

State space

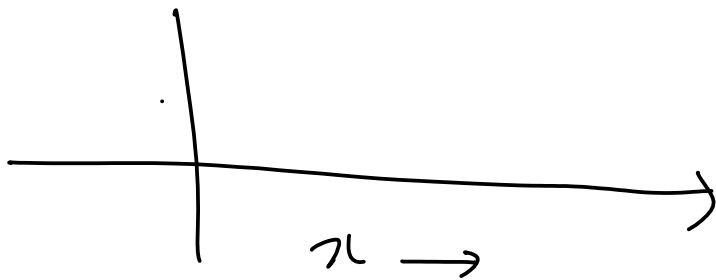
↳ Set of all states

What is a state

Example,

$S_t = \begin{bmatrix} x_t \\ y_t \end{bmatrix}$ (x, y) coordinate of the robot in the maze

State contains all information needed to plan the future course of action without having to look in the past.

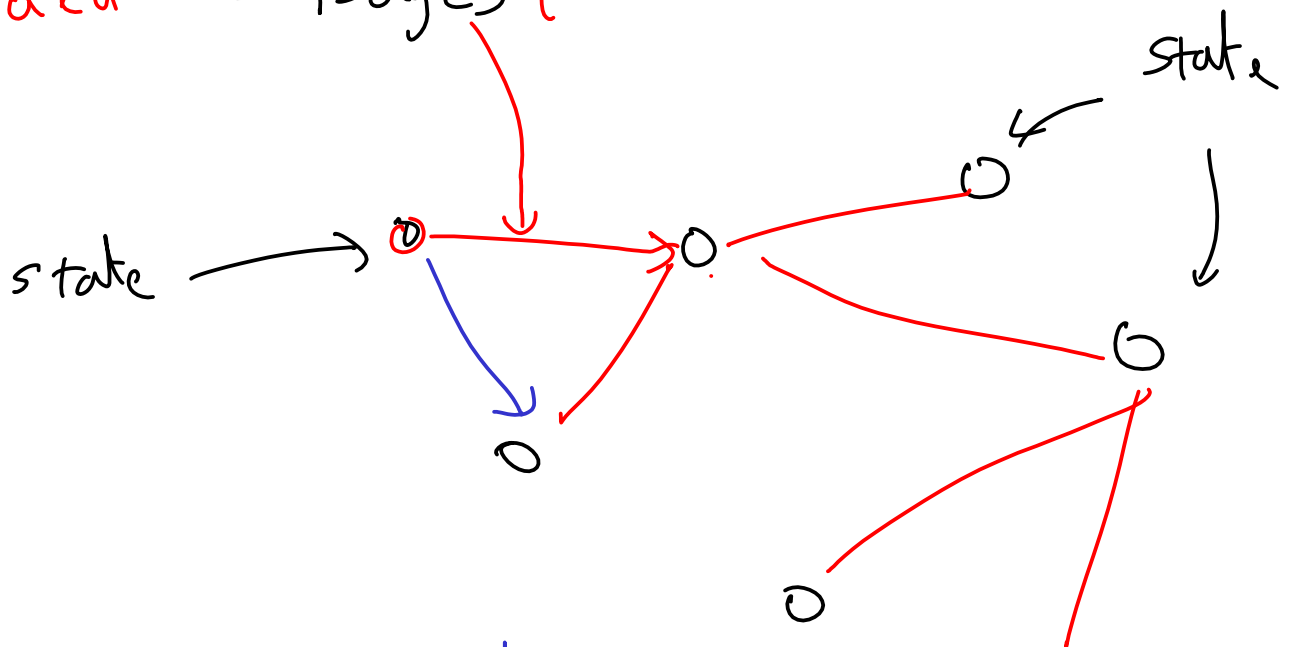


$$\underline{S_t} = \begin{bmatrix} x_t \\ v_t \end{bmatrix}$$

$$\begin{aligned} v &= \dot{x} \\ \frac{F}{m} = a &= \ddot{x} \end{aligned}$$

$$x_{t+\Delta t} = x_t + v_t \Delta t + \frac{1}{2} a \Delta t^2$$

Graph \rightarrow Vertices \equiv States
 Directed \rightarrow Edges (Directions)



Action space at a States \circ $A(s_t)$

set of actions you
 can take at a given
 state $\underline{s_t}$

Denote
 sets
 with
 curly capital
 letters

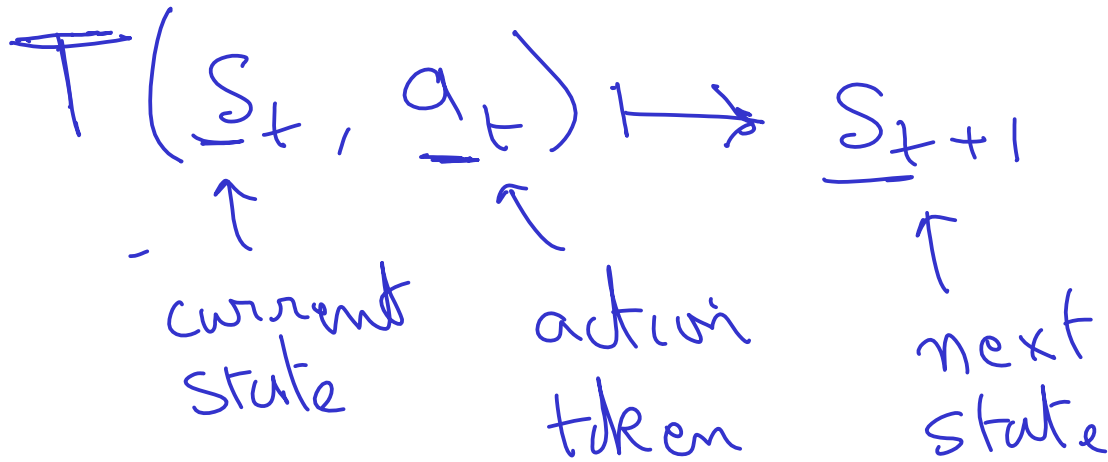
Actions?

Example: maze

$A(s_t) = \{ \text{Left, Right, Top, Bottom} \}$

(1) States (2) Actions

(3) Cost function (4) Transition function



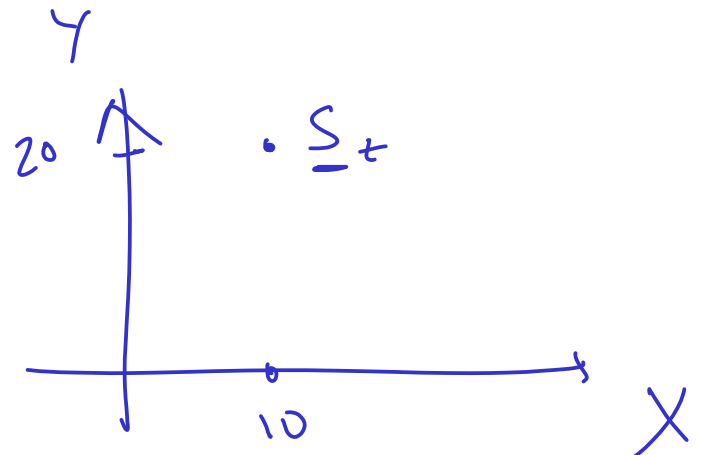
Example: maze

$$\underline{s}_t = \begin{bmatrix} 10 & 20 \end{bmatrix}$$

x y

$$\underline{s}_{t+1} = \begin{bmatrix} 9 \\ 20 \end{bmatrix}$$

$\underline{a}_t = \text{Left by 1 step}$



def transition_fn($\underline{s}_t, \underline{a}_t$):

