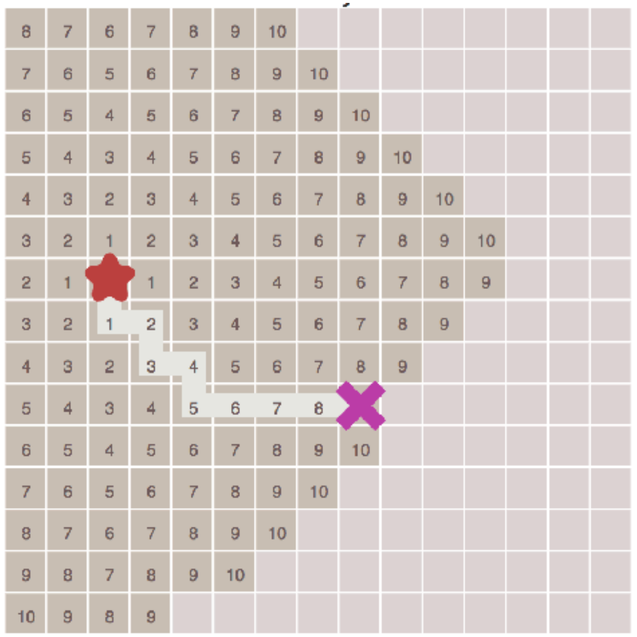


Breadth First Search

```
current = goal
path = []
while current != start:
    path.append(current)
    current = came_from[current]
path.append(start) # optional
path.reverse() # optional
```

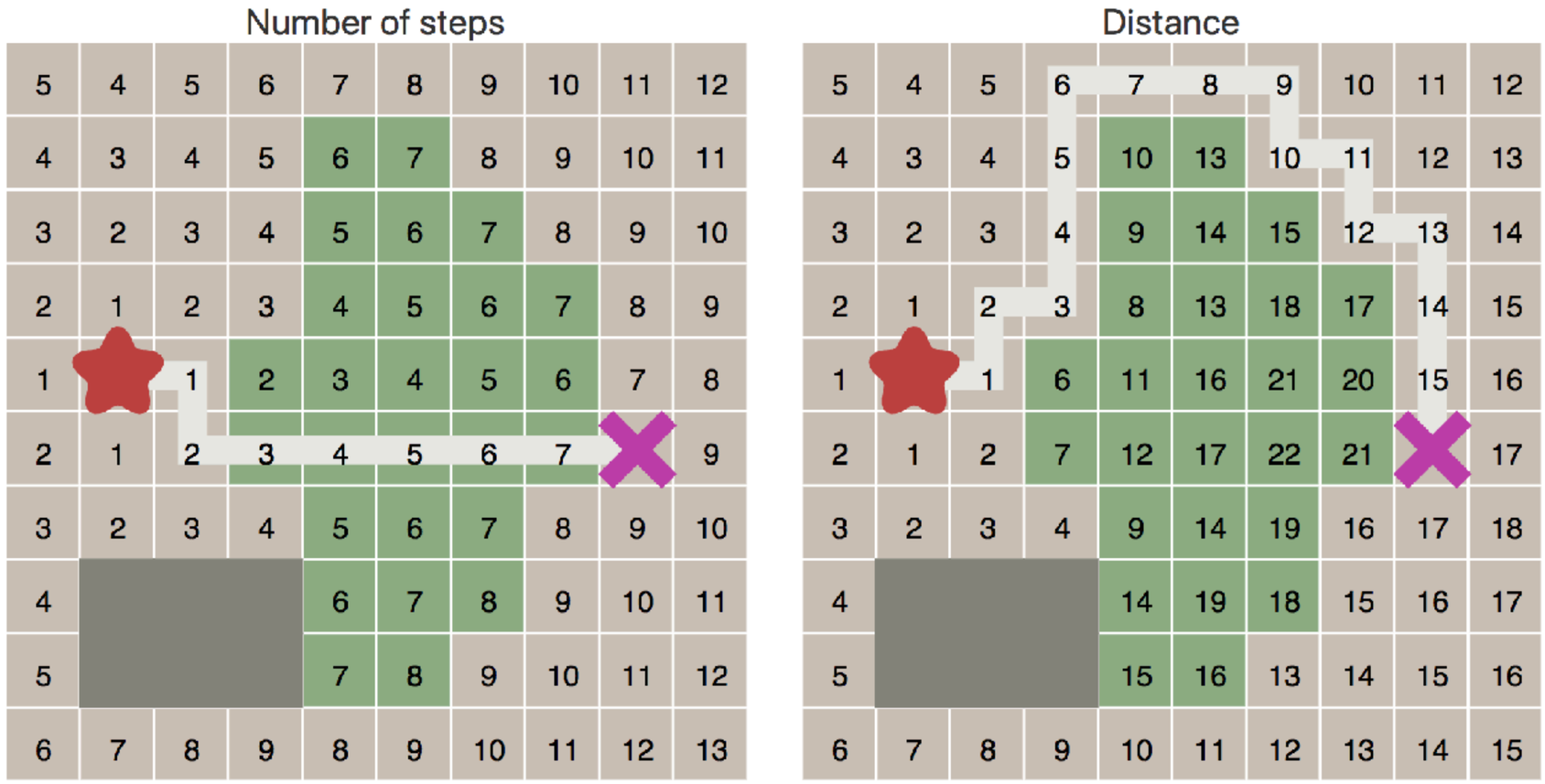
Stop Conditions






```
frontier = Queue()
frontier.put(start )
came_from = {}
came_from[start] = None

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      if next not in came_from:
         frontier.put(next)
         came_from[next] = current
```