$$\underline{s}_{t+1} = \underline{s}_t + \underline{a}_t$$

return \underline{s}_{t+1}

$$\underline{a}_t = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

Left

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

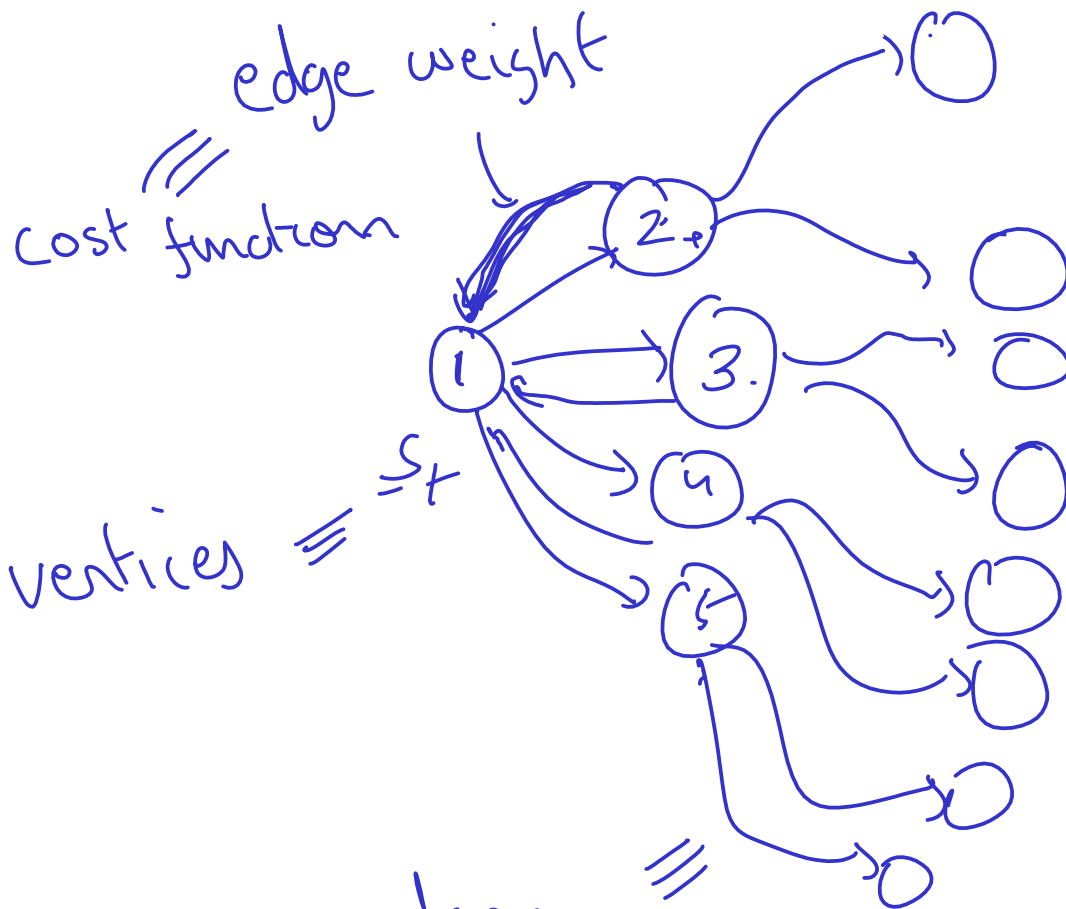
Right

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Top

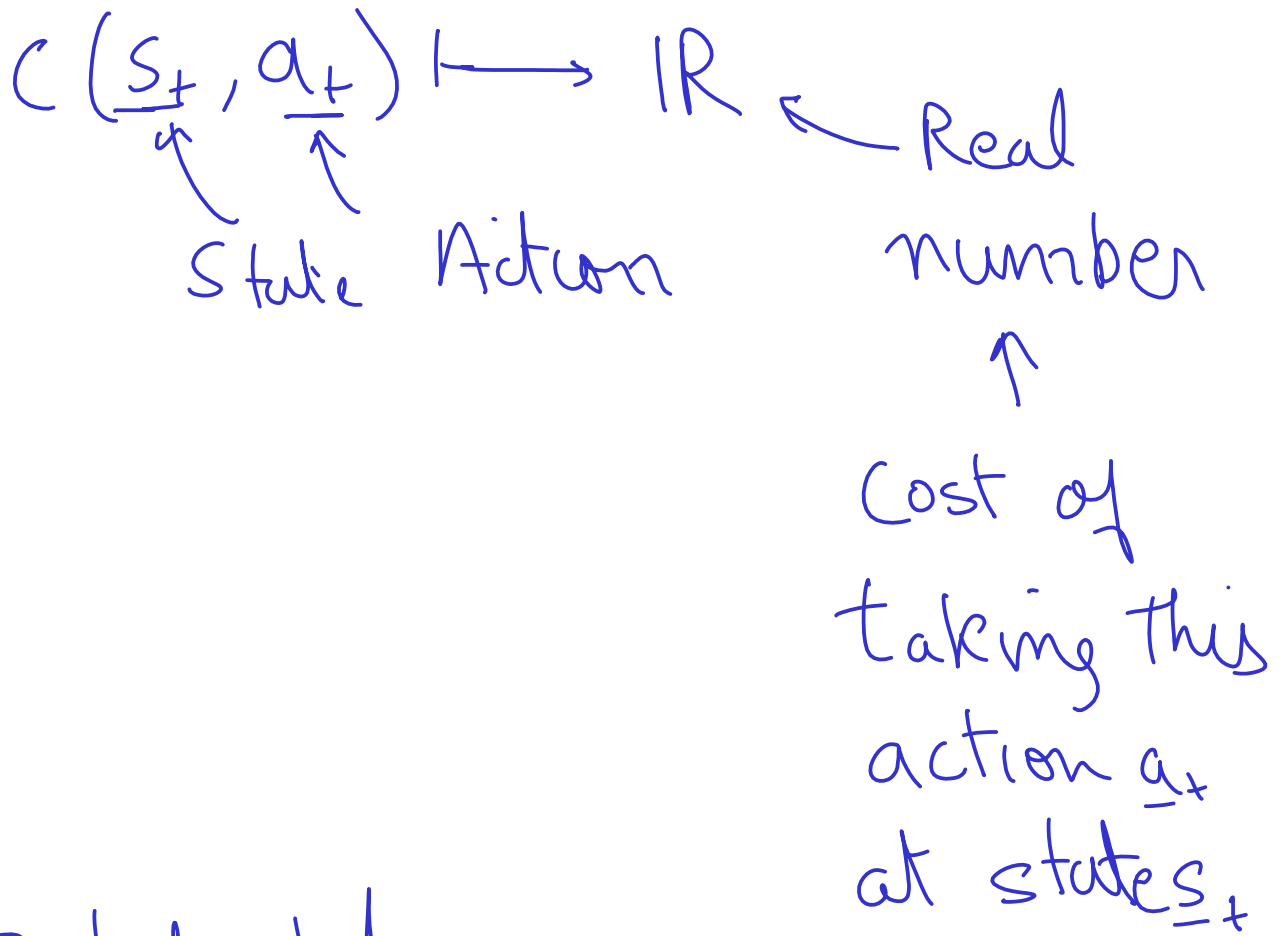
$$\begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

Bottom



\equiv action space at a state + Transition function

④ Cost function



⑥ Initial state \underline{s}_I

⑦ Goal state \underline{s}_G

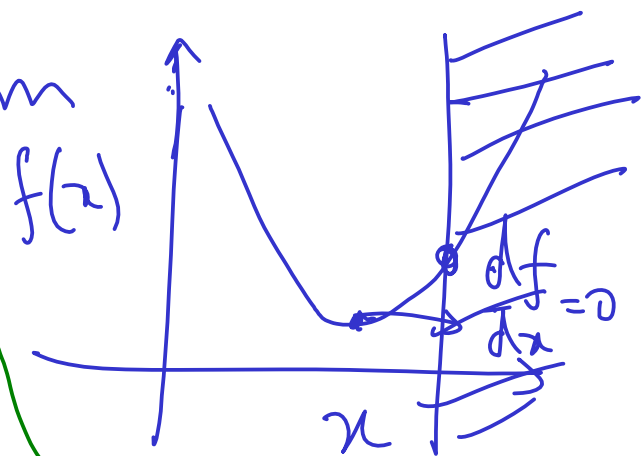
Planning problem

Find a sequence of actions to take that takes ~~us~~ ^{the robot} from \underline{s}_I to \underline{s}_G with minimum cost.

Planning problem as
an optimization problem

$$\text{minimize } \sum_{t=0}^T c(\underline{s}_t, \underline{a}_t)$$
$$\{\underline{a}_{-t}\}_{t=0}^T$$

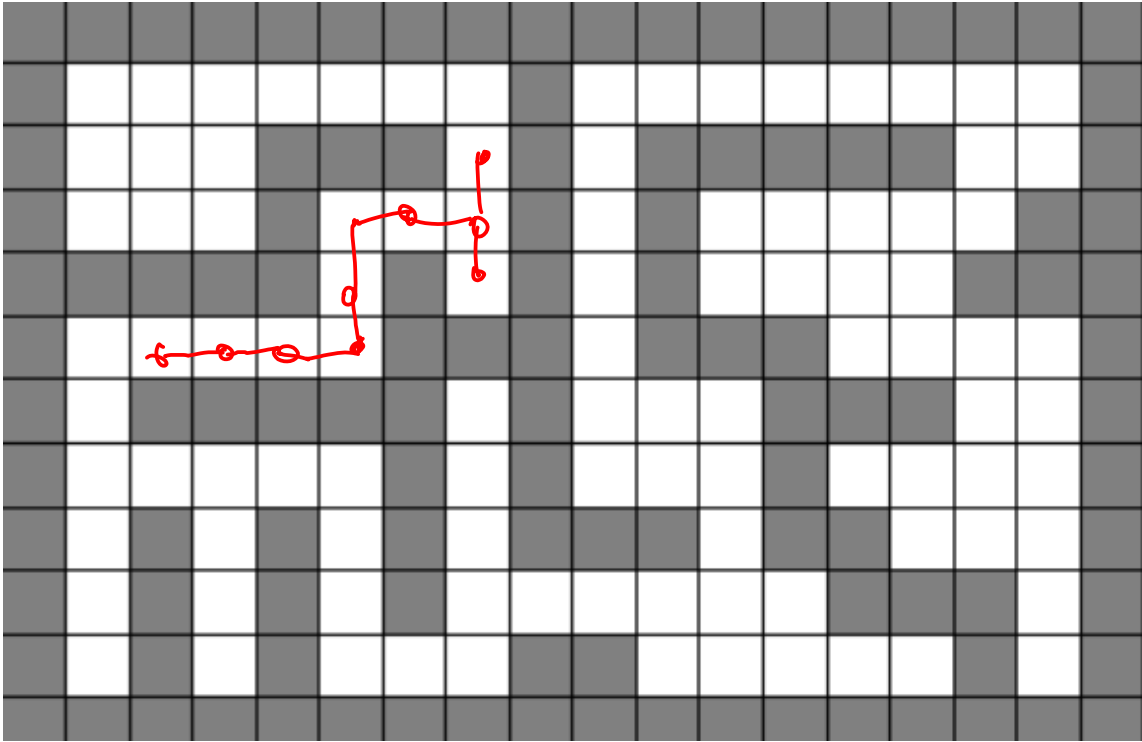
$$\text{such that } \underline{s}_{t+1} = T(\underline{s}_t, \underline{a}_t)$$



The category of approaches is

called Dynamic programming

(intuitively
similar to
mathematical
induction)



Graph Data structures

- ① Adjacency list
- ② Adjacency matrix

Planning (Chapter 2 from Lavalle book)

Abstraction of a planning problem

1. State space $\mathbf{s} \in \mathcal{S}$. For example, 2D coordinate of a grid $\mathbf{s} = (x, y)$.
2. Action space per state $\mathbf{u} \in \mathcal{U}(\mathbf{s})$. For example, up, down, left right movement can be encoded as $\mathcal{U}(\mathbf{s}_t) = \{(0, -1), (0, 1), (1, 0), (-1, 0)\}$.
3. State transition function $\mathbf{s}_{t+1} = f(\mathbf{s}_t, \mathbf{u}_t)$. For example, the up-down-left-right action can be combined as addition to get the next state
$$\mathbf{s}_{t+1} = \mathbf{s}_t + \mathbf{u}_t.$$
4. Initial State $\mathbf{s}_I \in \mathcal{S}$
5. Goal states $\mathbf{s}_G \subseteq \mathcal{S}$

A Graph

A graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ is defined by a set of vertices \mathcal{V} and a set of edges \mathcal{E} such that each edge $e \in \mathcal{E}$ is formed by a pair of start and end vertices $e = (v_s, v_e), v_s \in \mathcal{V}, v_e \in \mathcal{V}$. The first vertex is called the start of the edge $v_s = \text{start}(e)$ and second vertex is called the end $v_e = \text{end}(e)$.

A discrete planning problem can be converted into a graph by defining

1. Vertices as the state space $\mathcal{V} = \mathcal{S}$.
2. The action space at each state as the edges connected to that vertex/state,
$$\mathcal{U}(\mathbf{s}_t) = \{(\mathbf{s}_t, \mathbf{s}_j) \mid (\mathbf{s}_t, \mathbf{s}_j) \in \mathcal{E}\}.$$
3. State transition function is the other end of the edge,
$$\mathbf{s}_{t+1} = f(\mathbf{s}_t, \mathbf{u}_t) = \text{end}(\mathbf{u}_t), \text{ where } \mathbf{s}_t = \text{start}(\mathbf{u}_t).$$

Representations of Graphs

Undirected graph

```
In [1]: # Programmatically you can represent a adjacency list as python lists
# Python lists are not linked lists, they are arrays under the hood.
G_adjacency_list = {
    1 : [2, 5],
    2 : [1, 5, 3, 4],
    3 : [2, 4],
    4 : [2, 5, 3],
    5 : [4, 1, 2]
}

# Prefer to represent a matrix in python either as a list of lists or a numpy
import numpy as np
G_adjacency_matrix = np.array([
    [0, 1, 0, 0, 1],
    [1, 0, 1, 1, 1],
    [0, 1, 0, 1, 0],
    [0, 1, 1, 0, 1],
    [1, 1, 0, 1, 0]
])

# Edge list is another possible representation
G_edge_list = [
    (1, 2), (1, 5),
    (2, 1), (2, 5), (2, 3), (2, 4),
    (3, 2), (3, 4),
    (4, 2), (4, 5), (4, 3),
    (5, 4), (5, 1), (5, 2)
]
```

Directed graph representation

```
In [2]: # Programmatically you can represent a adjacency list as python lists
# Python lists are not linked lists, they are arrays under the hood.
G_adjacency_list = {
    1 : [2, 4],
    2 : [5],
    3 : [6, 5],
    4 : [2],
    5 : [4],
    6 : [6]
}

# Prefer to represent a matrix in python either as a list of lists or a numpy
import numpy as np
G_adjacency_matrix = np.array([
    [0, 1, 0, 1, 0, 0],
```

```

    [0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 1, 1],
    [0, 1, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 1]
])

# Edge list is another possible representation
G_edge_list = [
    (1, 2), (1, 4),
    (2, 5),
    (3, 6), (3, 5),
    (4, 2),
    (5, 6)
]

```

In [3]: *# Exercise 1*

```

# Write a function that converts a graph in adjacency list format to adjacency matrix
def adjacency_list_to_matrix(G_adj_list):
    G_adj_mat = None # TODO: Write code to convert to adj_mat
    return G_adj_mat

def adjacency_matrix_to_list(G_adj_mat):
    G_adj_list = None # TODO: Write code to convert to adj_mat
    return G_adj_list

# Use the above graphs to test
print(adjacency_list_to_matrix(G_adjacency_list))
print(adjacency_matrix_to_list(G_adjacency_matrix))

```


None

None

Graph Search algorithms

1. Breadth First Search 

2. Depth First Search

 dfs.png

Breadth first search (BFS)

 bfs-states

In [4]: **from** queue **import** Queue, LifoQueue, PriorityQueue

```

graph = {
    's' : ['w', 'r'],
    'r' : ['v'],
    'w' : ['t', 'x'],
    'x' : ['y'],
    't' : ['u'],

```

```

    'u' : ['y']
}

def bfs(graph, start, debug=False):
    seen = set() # Set for seen nodes (contains both frontier and dead state)
    # Frontier is the boundary between seen and unseen (Also called the alive)
    frontier = Queue() # Frontier of unvisited nodes as FIFO
    node2dist = {start : 0} # Keep track of distances
    search_order = []
    seen.add(start)
    frontier.put(start)

    i = 0 # step number
    while not frontier.empty(): # Creating loop to visit each node
        if debug: print("%d) Q = " % i, list(frontier.queue), end='; ')
        if debug: print("dists = " , [node2dist[n] for n in frontier.queue])
        m = frontier.get() # Get the oldest addition to frontier
        search_order.append(m)

        for neighbor in graph.get(m, []):
            if neighbor not in seen:
                seen.add(neighbor)
                frontier.put(neighbor)
                node2dist[neighbor] = node2dist[m] + 1
            else:
                assert node2dist[neighbor] <= node2dist[m] + 1, 'this should not happen'
                node2dist[neighbor] = min(node2dist[neighbor], node2dist[m] + 1)

        i += 1
    if debug: print("%d) Q = " % i, list(frontier.queue))
    return search_order, node2dist

```

In [5]: `print("Following is the Breadth-First Search order")`
`print(bfs(graph, 's', debug=True))` # function calling

Following is the Breadth-First Search order

```

0) Q = ['s']; dists = [0]
1) Q = ['w', 'r']; dists = [1, 1]
2) Q = ['r', 't', 'x']; dists = [1, 2, 2]
3) Q = ['t', 'x', 'v']; dists = [2, 2, 2]
4) Q = ['x', 'v', 'u']; dists = [2, 2, 3]
5) Q = ['v', 'u', 'y']; dists = [2, 3, 3]
6) Q = ['u', 'y']; dists = [3, 3]
7) Q = ['y']; dists = [3]
8) Q = []
(['s', 'w', 'r', 't', 'x', 'v', 'u', 'y'], {'s': 0, 'w': 1, 'r': 1, 't': 2, 'x': 2, 'v': 2, 'u': 3, 'y': 3})

```

Depth first search



```
In [6]: graph = {
    's' : ['w', 'r'],
    'r' : ['v'],
    'w' : ['t', 'x'],
    'x' : ['y'],
    't' : ['u'],
    'u' : ['y']
}

def dfs(graph, start, debug=False):
    seen = set([start]) # List for seen nodes (contains both frontier and de
    # Frontier is the boundary between seen and unseen (Also called the alive
    frontier = LifoQueue() # Frontier of unvisited nodes as FIFO
    node2dist = {start : 0} # Keep track of distances
    search_order = [] # Keep track of search order
    frontier.put(start)

    i = 0 # step number
    while not frontier.empty(): # Creating loop to visit each node
        if debug: print("%d) Q = " % i, list(frontier.queue), end='; ')
        if debug: print("dists = " , [node2dist[n] for n in frontier.queue])
        m = frontier.get() # Get the oldest addition to frontier
        search_order.append(m)

        for neighbor in graph.get(m, []):
            if neighbor not in seen:
                seen.add(neighbor)
                frontier.put(neighbor)
                node2dist[neighbor] = node2dist[m] + 1
            else:
                node2dist[neighbor] = min(node2dist[neighbor], node2dist[m])
        i += 1
    if debug: print("%d) Q = " % i, list(frontier.queue))
    return search_order, node2dist
```

```
In [7]: # Driver Code
print("Following is the Depth-First Search path")
print(dfs(graph, 's', debug=True)) # function calling
```

Following is the Depth-First Search path

```
0) Q = ['s']; dists = [0]
1) Q = ['w', 'r']; dists = [1, 1]
2) Q = ['w', 'v']; dists = [1, 2]
3) Q = ['w']; dists = [1]
4) Q = ['t', 'x']; dists = [2, 2]
5) Q = ['t', 'y']; dists = [2, 3]
6) Q = ['t']; dists = [2]
7) Q = ['u']; dists = [3]
8) Q = []
(['s', 'r', 'v', 'w', 'x', 'y', 't', 'u'], {'s': 0, 'w': 1, 'r': 1, 'v': 2,
't': 2, 'x': 2, 'y': 3, 'u': 3})
```

Converting a maze search to a graph search

In [8]: *# Skip these utilities for the class*

```
def batched(iterable, n):
    "Batch data into tuples of length n. The last batch may be shorter."
    # batched('ABCDEFGG', 3) --> ABC DEF G
    if n < 1:
        raise ValueError('n must be at least one')
    it = iter(iterable)
    while batch := tuple(islice(it, n)):
        yield batch

def draw_path(self, path, visited='*'):
    new_maze_lines = [list(l) for l in self.maze_lines]
    for (r, c) in path:
        new_maze_lines[r][c] = visited
        print('\n'.join([''.join(l) for l in new_maze_lines]))
        print('\n\n\n')

def init_plots(self, reinit=False):
    if self.fig is None or reinit:
        self.fig, self.ax = plt.subplots()

def plot_maze(self):
    self.init_plots()
    replace = { ' ' : 1, '+' : 0 }
    maze_mat = np.array([[replace[c] for c in line]
                          for line in self.maze_lines])
    return [self.ax.imshow(maze_mat, cmap='gray')]

def plot_step(self, i_node):
    i, (r, c) = i_node
    return [self.ax.text(c, r, '%d' % (i+1))]

def plot_path(self, path):
    self.plot_maze()
    return [self.plot_step((i, (r,c)))
            for i, (r, c) in enumerate(path)]

def animate_search_path(maze, search_path, node2dist):
    maze.init_plots()
    return animation.FuncAnimation(maze.fig, maze.plot_step, frames=[(node2c
                                                                    for n
                                                                    init_func=maze.plot_maze, blit=True, repea
```

In [9]: **import** matplotlib.pyplot **as** plt

import numpy **as** np

maze_str = \

"""

+++++++

+ +

+ + + +++

+ + + +

+ + + +

+ + +++ +

```

+      + +
+ +++ + +
+   +
+++++++
"""

class Maze:
    def __init__(self, maze_str, freepath=' '):
        self.maze_lines = [l for l in maze_str.split("\n")
                           if len(l)]
        self.FREEPATH = freepath
        self.fig = None

    def get(self, node, default):
        (r, c) = node
        m_row = self.maze_lines[r]
        nbrs = []
        if c-1 >= 0 and m_row[c-1] == self.FREEPATH:
            nbrs.append((r, c-1))
        if c+1 < len(m_row) and m_row[c+1] == self.FREEPATH:
            nbrs.append((r, c+1))
        if r-1 >= 0 and self.maze_lines[r-1][c] == self.FREEPATH:
            nbrs.append((r-1, c))
        if r+1 < len(self.maze_lines) and self.maze_lines[r+1][c] == self.FREEPATH:
            nbrs.append((r+1, c))
        return nbrs if len(nbrs) else default
    init_plots = init_plots
    plot_maze = plot_maze
    plot_step = plot_step
    plot_path = plot_path
    animate_search_path = animate_search_path

```

```

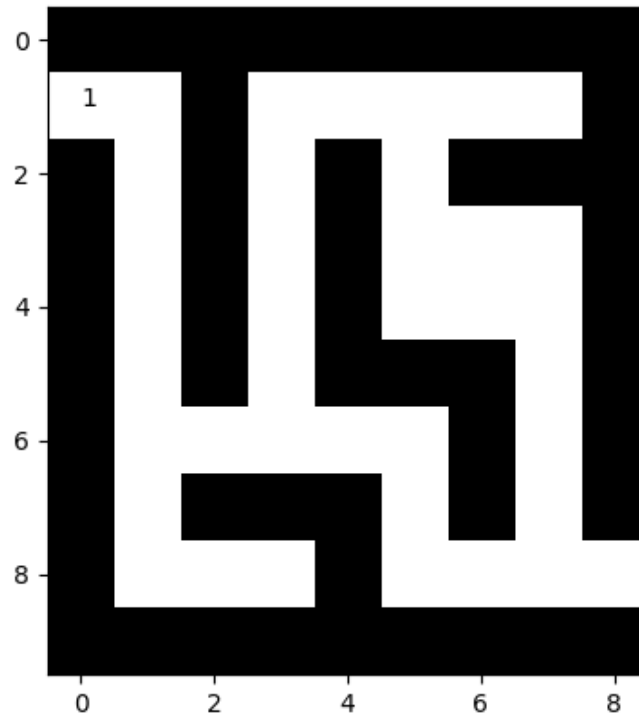
In [10]: import matplotlib.pyplot as plt
import matplotlib.animation as animation
import matplotlib as mpl
%matplotlib inline
mpl.rc('animation', html='jshtml')

maze = Maze(maze_str)
search_path, node2dist = bfs(maze, (1, 0)) # prints the order of search all
maze.plot_maze()

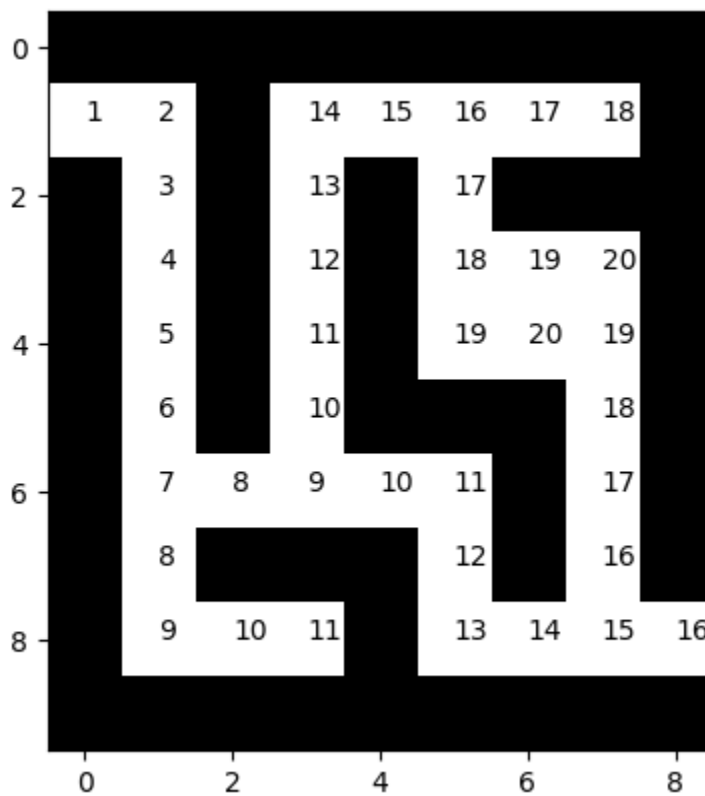
maze.animate_search_path(search_path, node2dist)

```

Out[10]:



☒ Once ☐ Loop ☐ Reflect



```
In [11]: def bfs_path(graph, start, goal):
    """
    Returns success and node2parent

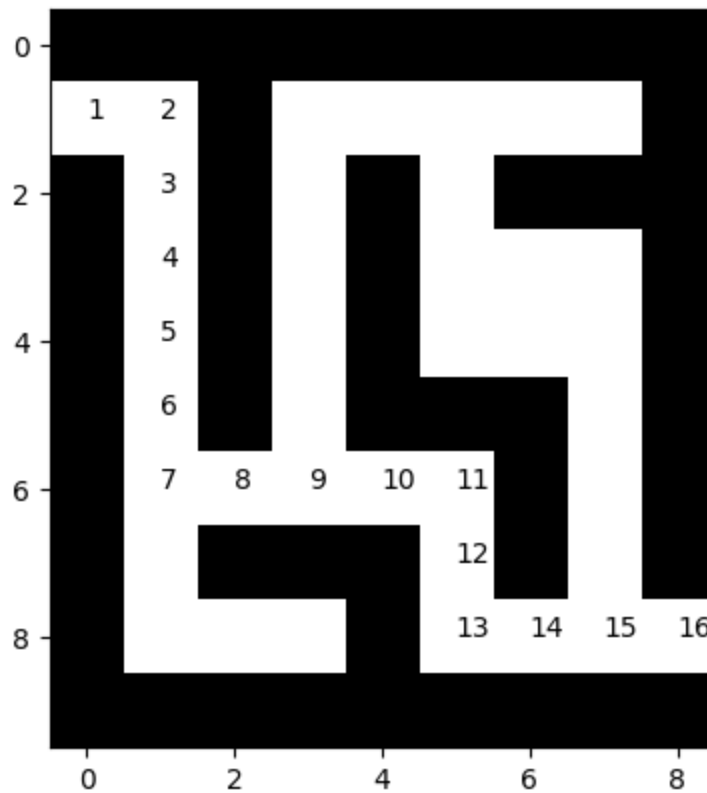
    success: True if goal is found otherwise False
    node2parent: A dictionary that contains the nearest parent for node
    """
    seen = [start] # List for seen nodes.
    # Frontier is the boundary between seen and unseen
    frontier = Queue() # Frontier of unvisited nodes as FIFO
    node2parent = dict() # Keep track of nearest parent for each node (required)
    frontier.put(start)

    while not frontier.empty(): # Creating loop to visit each node
        m = frontier.get() # Get the oldest addition to frontier
        if m == goal:
            return True, node2parent

        for neighbor in graph.get(m, []):
            if neighbor not in seen:
                seen.append(neighbor)
                frontier.put(neighbor)
                node2parent[neighbor] = m
    return False, []
```

```
In [12]: def backtrace_path(node2parent, start, goal):
    c = goal
    r_path = [c]
    parent = node2parent.get(c, None)
    while parent != start:
        r_path.append(parent)
        c = parent
        parent = node2parent.get(c, None) # Keep getting the parent until you reach start
        #print(parent)
    r_path.append(start)
    return reversed(r_path) # Reverses the path

maze = Maze(maze_str)
start = (1, 0)
goal = (8, 8)
success, node2parent = bfs_path(maze, (1, 0), (8, 8))
path = backtrace_path(node2parent, (1, 0), (8, 8))
#print(list(path))
maze.plot_path(path) # Draws all the searched nodes
plt.show()
#node2parent
```

Dijkstra algorithm



PriorityQueue

PriorityQueue returns the smallest (or the largest) item in the queue faster than other data structures

```
In [13]: #from queue import PriorityQueue
from hw2_solution import PriorityQueueUpdatable
from dataclasses import dataclass, field
from typing import Any

# https://docs.python.org/3/library/queue.html#queue.PriorityQueue
@dataclass(order=True)
class PItem:
    dist: int
    node: Any=field(compare=False)

    # Make the PItem hashable
    # https://docs.python.org/3/glossary.html#term-hashable
    def __hash__(self):
        return hash(self.node)

graph = {
    's' : [('x', 5), ('u', 10)],
```

```

'u' : [('v', 1), ('x', 2)],
'x' : [('u', 3), ('v', 9), ('y', 2)],
'y' : [('v', 6), ('s', 7)],
'v' : [('y', 4)]
}

def dijkstra(graph, start, goal, debug=False):
    """
    edgecost: cost of traversing each edge

    Returns success and node2parent

    success: True if goal is found otherwise False
    node2parent: A dictionary that contains the nearest parent for node
    """
    seen = set([start]) # Set for seen nodes.
    # Frontier is the boundary between seen and unseen
    frontier = PriorityQueueUpdatable() # Frontier of unvisited nodes as a P
    node2parent = {start : None} # Keep track of nearest parent for each node
    node2dist = {start: 0} # Keep track of cost to arrive at each node
    search_order = []
    frontier.put(PItem(0, start))
    i = 0
    while not frontier.empty():
        # Creating loop to visit each node
        dist_m = frontier.get() # Get the smallest addition to the frontier
        if debug: print("%d) Q = " % i, list(frontier.queue), end='; ')
        if debug: print("dists = " , [node2dist[n.node] for n in frontier.queue])
        m = dist_m.node
        m_dist = node2dist[m]
        search_order.append(m)
        if goal is not None and m == goal:
            return True, search_order, node2parent, node2dist

        for neighbor, edge_cost in graph.get(m, []):
            old_dist = node2dist.get(neighbor, float("inf"))
            new_dist = edge_cost + m_dist
            if neighbor not in seen:
                seen.add(neighbor)
                frontier.put(PItem(new_dist, neighbor))
                node2parent[neighbor] = m
                node2dist[neighbor] = new_dist
            elif new_dist < old_dist:
                node2parent[neighbor] = m
                node2dist[neighbor] = new_dist
                # ideally you would update the dist of this item in the priority queue
                # as well. But python priority queue does not support fast updates
                old_item = PItem(old_dist, neighbor)
                if old_item in frontier:
                    frontier.replace(old_item, PItem(new_dist, neighbor))

        i += 1
    if goal is not None:
        return False, [], {}, node2dist
    else:
        return True, search_order, node2parent, node2dist

```

```
In [14]: success, search_path, node2parent, node2dist = dijkstra(graph, 's', None, de
print(success, node2parent, node2dist)
```

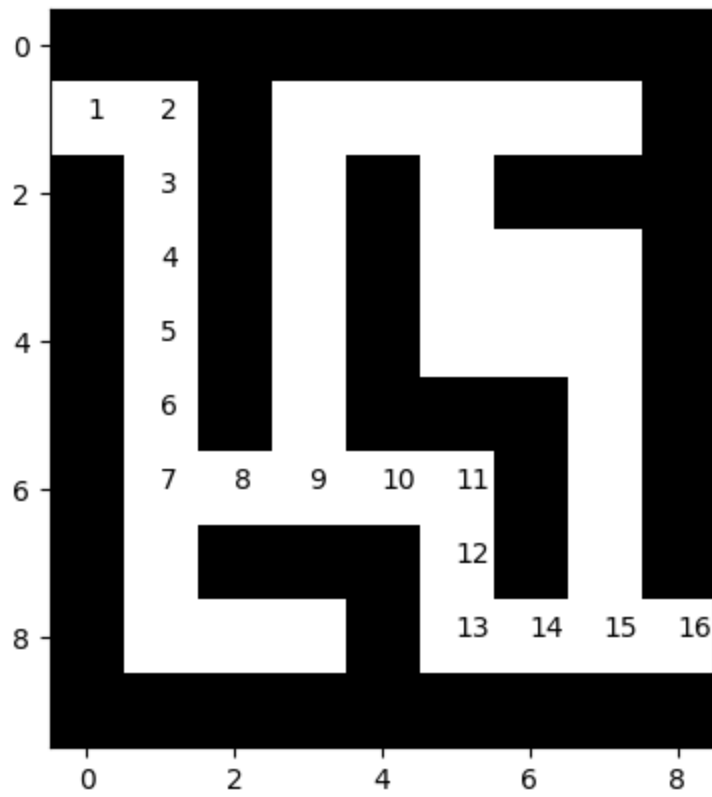
```
0) Q = []; dists = []
1) Q = [PItem(dist=10, node='u')]; dists = [10]
2) Q = [PItem(dist=8, node='u'), PItem(dist=14, node='v')]; dists = [8, 1
4]
3) Q = [PItem(dist=13, node='v')]; dists = [13]
4) Q = []; dists = []
True {'s': None, 'x': 's', 'u': 'x', 'v': 'u', 'y': 'x'} {'s': 0, 'x': 5,
'u': 8, 'v': 9, 'y': 7}
```

```
In [15]: import itertools
```

```
class MazeD(Maze):
    def get(self, node, default):
        nbrs = Maze.get(self, node, default)
        return zip(nbrs, itertools.repeat(1))

maze = MazeD(maze_str)
success, search_path, node2parent, node2dist = dijkstra(maze, (1, 0), (8, 8))
print(success, node2parent)
if success:
    path = backtrace_path(node2parent, (1, 0), (8, 8))
    maze.plot_path(path) # Draws all the searched nodes
```

```
True {(1, 0): None, (1, 1): (1, 0), (2, 1): (1, 1), (3, 1): (2, 1), (4, 1):
(3, 1), (5, 1): (4, 1), (6, 1): (5, 1), (6, 2): (6, 1), (7, 1): (6, 1), (6,
3): (6, 2), (8, 1): (7, 1), (6, 4): (6, 3), (5, 3): (6, 3), (8, 2): (8, 1),
(6, 5): (6, 4), (4, 3): (5, 3), (8, 3): (8, 2), (7, 5): (6, 5), (3, 3): (4,
3), (8, 5): (7, 5), (2, 3): (3, 3), (8, 6): (8, 5), (1, 3): (2, 3), (8, 7):
(8, 6), (1, 4): (1, 3), (8, 8): (8, 7), (7, 7): (8, 7), (1, 5): (1, 4)}
```



```
In [16]: maze_str = \
        """
```