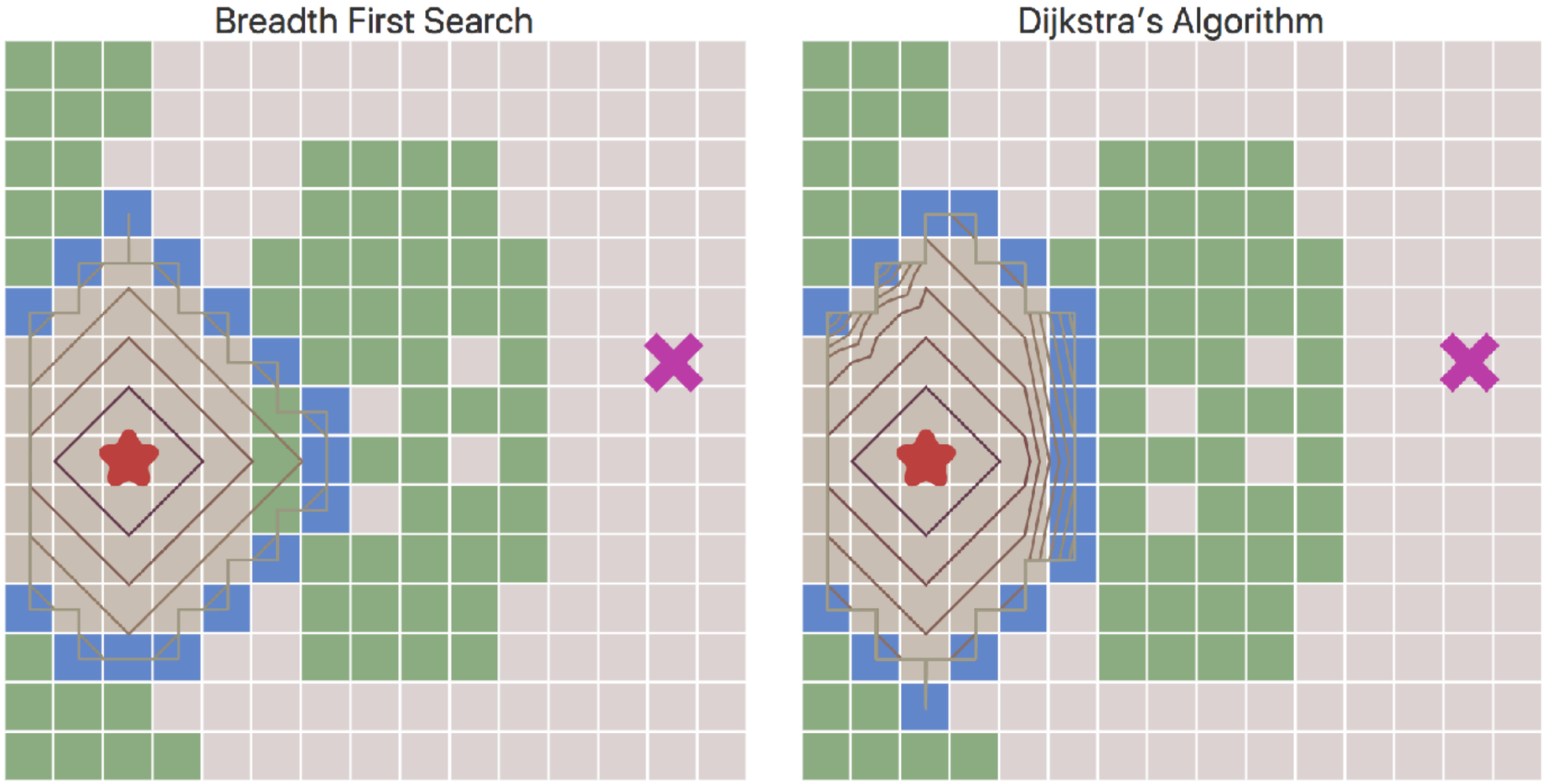
Movement Costs



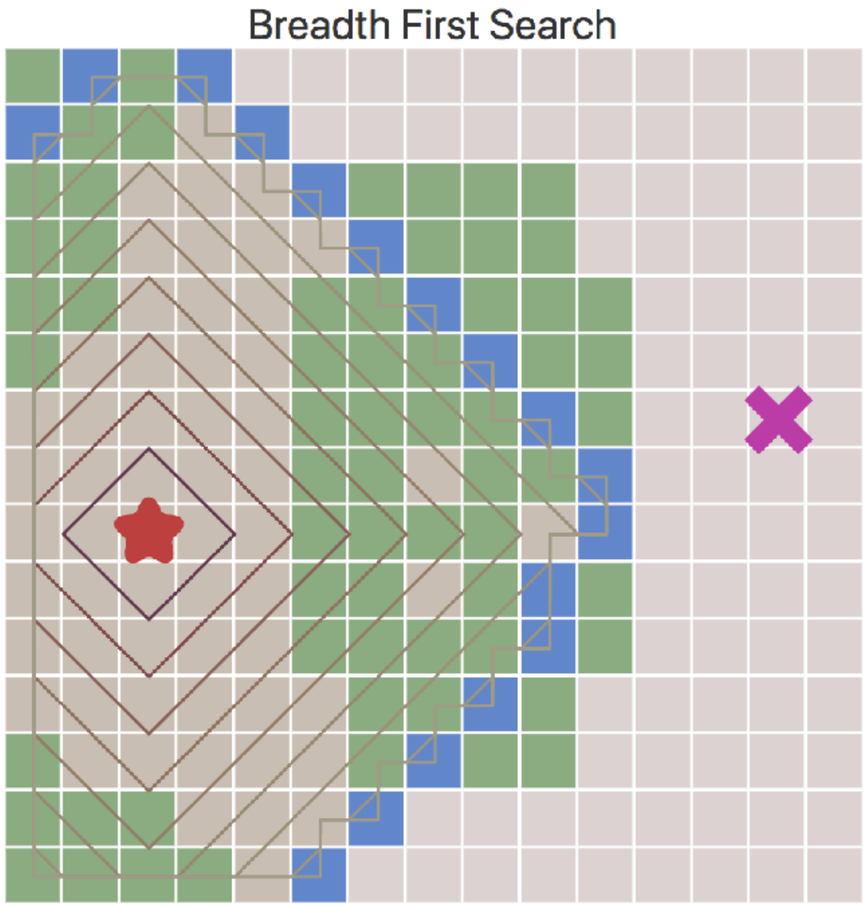
Dijkstra's Algorithm

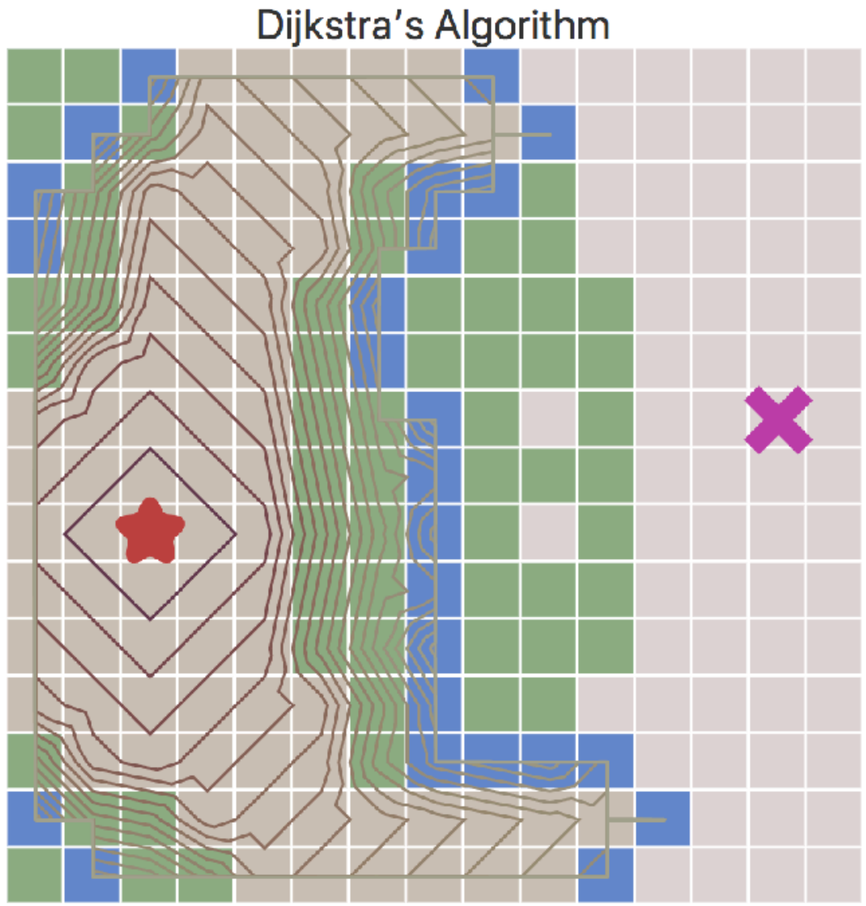
```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost
         frontier.put(next, priority)
         came_from[next] = current
```