[illegible]


```

init_func=self.plot_maze, blit=True, r
return anim

```

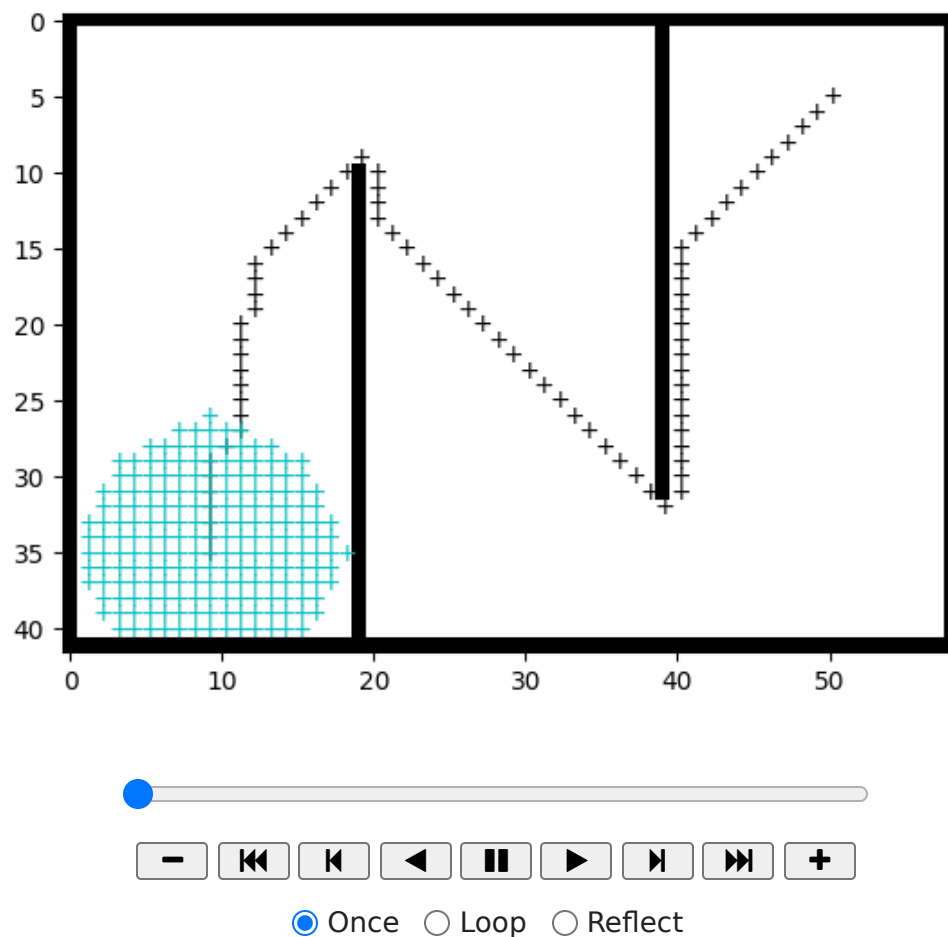
```

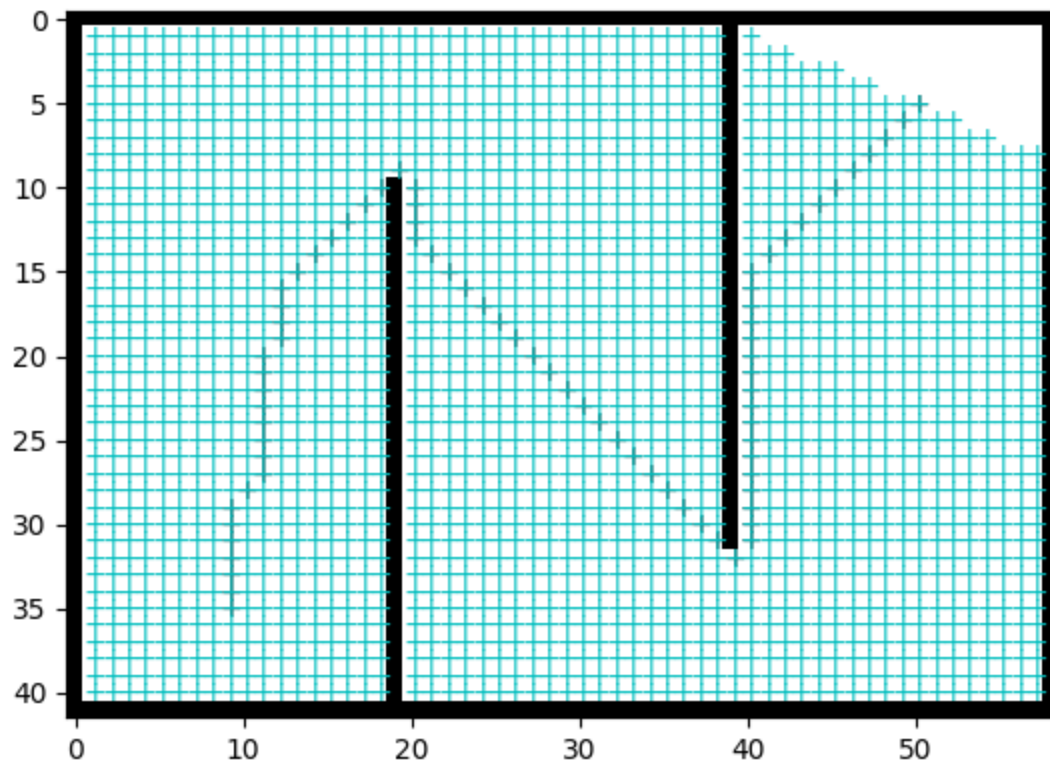
In [18]: maze = Maze8(maze_str)
success, search_path, node2parent, node2dist = dijkstra(maze, start_pos, goal_pos)
#print(success, search_path)
assert success
anim = maze.animate(search_path)
path = backtrace_path(node2parent, start_pos, goal_pos)
#maze.init_plots(reinit=True)
path_plot = maze.plot_path(path, color='k') # Draws the traced shortest path
anim

```

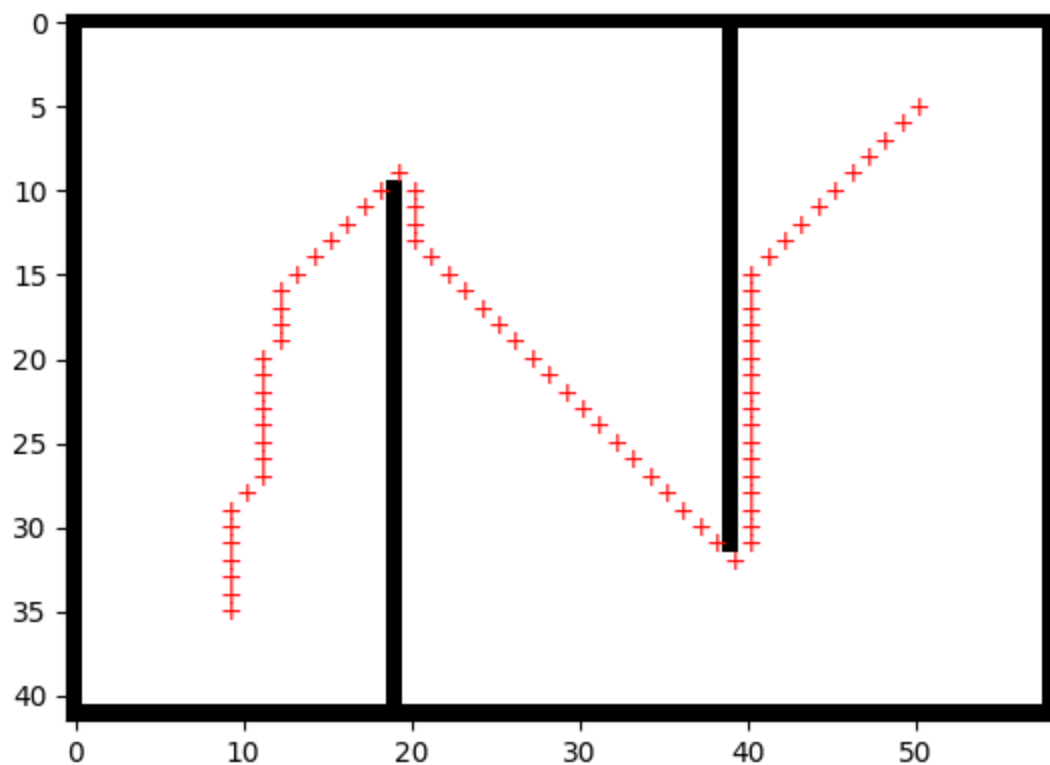
/tmp/ipykernel_26974/955263672.py:37: UserWarning: frames=<generator object
batched at 0x7f5c0c3632e0> which we can infer the length of, did not pass an
explicit *save_count* and passed cache_frame_data=True. To avoid a possibly
unbounded cache, frame data caching has been disabled. To suppress this warn
ing either pass `cache_frame_data=False` or `save_count=MAX_FRAMES`.
anim = animation.FuncAnimation(self.fig, self._plot_path,

Out[18]:



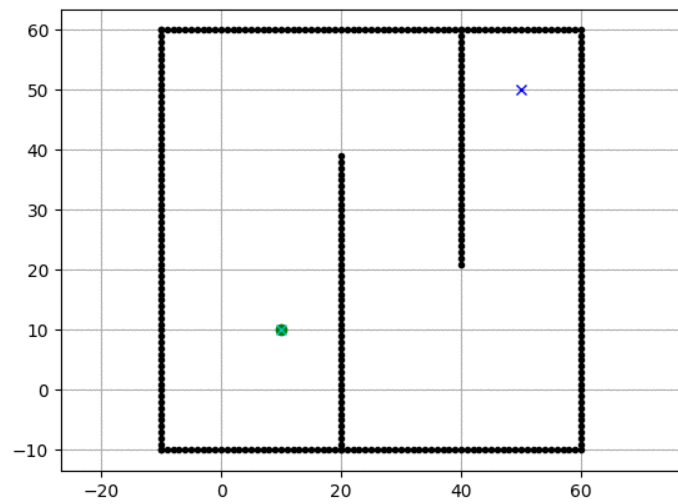
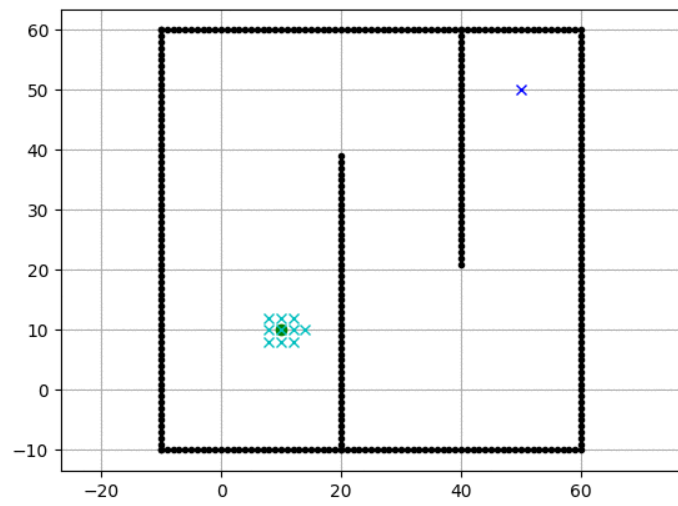


```
In [19]: path = backtrace_path(node2parent, (35, 9), (5, 50))
maze.init_plots(reinit=True)
maze.plot_path(path, color='r') # Draws the traced shortest path
plt.show()
```

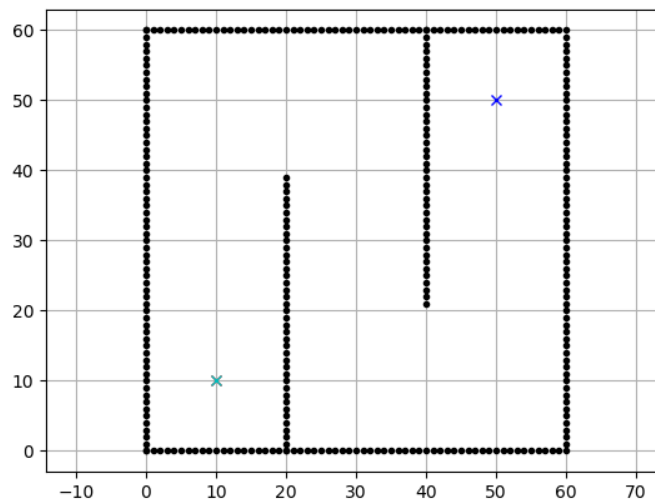
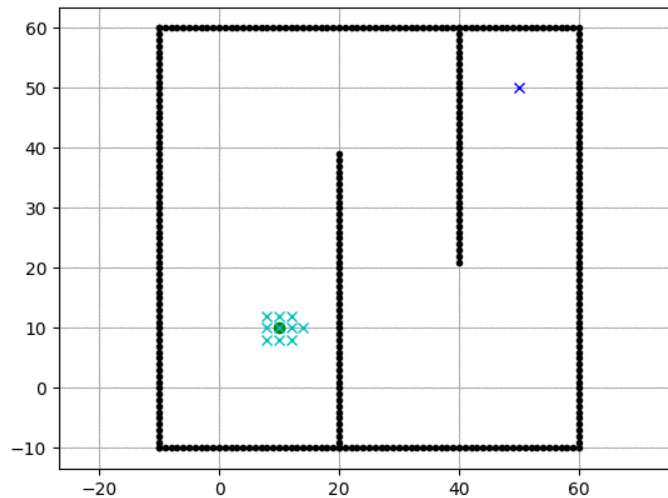


Search order in BFS vs DFS vs Dijkstra

Breadth first search vs Depth first search



Breadth first search vs Dijkstra



Computational complexity of BFS

```
In [20]: # Write down the computational complexity of each line in big-O notation  $O()$ 
# Assume the graph has  $|V|$  nodes and  $|E|$  edges
def bfs_barebones(graph, start):
    seen = {start} # Set for seen nodes (contains both frontier and dead sta
    # Frontier is the boundary between seen and unseen (Also called the aliv
    frontier = Queue() # Frontier of unvisited nodes as FIFO #  $O(1)$ 
    frontier.put(start) #  $O(1)$ 

    while not frontier.empty(): # Creating loop to visit each node #  $O(|V|)$ 
        m = frontier.get() # Get the oldest addition to frontier #  $O(|V| * 1)$ 

        for neighbor in graph.get(m, []): #  $O(|V| * |E|/|V|) = O(|E|)$ 
            if neighbor not in seen: #  $O(|E| * 1)$ 
                seen.add(neighbor) #  $O(|E| * 1)$ 
                frontier.put(neighbor) #  $O(|E| * 1)$ 
```

```
# The computational complexity of BFS is  $O(|E|)$ . Some books write it as  $O(|V|)$ 
# where  $O(|V|)$  is the cost of initializing states of different nodes
```

Computational complexity of Dijkstra

```
In [21]: # Write down the computational complexity of each line in big-O notation  $O()$ 
# Assume the graph has  $|V|$  nodes and  $|E|$  edges
def dijkstra_barebones(graph, start):
    seen = {start} # Set for seen nodes (contains both frontier and dead states)
    # Frontier is the boundary between seen and unseen (Also called the alive frontier)
    frontier = PriorityQueue() # Frontier of unvisited nodes as PriorityQueue
    frontier.put(PItem(0, start)) #  $O(1)$ 
    node2dist = {start: 0} # Keep track of cost to arrive at each node #  $O(1)$ 

    while not frontier.empty(): # Creating loop to visit each node
        dist_and_node = frontier.get() # Get the smallest dist node #  $O(|V|)$ 
        m_dist = dist_and_node.dist
        m = dist_and_node.node

        for neighbor, edge_dist in graph.get(m, []): #  $O(|V| * |E|/|V|) = O(|E|)$ 
            if neighbor not in seen: #  $O(|E| * 1)$ 
                seen.add(neighbor) #  $O(|E| * 1)$ 
                frontier.put(neighbor) #  $O(|E| * \log(1))$  # for fibonacci heap
                node2dist[neighbor] = m_dist + edge_dist #  $O(1)$ 
            elif node2dist[neighbor] > m_dist + edge_dist: #  $O(1)$ 
                node2dist[neighbor] = m_dist + edge_dist #  $O(1)$ 

# The computational complexity of Dijkstra is  $O(|V|\log(|V|) + |E|)$  when implemented
# using a Fibonacci heap based PriorityQueue
```

PriorityQueue (Heaps Chapter 7 of Carmen's intro to algorithms)



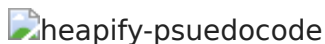
Heap property

1. $H[\text{Parent}(i)] \geq H[i]$
2. $\text{Parent}(i) = \text{ceil}(i/2)$
3. $\text{LeftChild}(i) = 2i$
4. $\text{RightChild}(i) = 2i+1$

Heapify



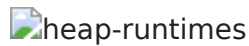
Heapify pseudocode



Heap Insert



Heap runtimes



A-star (A*) algorithm

(Required reading: 3.5.2 of Russel and Norving: Artificial Intelligence)



```
In [22]: from hw2_solution import PriorityQueueUpdatable
import sys
def astar(graph, heuristic_dist_fn, start, goal, debug=False, debugf=sys.stdout):
    """
    edgecost: cost of traversing each edge

    Returns success and node2parent

    success: True if goal is found otherwise False
    node2parent: A dictionary that contains the nearest parent for node
    """
    seen = set([start]) # Set for seen nodes.
    # Frontier is the boundary between seen and unseen
    frontier = PriorityQueueUpdatable() # Frontier of unvisited nodes as a P
    node2parent = {start : None} # Keep track of nearest parent for each node
    hfn = heuristic_dist_fn # make the name shorter
    node2dist = {start: 0 } # Keep track of cost to arrive at each node
    search_order = []
    frontier.put(PItem(0 + hfn(start, goal), start)) # <----- Dift

    if debug: debugf.write("goal = " + str(goal) + '\n')
    i = 0
    while not frontier.empty():
        # Creating loop to visit each node
        dist_m = frontier.get() # Get the smallest addition to the frontier
        if debug: debugf.write("%d) Q = " % i + str(list(frontier.queue)) +
        if debug: debugf.write("%d) node = " % i + str(dist_m) + '\n')
        #if debug: print("dists = " , [node2dist[n.node] for n in frontier.queue])
        m = dist_m.node
        m_dist = node2dist[m]
        search_order.append(m)
        if goal is not None and m == goal:
            return True, search_order, node2parent, node2dist

        for neighbor, edge_cost in graph.get(m, []):
            old_dist = node2dist.get(neighbor, float("inf"))
            new_dist = edge_cost + m_dist
            if neighbor not in seen:
                seen.add(neighbor)
                frontier.put(PItem(new_dist + hfn(neighbor, goal), neighbor))
```

```

        node2parent[neighbor] = m
        node2dist[neighbor] = new_dist
    elif new_dist < old_dist:
        node2parent[neighbor] = m
        node2dist[neighbor] = new_dist
        # ideally you would update the dist of this item in the priority queue
        # as well. But python priority queue does not support fast updates
        # ----- Different from dijkstra -----
        old_item = PItem(old_dist + hfn(neighbor, goal), neighbor)
        if old_item in frontier:
            frontier.replace(
                old_item,
                PItem(new_dist + hfn(neighbor, goal), neighbor))
    i += 1
    if goal is not None:
        return False, [], {}, node2dist
    else:
        return True, search_order, node2parent, node2dist

```

```

In [23]: import math
        from functools import partial

        def euclidean_heurist_dist(node, goal, scale=1):
            x_n, y_n = node
            x_g, y_g = goal
            return scale*math.sqrt((x_n-x_g)**2 + (y_n - y_g)**2)

```

```

In [24]: maze = Maze8(maze_str)
        debugf=open('log.txt', 'w')
        success, search_path, node2parent, node2dist = astar(
            maze, partial(euclidean_heurist_dist, scale=1),
            start_pos, goal_pos, debug=True, debugf=debugf)
        debugf.close()

        #print(success, search_path)
        assert success
        anim = maze.animate(search_path)
        path = backtrace_path(node2parent, start_pos, goal_pos)
        #maze.init_plots(reinit=True)
        path_plot = maze.plot_path(path, color='k') # Draws the traced shortest path
        anim

```

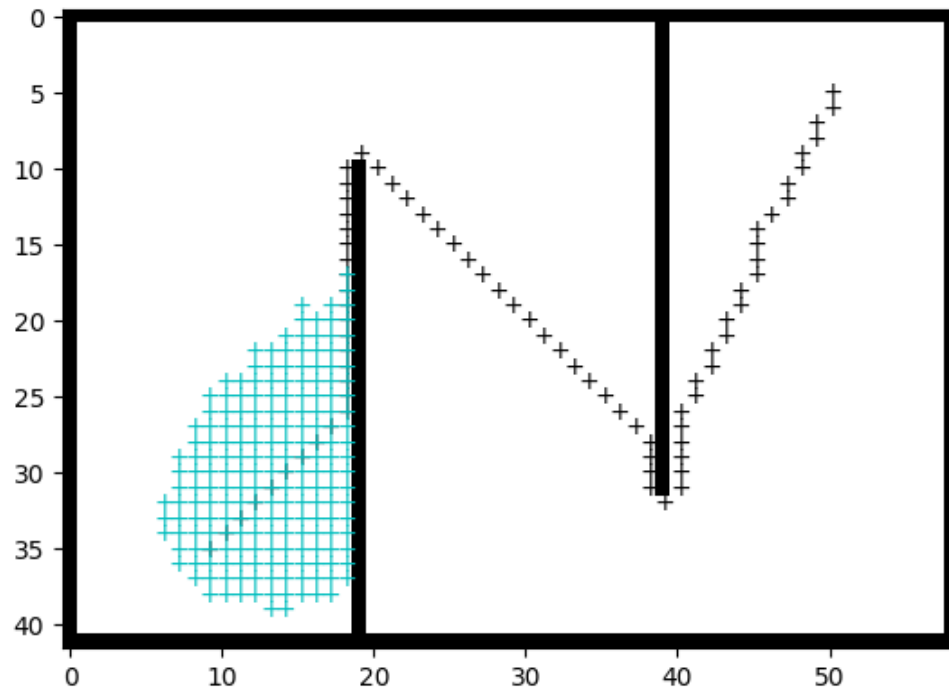
/tmp/ipykernel_26974/955263672.py:37: UserWarning: frames=<generator object batched at 0x7f5c07250430> which we can infer the length of, did not pass an explicit *save_count* and passed cache_frame_data=True. To avoid a possibly unbounded cache, frame data caching has been disabled. To suppress this warning either pass `cache_frame_data=False` or `save_count=MAX_FRAMES`.

```

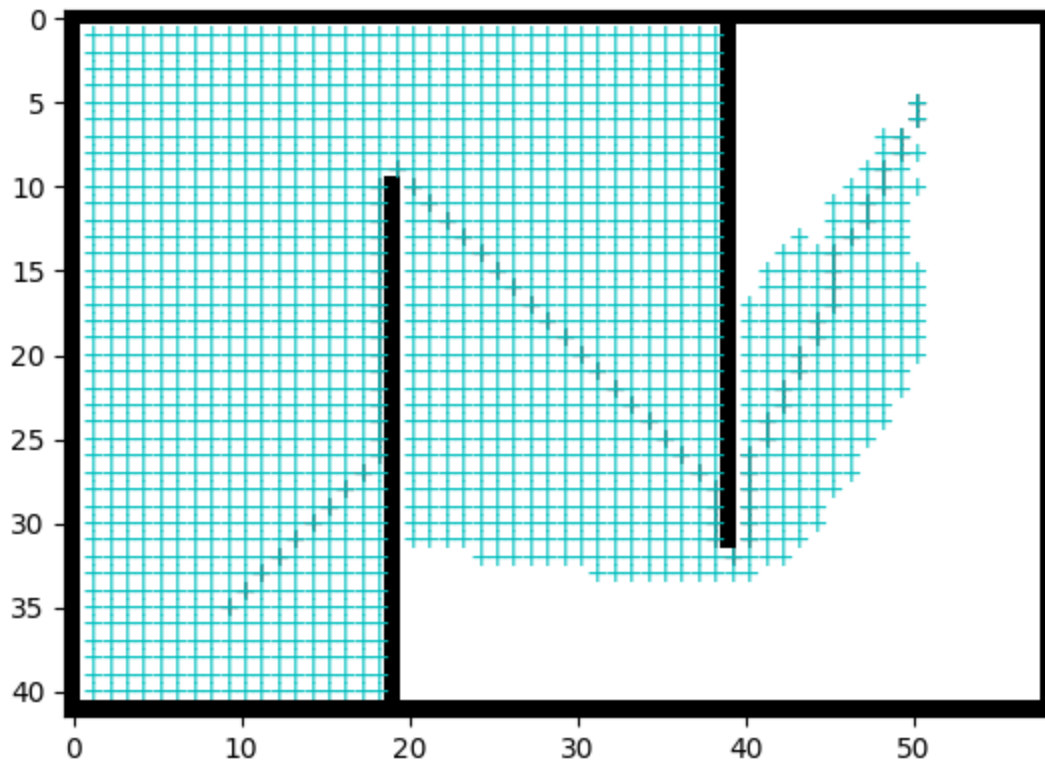
    anim = animation.FuncAnimation(self.fig, self._plot_path,

```

Out[24]:



☒ Once ☐ Loop ☐ Reflect




```

path = backtrace_path(node2parent, start_pos, goal_pos)
#maze.init_plots(reinit=True)
path_plot = maze.plot_path(path, color='k') # Draws the traced shortest path
anim

```

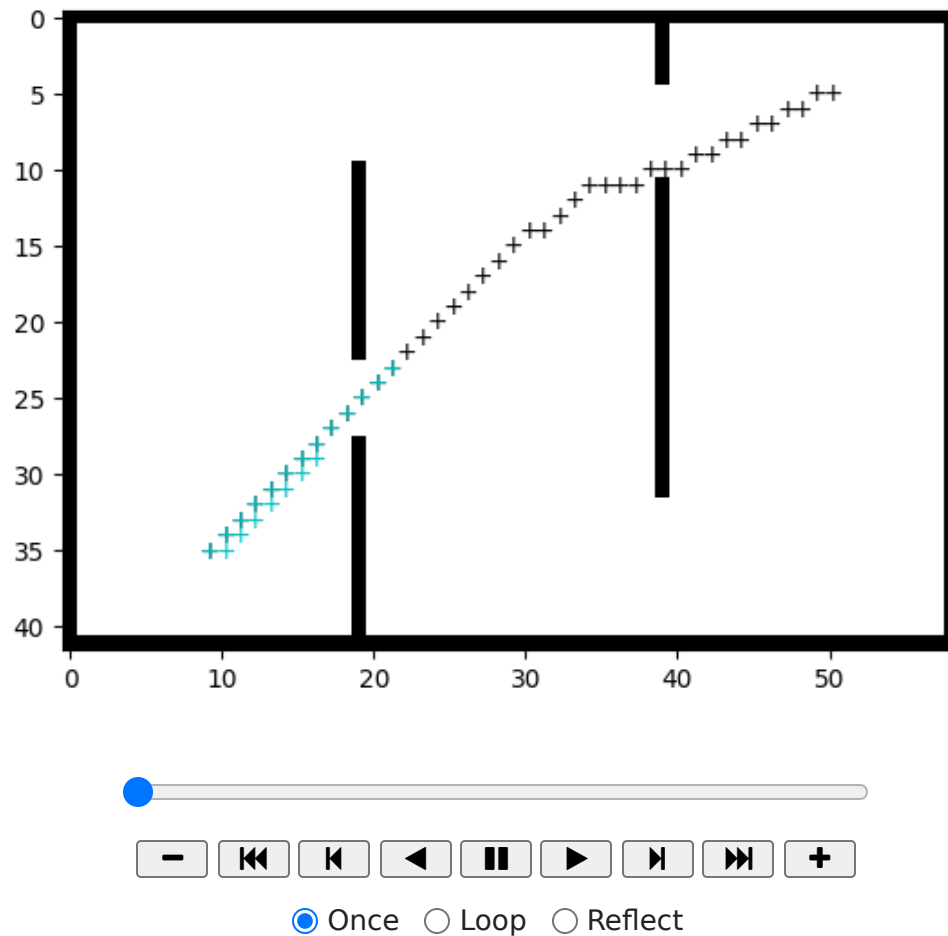
/tmp/ipykernel_26974/955263672.py:37: UserWarning: frames=<generator object batched at 0x7f5c0724e2e0> which we can infer the length of, did not pass an explicit *save_count* and passed cache_frame_data=True. To avoid a possibly unbounded cache, frame data caching has been disabled. To suppress this warning either pass `cache_frame_data=False` or `save_count=MAX_FRAMES`.

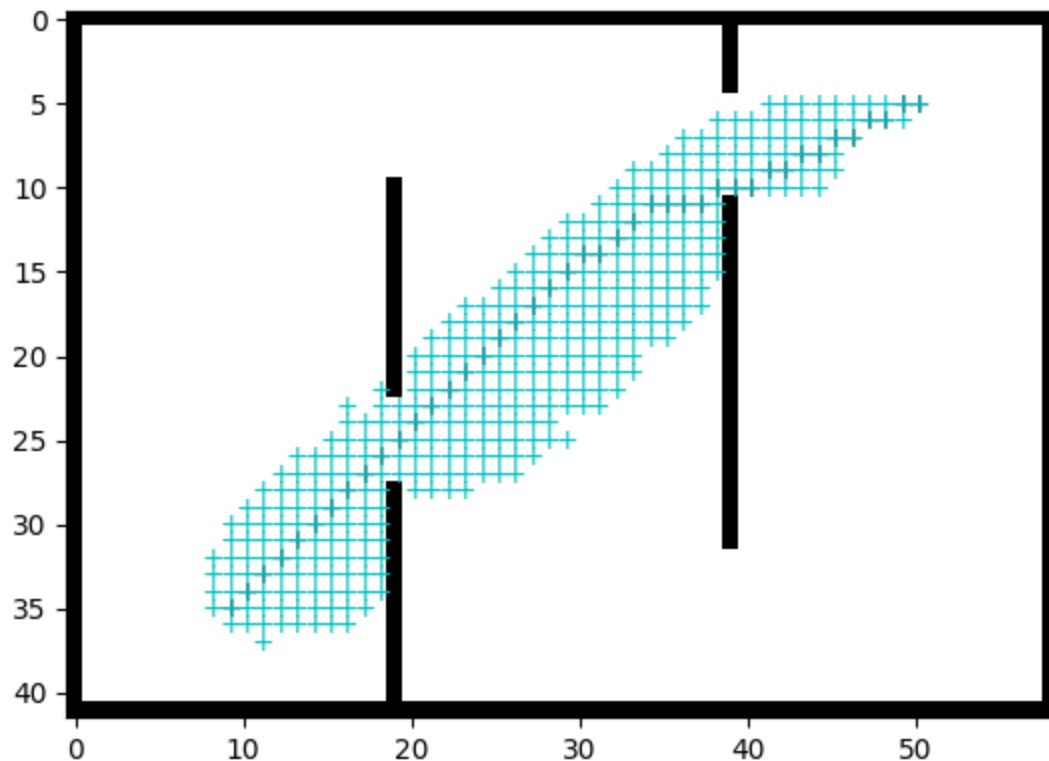
```

anim = animation.FuncAnimation(self.fig, self._plot_path,

```

Out[26]:



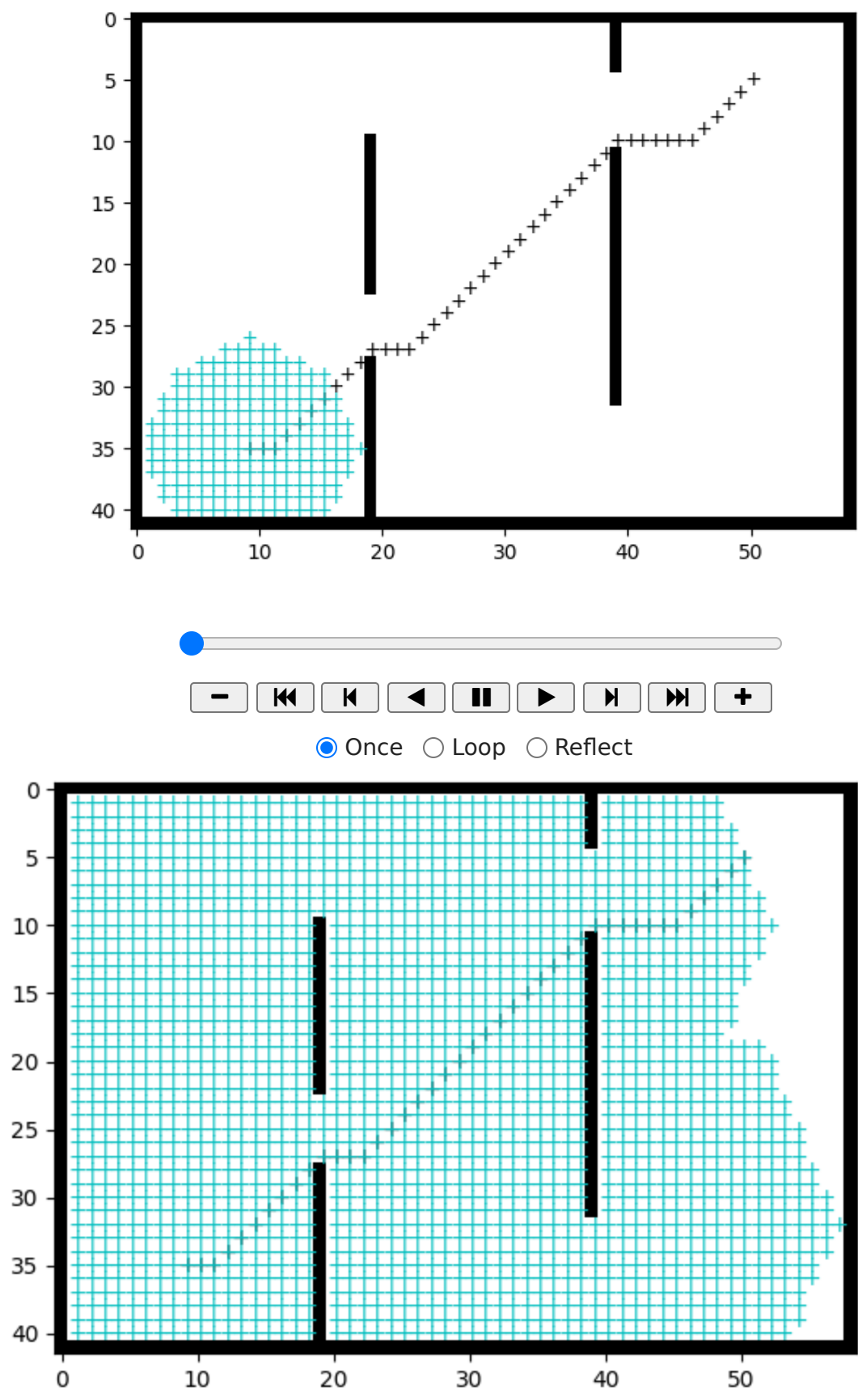


```
In [27]: maze = Maze8(maze_str)
success, search_path, node2parent, node2dist = astar(
    maze, partial(euclidean_heurist_dist, scale=0),
    start_pos, goal_pos)
#print(success, search_path)
assert success
anim = maze.animate(search_path)
path = backtrace_path(node2parent, start_pos, goal_pos)
#maze.init_plots(reinit=True)
path_plot = maze.plot_path(path, color='k') # Draws the traced shortest path
anim
```

/tmp/ipykernel_26974/955263672.py:37: UserWarning: frames=<generator object batched at 0x7f5c060b4e40> which we can infer the length of, did not pass an explicit *save_count* and passed cache_frame_data=True. To avoid a possibly unbounded cache, frame data caching has been disabled. To suppress this warning either pass `cache_frame_data=False` or `save_count=MAX_FRAMES`.

```
anim = animation.FuncAnimation(self.fig, self._plot_path,
```


Out[27]:




```

path = backtrace_path(node2parent, start_pos, goal_pos)
#maze.init_plots(reinit=True)
path_plot = maze.plot_path(path, color='k') # Draws the traced shortest path
anim