Dijkstra’s Algorithm

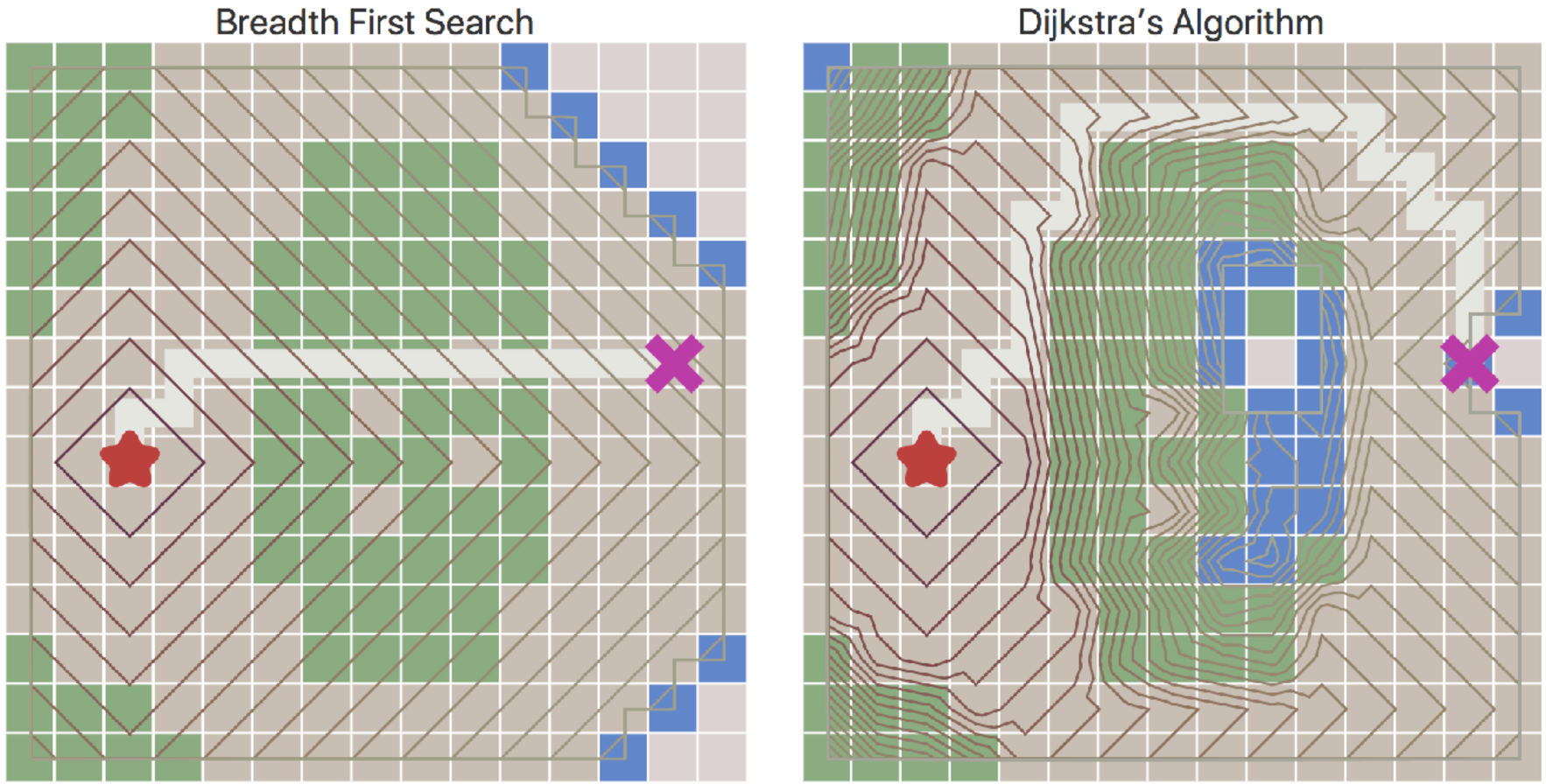


Dijkstra's Algorithm





Dijkstra's Algorithm



Heuristic Search

```
def heuristic(a, b):
   # Manhattan distance on a square grid
   return abs(a.x - b.x) + abs(a.y - b.y)
```

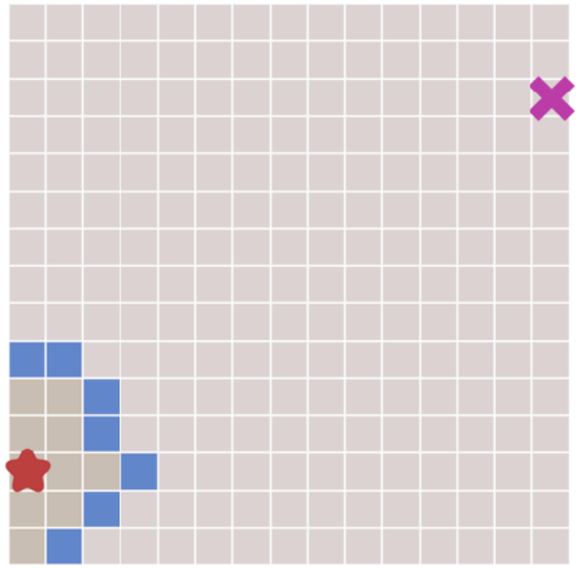
Heuristic Search

```
def heuristic(a, b):
    # Manhattan distance on a square grid
    return abs(a.x - b.x) + abs(a.y - b.y)
```

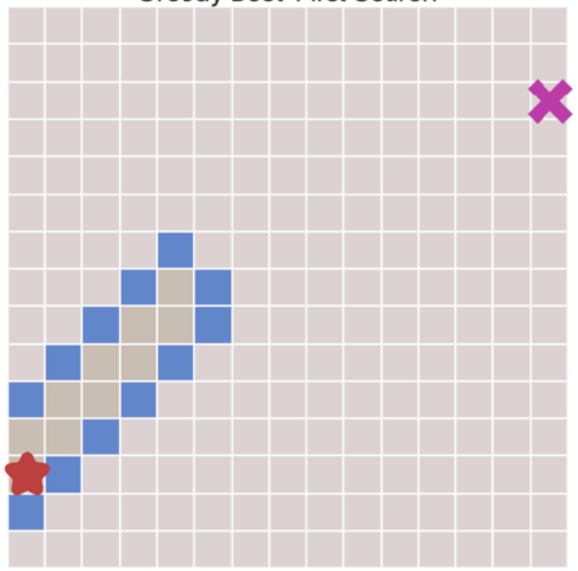
```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
came_from[start] = None

while not frontier.empty():
    current = frontier.get()

    if current == goal:
        break

    for next in graph.neighbors(current):
        if next not in came_from:
            priority = heuristic(goal, next)
            frontier.put(next, priority)
            came_from[next] = current
```

Heuristic Search

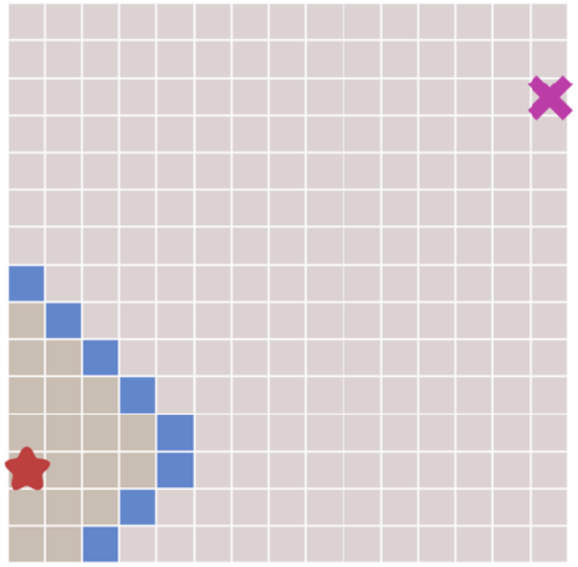
Breadth First Search



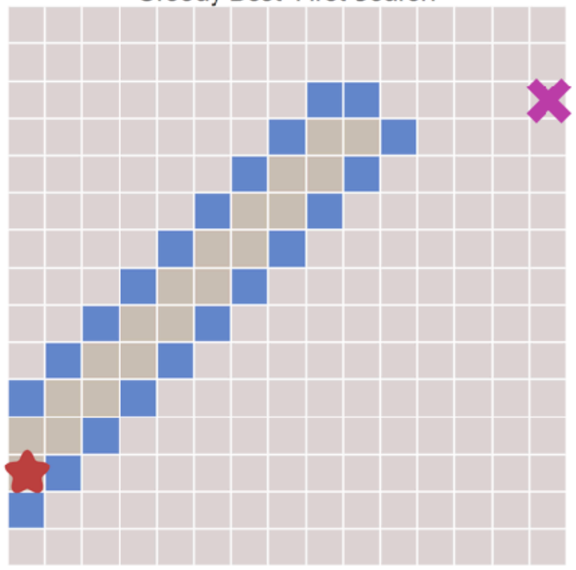
Greedy Best-First Search



Breadth First Search

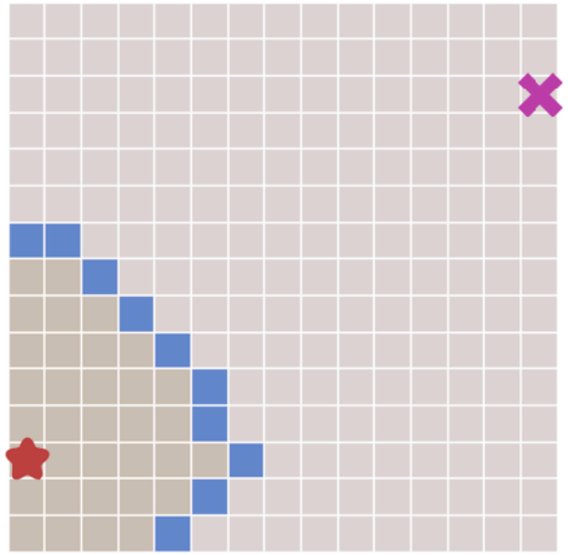


Greedy Best-First Search

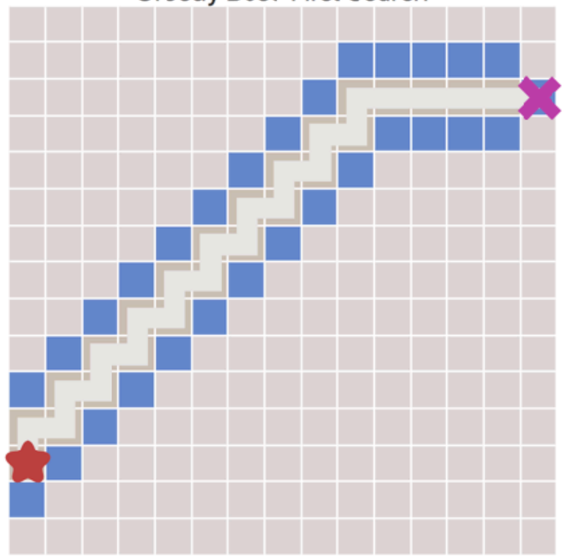


Heuristic Search

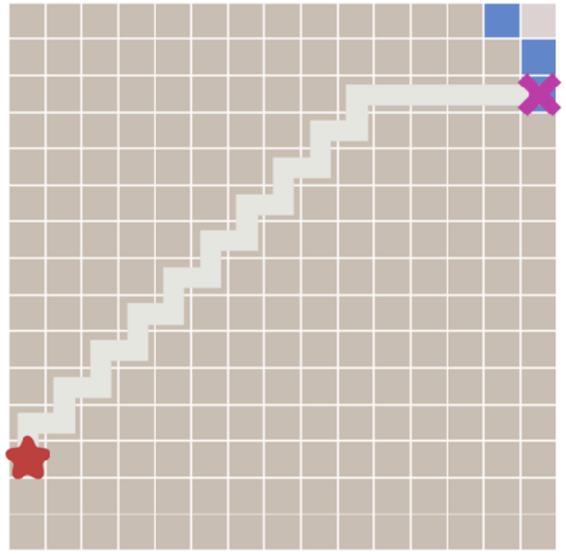
Breadth First Search



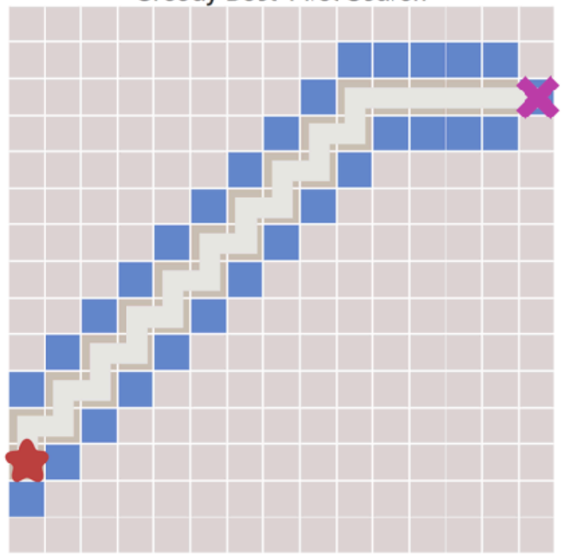
Greedy Best-First Search



Breadth First Search

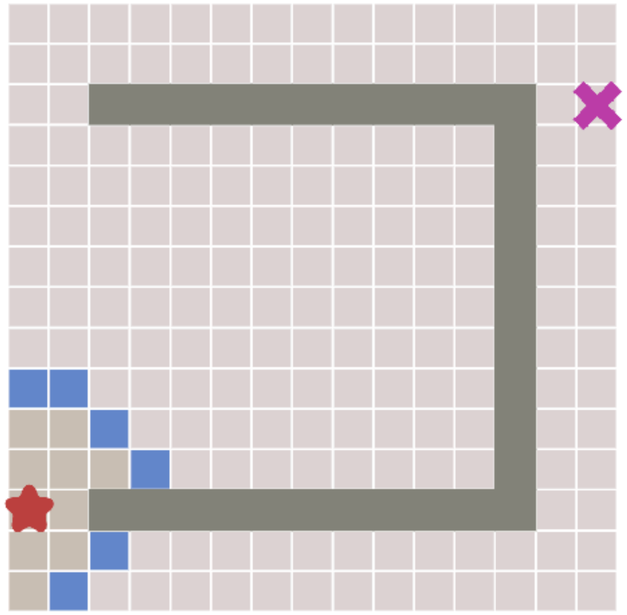


Greedy Best-First Search

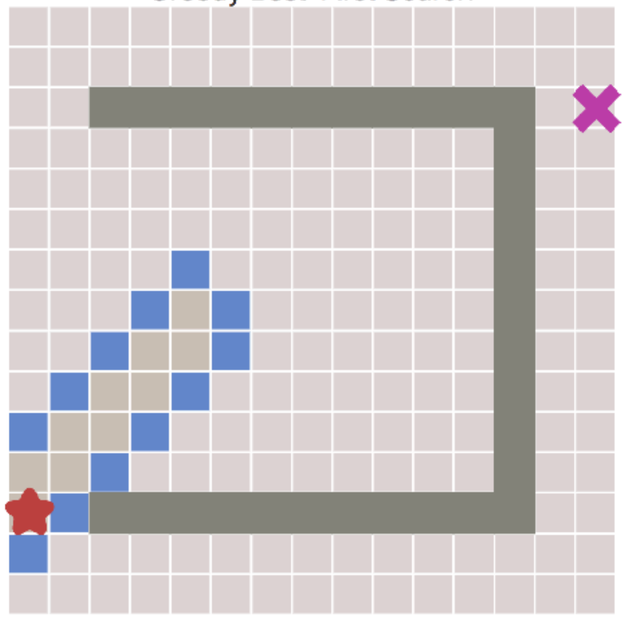


More Complex Maps

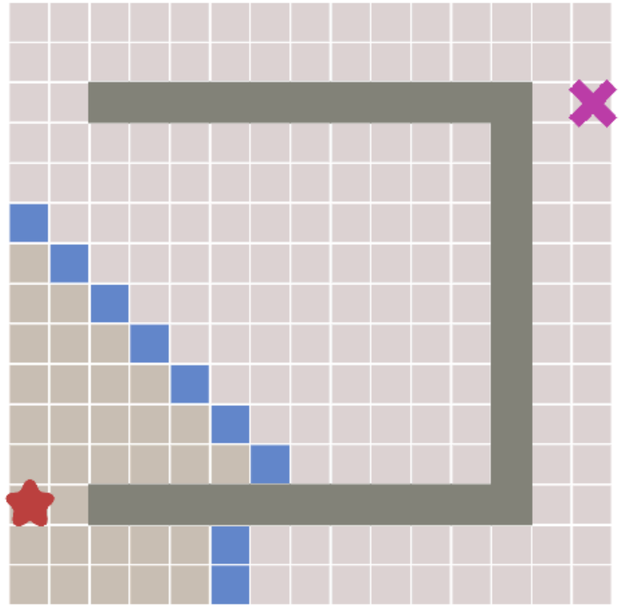
Breadth First Search



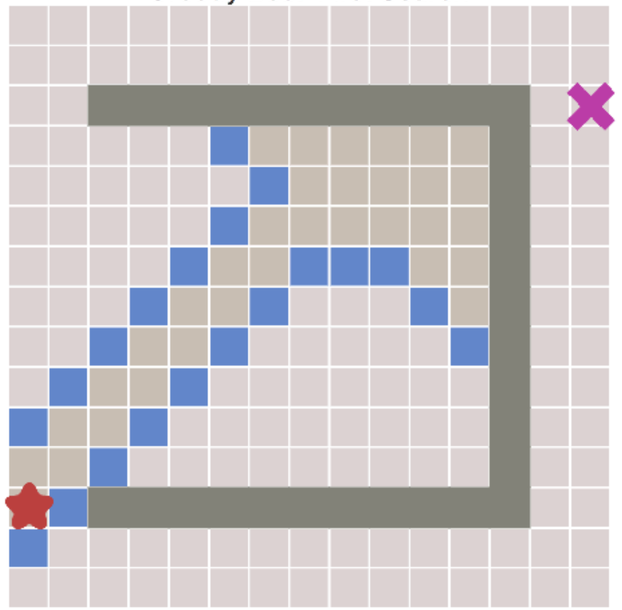
Greedy Best-First Search



Breadth First Search

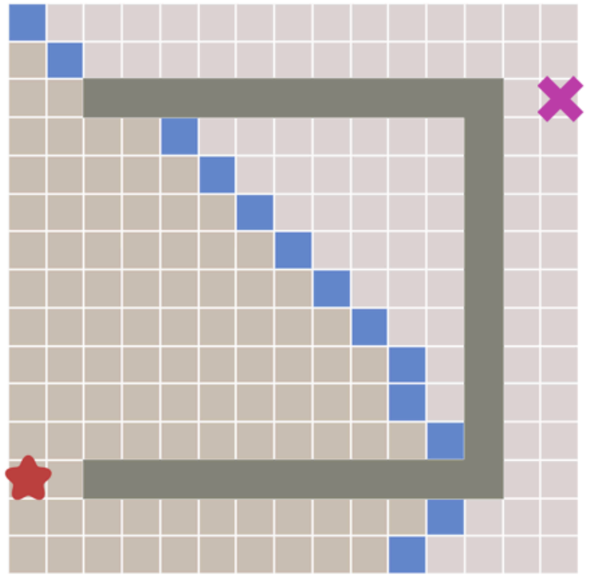


Greedy Best-First Search

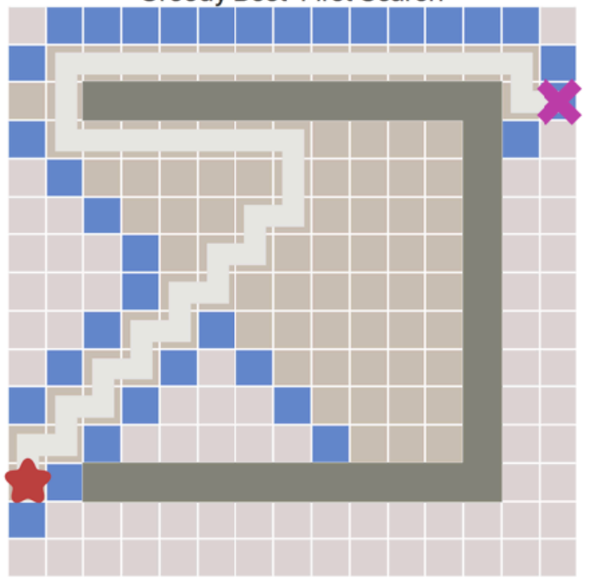


More Complex Maps

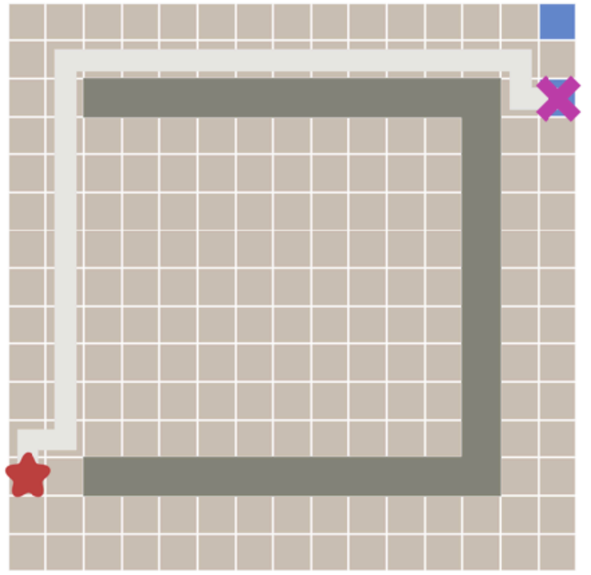
Breadth First Search



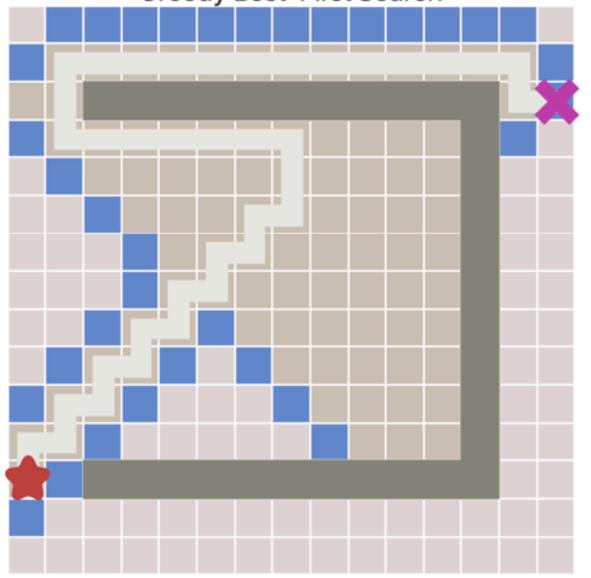
Greedy Best-First Search



Breadth First Search



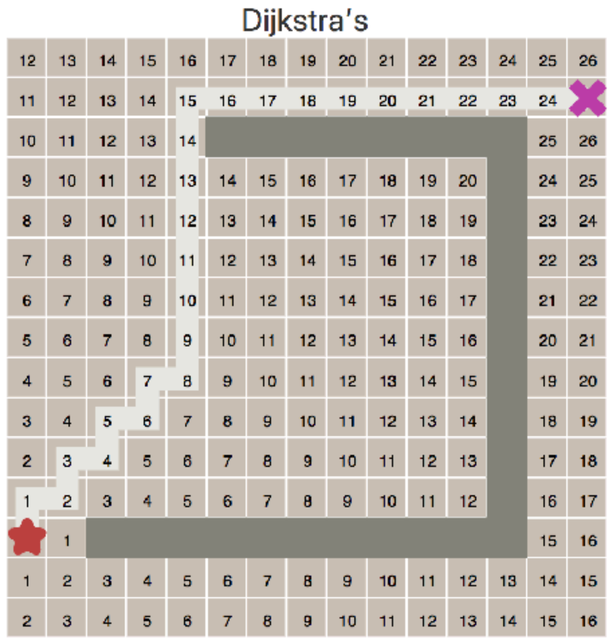
Greedy Best-First Search

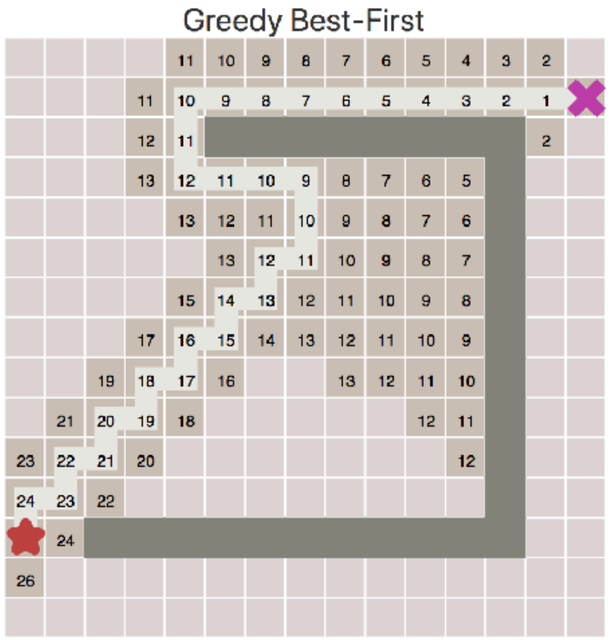


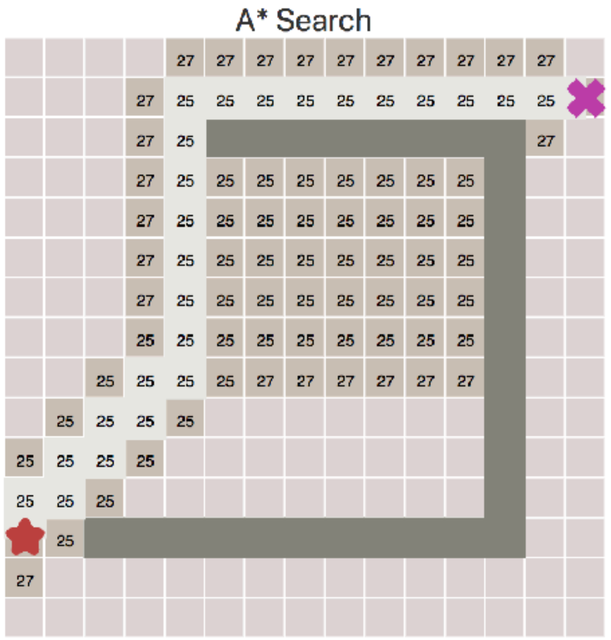
A* Algorithm

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost + heuristic(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```

Comparison

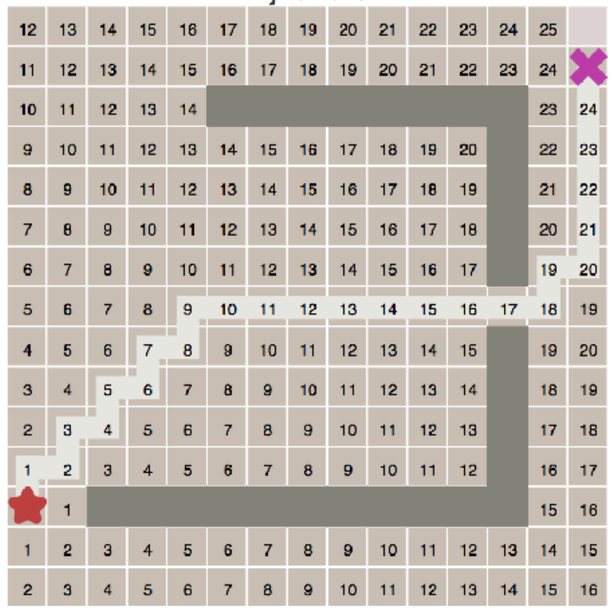




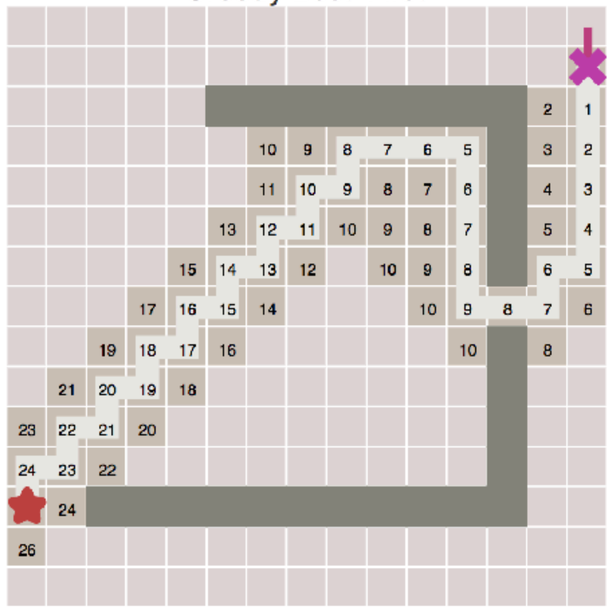


Comparison

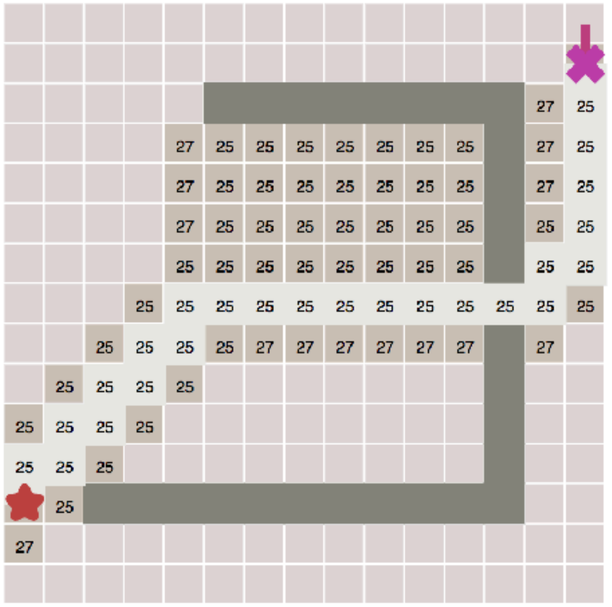
Dijkstra's



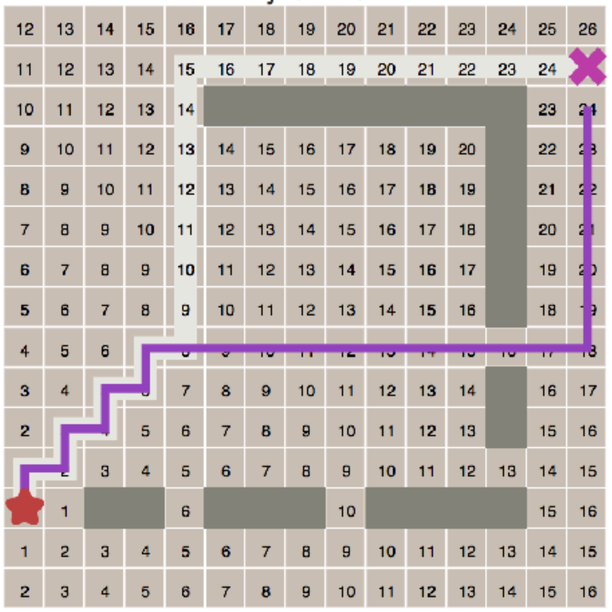
Greedy Best-First



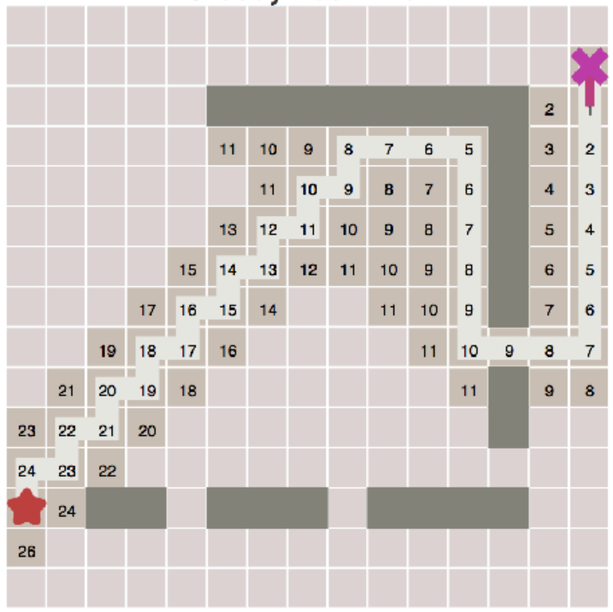
A* Search



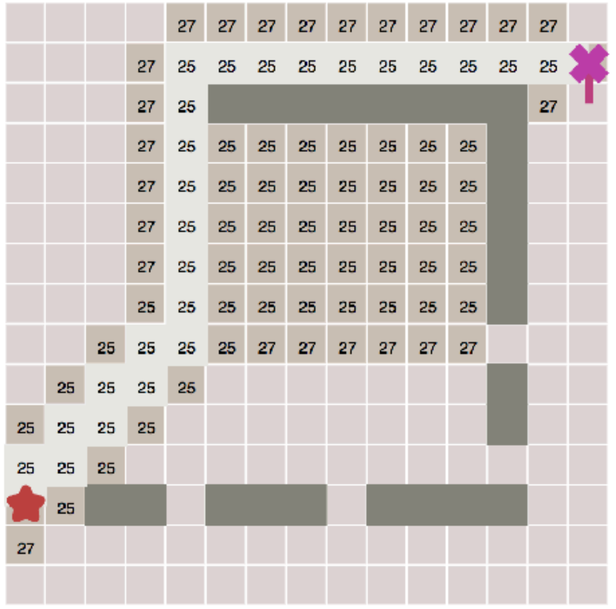
Dijkstra's



Greedy Best-First



A* Search



Code Implementation

Breadth First Search

1. **Graph:** Data structure to tell me the neighbors for each location. A weighted graph can also tell the cost of moving along the edge.

2. **Locations:** A value (int, string, tuple, etc.) that labels locations in the graph

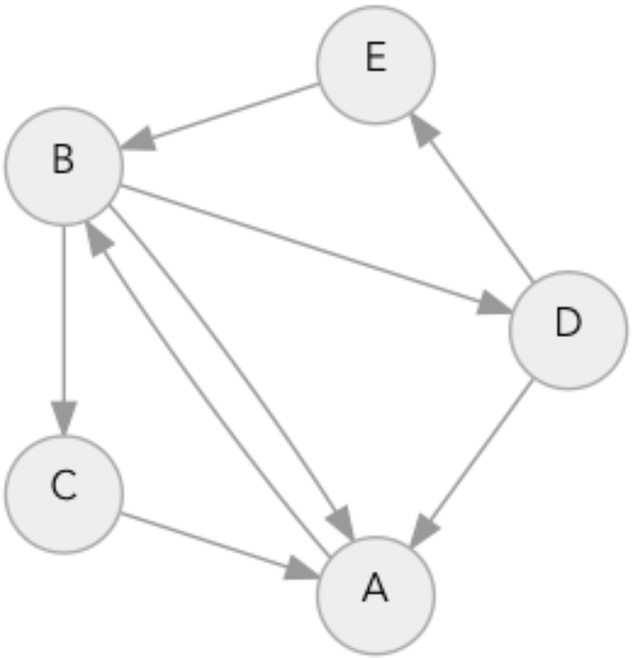
3. **Search:** Algorithm that takes a graph, starting location, and goal location, and calculates some useful information for some of all locations.

4. **Queue:** A data structure used by the search algorithm to decide the ordering which to process the locations.

Graph

```
1  class SimpleGraph:
2      def __init__(self):
3          self.edges = {}
4
5      def neighbors(self, id):
6          return self.edges[id]
7
```

```python
class SimpleGraph:
    def __init__(self):
        self.edges = {}

    def neighbors(self, id):
        return self.edges[id]

```



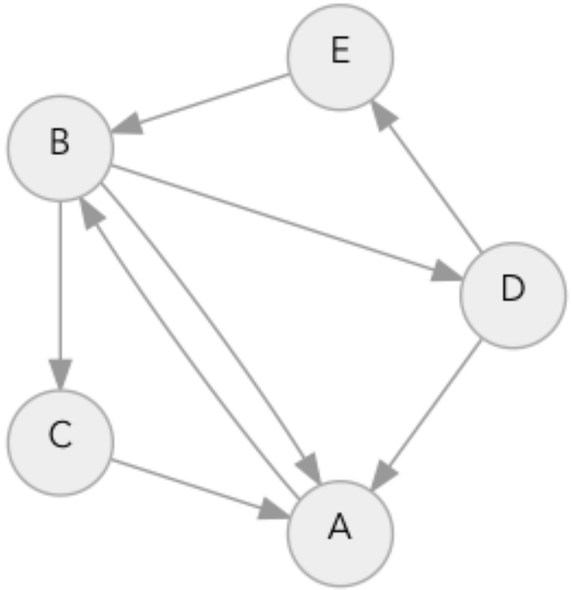

```python
example_graph = SimpleGraph()
example_graph.edges = {'A': ['B'], 'B': ['A', 'C',
    'D'], 'C': ['A'], 'D': ['E', 'A'], 'E': ['B']}

```

Queue

```
14  class Queue:
15      def __init__(self):
16          self.elements = collections.deque()
17
18      def empty(self):
19          return len(self.elements) == 0
20
21      def put(self, x):
22          self.elements.append(x)
23
24      def get(self):
25          return self.elements.popleft()
```

Breadth First Search

```
28  def breadth_first_search_1(graph, start):
29      #print out what we find
30      frontier = Queue()
31      frontier.put(start)
32      visited = {}
33      visited[start] = True
34
35      while not frontier.empty():
36          current = frontier.get()
37          print("Visiting %r" % current)
38          for next in graph.neighbors(current):
39              if next not in visited:
40                  frontier.put(next)
41                  visited[next] = True
42
43  breadth_first_search_1(example_graph, 'A')
```

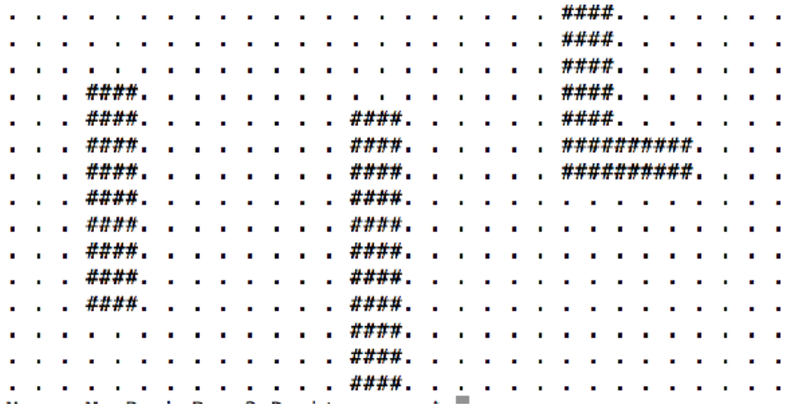



```
Visiting 'A'
Visiting 'B'
Visiting 'C'
Visiting 'D'
Visiting 'E'
```

Grids as Graphs

```python
class SquareGrid:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.walls = []

    def in_bounds(self, id):
        (x,y) = id
        return 0 <= x < self.width and 0 <= y < self.height

    def passable(self, id):
        return id not in self.walls

    def neighbors(self, id):
        (x,y) = id
        results = [(x+1, y), (x, y-1), (x-1, y), (x, y+1)]
        if (x+y)%2 == 0: results.reverse() #aesthetics
        results = filter(self.in_bounds, results)
        results = filter(self.passable, results)
        return results
```

```python
g = SquareGrid(30, 15)
g.walls = DIAGRAM1_WALLS
draw_grid(g)
```



Grids as Graphs

```
75  def breadth_first_search_2(graph, start):
76      # return "came_from"
77      frontier = Queue()
78      frontier.put(start)
79      came_from = {}
80      came_from[start] = None
81
82      while not frontier.empty():
83          current = frontier.get()
84          for next in graph.neighbors(current):
85              if next not in came_from:
86                  frontier.put(next)
87                  came_from[next] = current
88
89      return came_from
90
91  g = SquareGrid(30, 15)
92  g.walls = DIAGRAM1_WALLS
93
94  parents = breadth_first_search_2(g, (8,7))
95  draw_grid(g, width = 2, point_to = parents, start = (8,7))
96
```

Stop Condition

```
75  def breadth_first_search_3(graph, start, goal):
76      frontier = Queue()
77      frontier.put(start)
78      came_from = {}
79      came_from[start] = None
80
81      while not frontier.empty():
82          current = frontier.get()
83
84          if current == goal:
85              break
86
87          for next in graph.neighbors(current):
88              if next not in came_from:
89                  frontier.put(next)
90                  came_from[next] = current
91
92      return came_from
93
94  g = SquareGrid(30, 15)
95  g.walls = DIAGRAM1_WALLS
96
97  parents = breadth_first_search_3(g, (8, 7), (17, 2))
98  draw_grid(g, width=2, point_to=parents, start=(8, 7), goal=(17, 2))
99
```

Dijkstra's Algorithm

1. Graph needs to know cost of movement
2. Queue needs to return nodes in a different order
3. Search needs to keep track of costs from the graph and give them to the queue

Graph with Weights

1. Graph needs to know cost of movement
2. Queue needs to return nodes in a different order
3. Search needs to keep track of costs from the graph and give them to the queue

```
54 class GridWithWeights(SquareGrid):
55     def __init__(self, width, height):
56         super().__init__(width, height)
57         self.weights = {}
58
59     def cost(self, from_node, to_node):
60         return self.weights.get(to_node, 1)
61
```

Queue with Priorities

1. Graph needs to know cost of movement
2. Queue needs to return nodes in a different order
3. Search needs to keep track of costs from the graph and give them to the queue

```
73  import heapq
74
75  class PriorityQueue:
76      def __init__(self):
77          self.elements = []
78
79      def empty(self):
80          return len(self.elements) == 0
81
82      def put(self, item, priority):
83          heapq.heappush(self.elements, (priority, item))
84
85      def get(self):
86          return heapq.heappop(self.elements)[1]
87
```

Search

1. Graph needs to know cost of movement
2. Queue needs to return nodes in a different order
3. Search needs to keep track of costs from the graph and give them to the queue

```
88  def dijkstra_search(graph, start, goal):
89      frontier = PriorityQueue()
90      frontier.put(start, 0)
91      came_from = {}
92      cost_so_far = {}
93      came_from[start] = None
94      cost_so_far[start] = 0
95
96      while not frontier.empty():
97          current = frontier.get()
98
99          if current == goal:
100             break
101
102         for next in graph.neighbors(current):
103             new_cost = cost_so_far[current] + graph.cost(current, next)
104             if next not in cost_so_far or new_cost < cost_so_far[next]:
105                 cost_so_far[next] = new_cost
106                 priority = new_cost
107                 frontier.put(next, priority)
108                 came_from[next] = current
109
110     return came_from, cost_so_far
```

Search

1. Graph needs to know cost of movement
2. Queue needs to return nodes in a different order
3. Search needs to keep track of costs from the graph and give them to the queue

```
113 def reconstruct_path(came_from, start, goal):
114     current = goal
115     path = []
116     while current != start:
117         path.append(current)
118         current = came_from[current]
119     path.append(start) # optional
120     path.reverse() # optional
121     return path
```

Dijkstra's Algorithm

```
came_from, cost_so_far = dijkstra_search(diagram4, (1, 4), (7, 8))
draw_grid(diagram4, width=3, point_to=came_from, start=(1, 4), goal=(7, 8))
print()
draw_grid(diagram4, width=3, number=cost_so_far, start=(1, 4), goal=(7, 8))
print()
draw_grid(diagram4, width=3, path=reconstruct_path(came_from, start=(1, 4), goal=(7, 8)))
```

```
v  v  <  <  <  <  <  <  <  <
v  v  <  <  <  ^  ^  <  <  <
v  v  <  <  <  <  ^  ^  <  <
v  v  <  <  <  <  <  ^  ^  .
>  A  <  <  <  <  .  .  .  .
^  ^  <  <  <  <  .  .  .  .
^  ^  <  <  <  <  <  .  .  .
^  #########^  <  v  .  .  .
^  #########v  v  v  Z  .  .
^  <  <  <  <  <  <  <  <  .

5  4  5  6  7  8  9  10 11 12
4  3  4  5  10 13 10 11 12 13
3  2  3  4  9  14 15 12 13 14
2  1  2  3  8  13 18 17 14 .
1  A  1  6  11 16 .  .  .  .
2  1  2  7  12 17 .  .  .  .
3  2  3  4  9  14 19 .  .  .
4  #########14 19 18 .  .  .
5  #########15 16 13 Z  .  .
6  7  8  9  10 11 12 13 14 .

.  .  .  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .  .  .
@  @  .  .  .  .  .  .  .  .
@  .  .  .  .  .  .  .  .  .
@  .  .  .  .  .  .  .  .  .
@  #########.  .  .  .  .  .
@  #########.  .  @  @  .  .
@  @  @  @  @  @  @  .  .  .
```

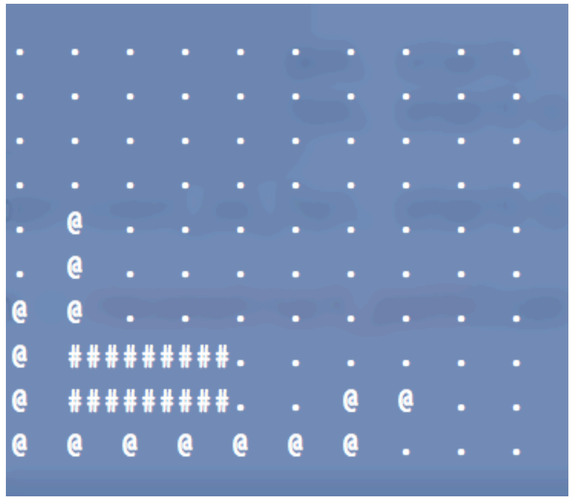
A* Algorithm

```
def heuristic(a, b):
    (x1, y1) = a
    (x2, y2) = b
    return abs(x1 - x2) + abs(y1 - y2)

def a_star_search(graph, start, goal):
    frontier = PriorityQueue()
    frontier.put(start, 0)
    came_from = {}
    cost_so_far = {}
    came_from[start] = None
    cost_so_far[start] = 0

    while not frontier.empty():
        current = frontier.get()

        if current == goal:
            break

        for next in graph.neighbors(current):
            new_cost = cost_so_far[current] + graph.cost(current, next)
            if next not in cost_so_far or new_cost < cost_so_far[next]:
                cost_so_far[next] = new_cost
                priority = new_cost + heuristic(goal, next)
                frontier.put(next, priority)
                came_from[next] = current

    return came_from, cost_so_far
```

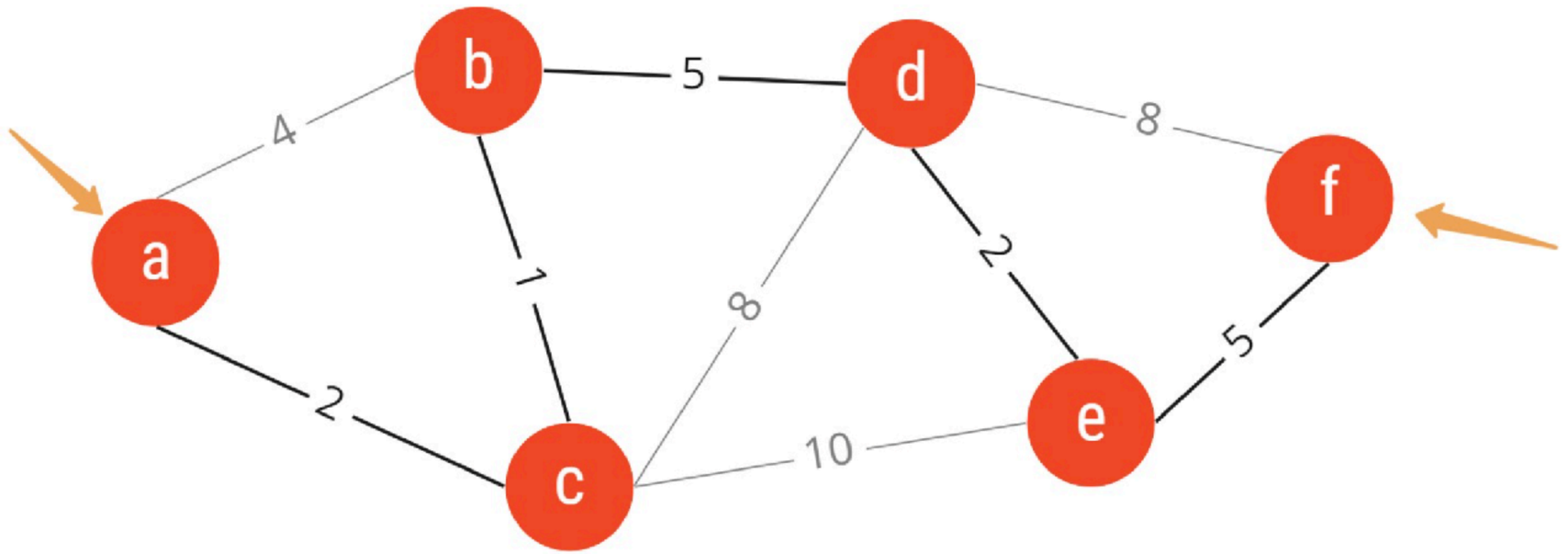
A* Algorithm

```
start, goal = (1, 4), (7, 8)
came_from, cost_so_far = a_star_search(diagram4, start, goal)
draw_grid(diagram4, width=3, point_to=came_from, start=start, goal=goal)
print()
draw_grid(diagram4, width=3, number=cost_so_far, start=start, goal=goal)
print()
draw_grid(diagram4, width=3, path = reconstruct_path(came_from, start = start, goal=goal))
print()
```

```
.  .  .  .  .  .  .  .  .  .
.  v  v  v  .  .  .  .  .  .
v  v  v  v  <  .  .  .  .  .
v  v  v  <  <  .  .  .  .  .
>  A  <  <  <  .  .  .  .  .
>  ^  <  <  <  .  .  .  .  .
>  ^  <  <  <  <  .  .  .  .
^  #########^  .  v  .  .  .
^  #########v  v  v  Z  .  .
^  <  <  <  <  <  <  <  .  .

.  .  .  .  .  .  .  .  .  .
.  3  4  5  .  .  .  .  .  .
3  2  3  4  9  .  .  .  .  .
2  1  2  3  8  .  .  .  .  .
1  A  1  6  11 .  .  .  .  .
2  1  2  7  12 .  .  .  .  .
3  2  3  4  9  14 .  .  .  .
4  #########14 .  18 .  .  .
5  #########15 16 13 Z  .  .
6  7  8  9  10 11 12 13 .  .
```



NetworkX: Shortest Path



2 + 1 + 5 + 2 + 5 = 15

Dijkstra's Algorithm

```python
import networkx as nx
import matplotlib.pyplot as plt

edges = [(1,2, {'weight':4}),
         (1,3,{'weight':2}),
         (2,3,{'weight':1}),
         (2,4, {'weight':5}),
         (3,4, {'weight':8}),
         (3,5, {'weight':10}),
         (4,5,{'weight':2}),
         (4,6,{'weight':8}),
         (5,6,{'weight':5})]
edge_labels = {(1,2):4, (1,3):2, (2,3):1, (2,4):5, (3,4):8, (3,5):10, (4,5):2, (4,6):8, (5,6):5}


G = nx.Graph()
for i in range(1,7):
    G.add_node(i)
G.add_edges_from(edges)

pos = nx.planar_layout(G)
nx.draw(G, with_labels = True)
plt.show()

# This will give us all the shortest paths from node 1 using the weights from the edges.
p1 = nx.shortest_path(G, source=1, weight='weight')

# This will give us the shortest path from node 1 to node 6.
p1to6 = nx.shortest_path(G, source=1, target=6, weight='weight')

# This will give us the length of the shortest path from node 1 to node 6.
length = nx.shortest_path_length(G, source=1, target=6, weight='weight')

print("All shortest paths from 1: " + str(p1))
print("Shortest path from 1 to 6: " + str(p1to6))
print("Length of the shortest path: " + str(length))

```

A*

```python
import networkx as nx
import matplotlib.pyplot as plt

def dist(a, b):
    (x1, y1) = a
    (x2, y2) = b
    return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5

G = nx.grid_graph(dim=[3, 3])  # nodes are two-tuples (x,y)
nx.set_edge_attributes(G, {e: e[1][0] * 2 for e in G.edges()}, "cost")
path = nx.astar_path(G, (0, 0), (2, 2), heuristic=dist, weight="cost")
length = nx.astar_path_length(G, (0, 0), (2, 2), heuristic=dist, weight="cost")
print("Path: " + str(path))
print("Path length: " + str(length))

pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels = True, node_color="#f86e00")
edge_labels = nx.get_edge_attributes(G, "cost")
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
plt.show()
```

RRT (Rapidly Exploring Random Trees)

https://github.com/mouad-boumediene/python-visualization-of-the-RRT-algorithm-with-pygame

https://youtu.be/TzfNzqjJ2VQ?si=Cf5dD4tOiO8xZ77D