```

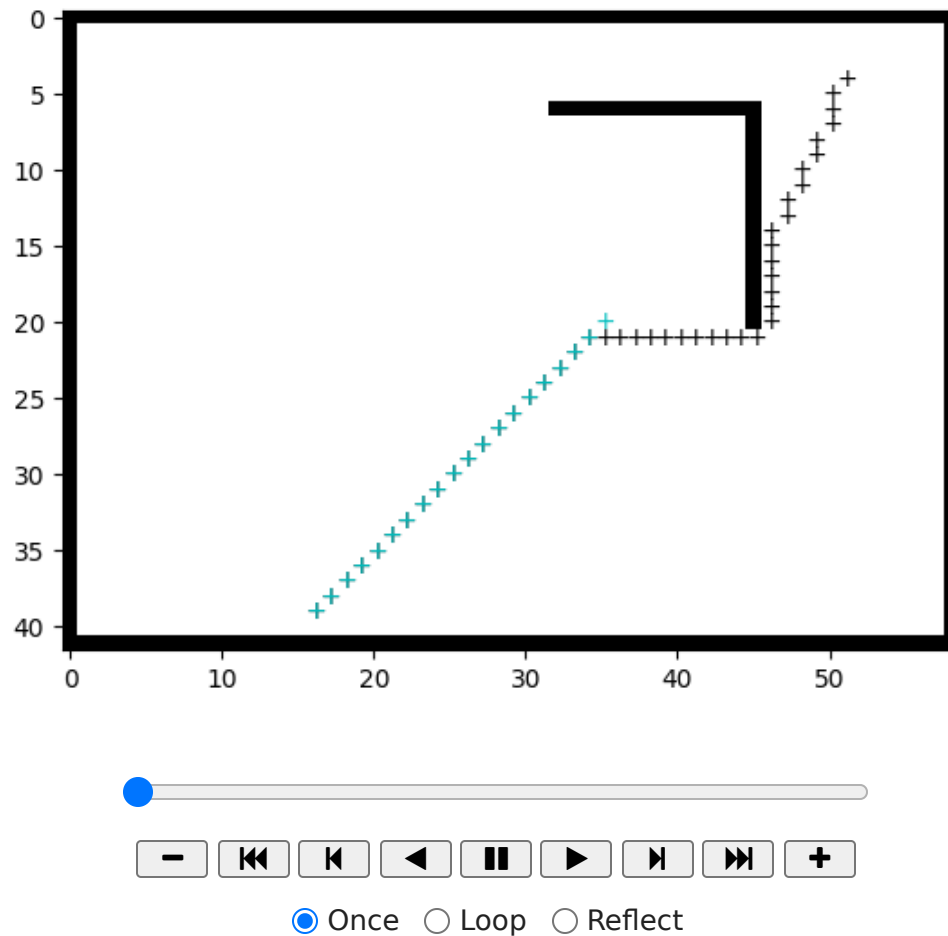
/tmp/ipykernel_26974/955263672.py:37: UserWarning: frames=<generator object batched at 0x7f5c05036900> which we can infer the length of, did not pass an explicit *save_count* and passed cache_frame_data=True. To avoid a possibly unbounded cache, frame data caching has been disabled. To suppress this warning either pass `cache_frame_data=False` or `save_count=MAX_FRAMES`.

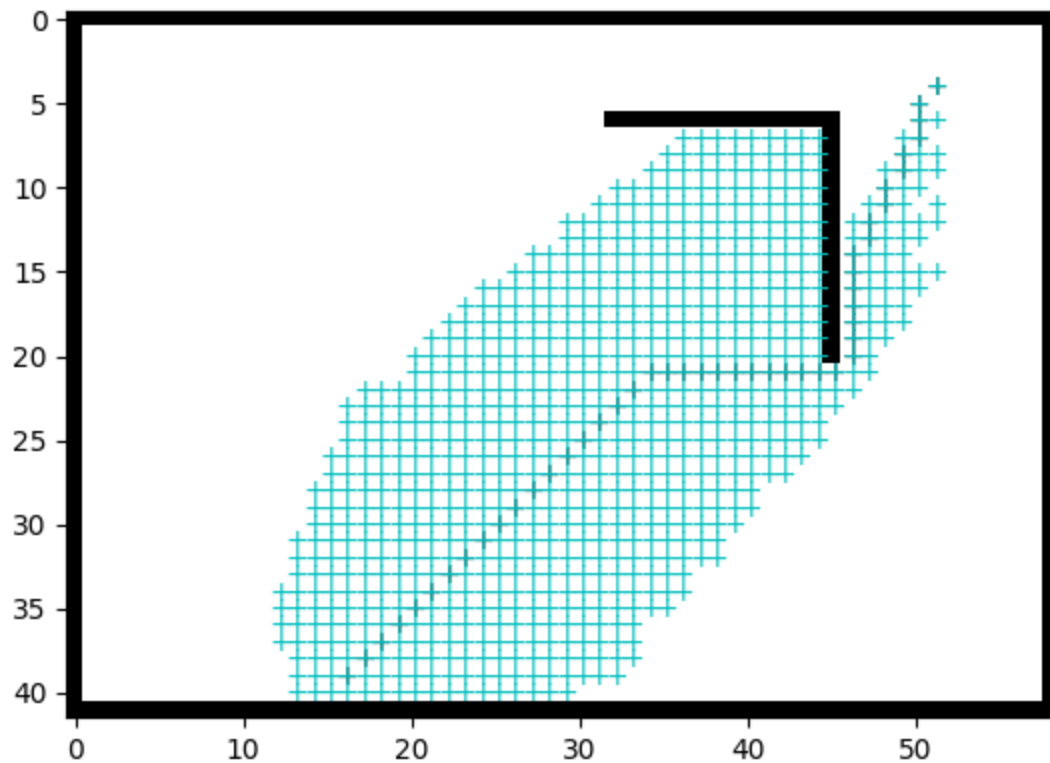
```

anim = animation.FuncAnimation(self.fig, self._plot_path,

```

Out[29]:



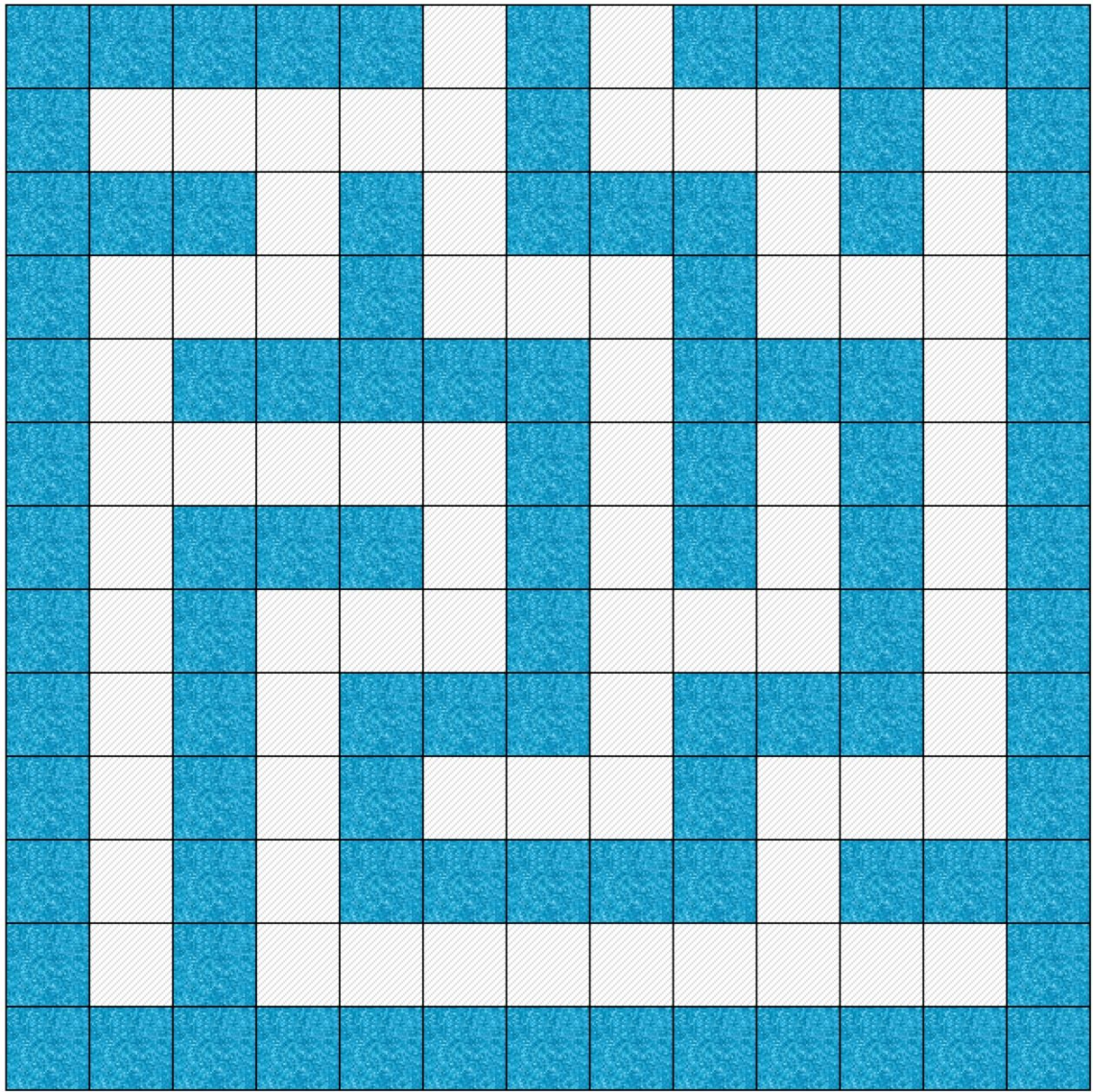


Admissibility and Consistency of heuristic function

1. An admissible heuristic is one that never overestimates the cost to reach the goal.
2. A heuristic $h(n)$ is consistent if it satisfies the triangle inequality:

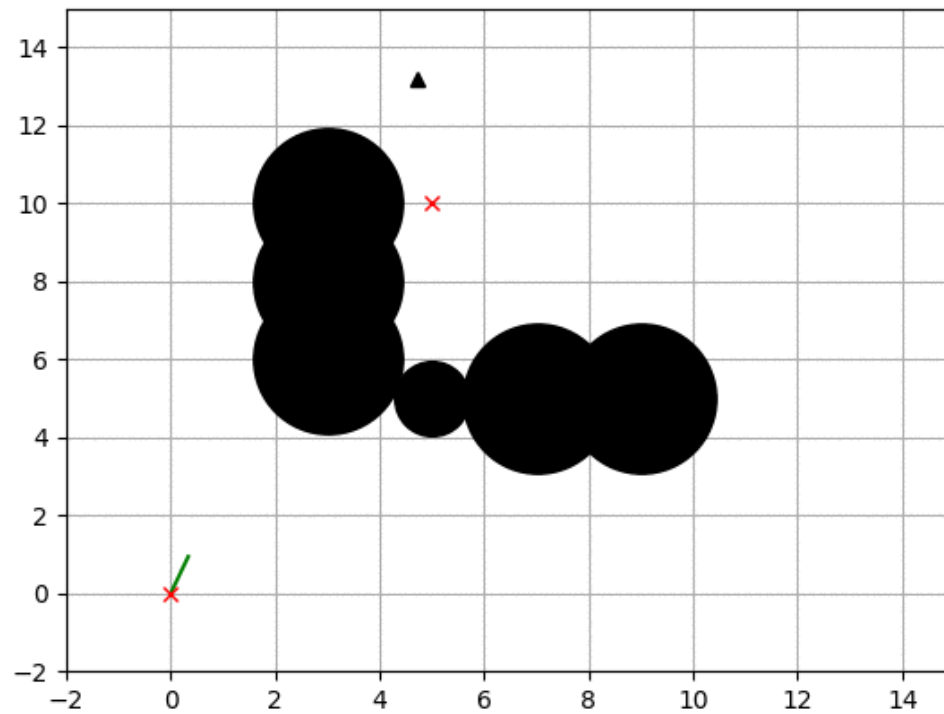
$$h(s_t) \leq c(s_t, s_{t+1}) + h(s_{t+1}).$$

Converting maze to grid to graph



Other ways of converting a maze into a graph

Rapidly exploring random trees



In []: