Planning (Chapter 2 from Lavalle book)

Abstraction of a planning problem

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

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 $v_s = \mathrm{start}(e)$ and second vertex is called the end $v_e = \mathrm{end}(e)$.

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

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, $\mathcal{U}(\mathrm{s}_t) = \{(\mathrm{s}_t, \mathrm{s}_j) \mid (\mathrm{s}_t, \mathrm{s}_j) \in \mathcal{E}\}$.
3. State transition function is the other end of th edge,

$\mathrm{s}_{t+1} = f(\mathrm{s}_t, \mathrm{u}_t) = \mathrm{end}(\mathrm{u}_t)$, where $\mathrm{s}_t = \mathrm{start}(\mathrm{u}_t)$.

Representations of Graphs

(Chapter 23 of Introduction to Algorithms by Carmen et al)

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 nump
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],
    5 : [6]
}

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

```python
    [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]:

```python
# Exercise 1

# Write a function that converts a graph in adjacency list format to adjacen
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

bfs.png

2. Depth First Search

dfs.png

Breadth first search (BFS)

bfs-states

In [4]:

```python
from queue import Queue, LifoQueue, PriorityQueue

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

```python
  '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 aliv
    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 shoulc
                node2dist[neighbor] = min(node2dist[neighbor], node2dist[m]

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

In [5]:

```python
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

image.png

bfs-states

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 aliv
    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('ABCDEFG', 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
```

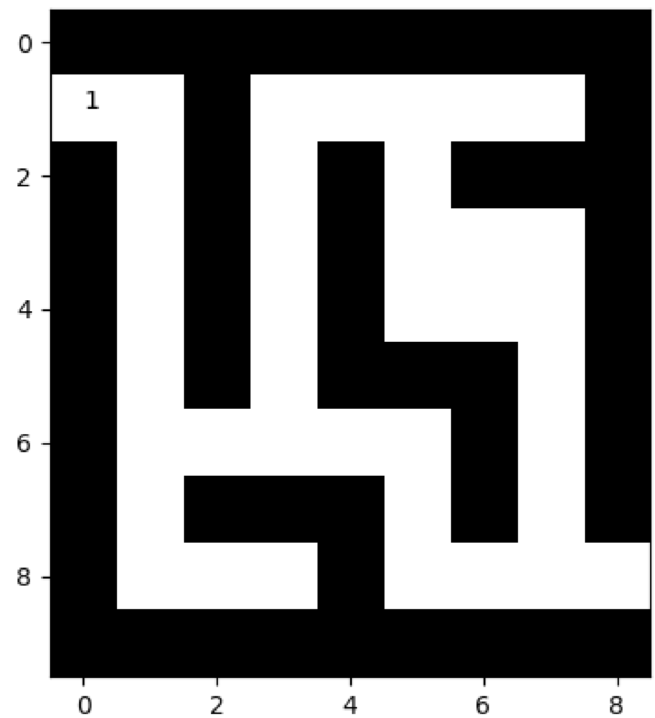
In [9]:
```
import matplotlib.pyplot as plt
import numpy as np
maze_str = \
"""
+++++++++
  +     +
+ + + +++
+ + +   +
+ + +   +
```

```
+ + +++ +
+     + +
+ +++ + +
+   +
+++++++++
"""

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.FR
            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
```

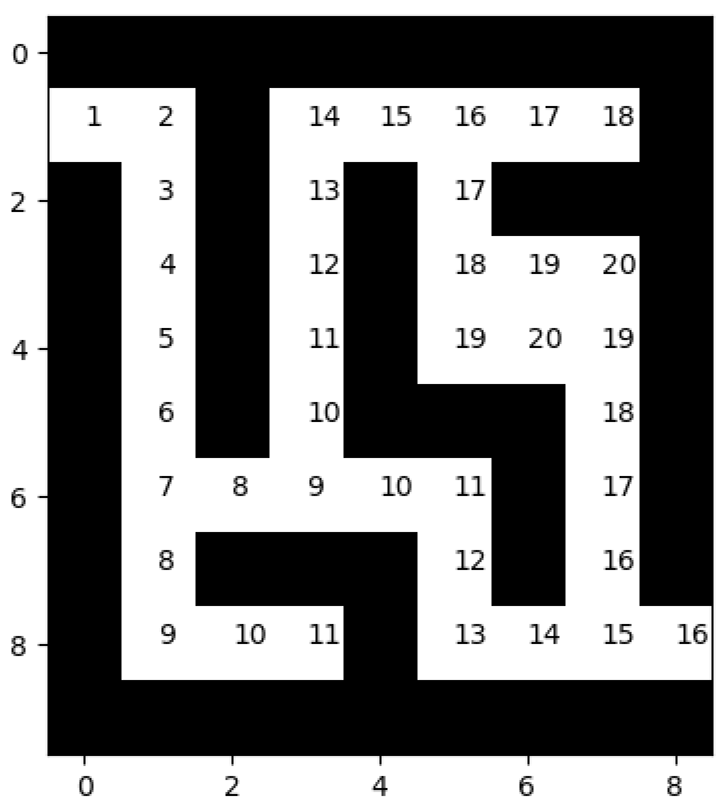
In [10]:

```
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

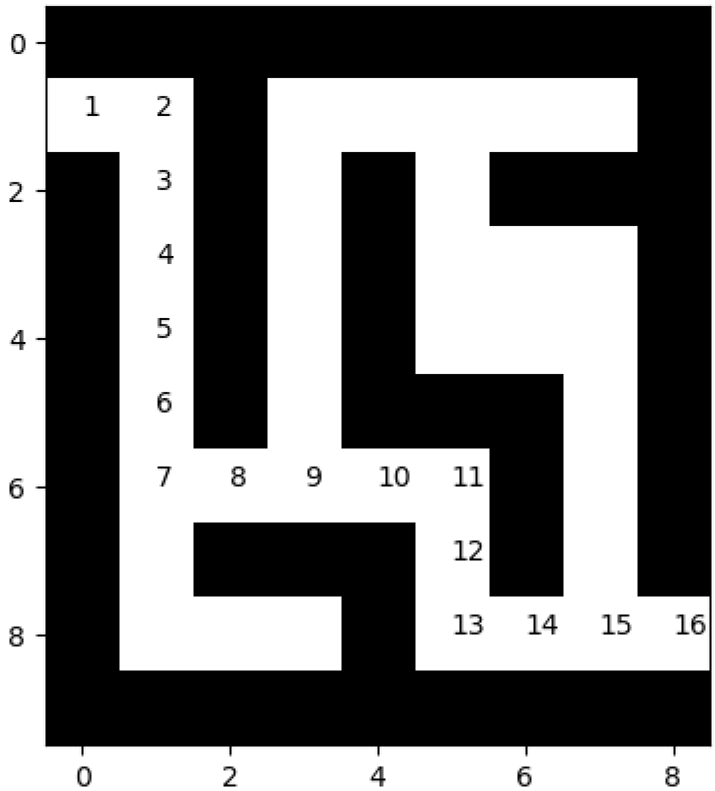


In [11]:

```python
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 (requi
    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]:

```python
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)
        #print(parent)
    r_path.append(start)
    return reversed(r_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

dijkstra-step-by-step

PriorityQueue

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

In [13]:

```
from queue import PriorityQueue
from dataclasses import dataclass, field
from typing import Any


@dataclass(order=True)
class PItem:
    dist: int
    node: Any=field(compare=False)

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)]
}
```

```python
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 = PriorityQueue() # Frontier of unvisited nodes as a Priority Q
    node2parent = {start : None} # Keep track of nearest parent for each noc
    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.qu
        m_dist = dist_m.dist
        m = dist_m.node
        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 pric
                # as well. But python priority queue does not support fast u
                # frontier.update(PItem(old_dist, neighbor), new_dist)
        i += 1
    if goal is not None:
        return False, [], {}, node2dist
    else:
        return True, search_order, node2parent, node2dist
```

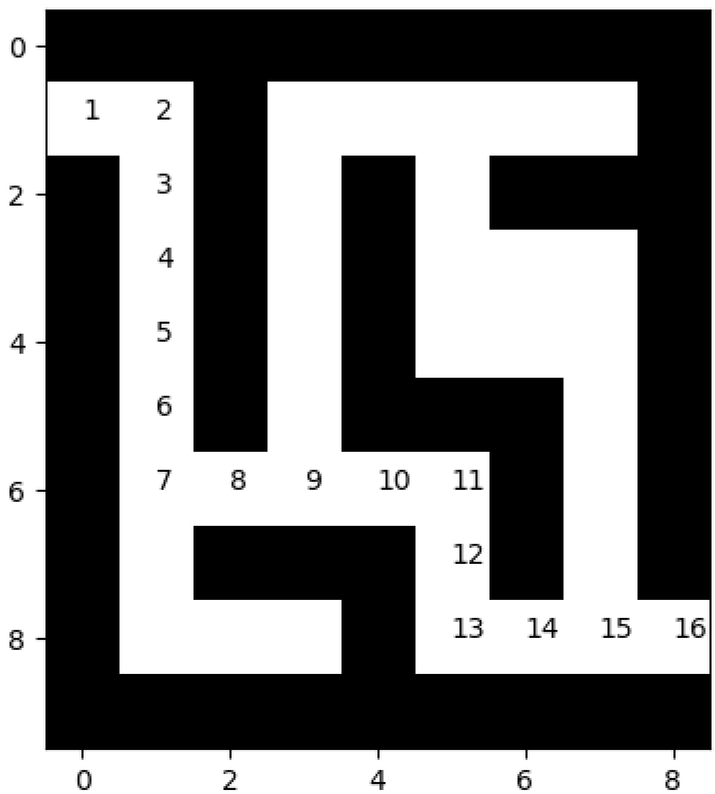
In [14]:
```python
success, search_path, node2parent, node2dist = dijkstra(graph, 's', None, de
print(success, node2parent, node2dist)
```

```
0) Q =  []; dists =  []
1) Q =  [PItem(dist=10, node='u')]; dists =  [10]
2) Q =  [PItem(dist=10, node='u'), PItem(dist=14, node='v')]; dists =  [8,
14]
3) Q =  [PItem(dist=14, 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': 11, 'y': 7}
```

In [15]:

```python
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]:

```python
maze_str = \
"""
```

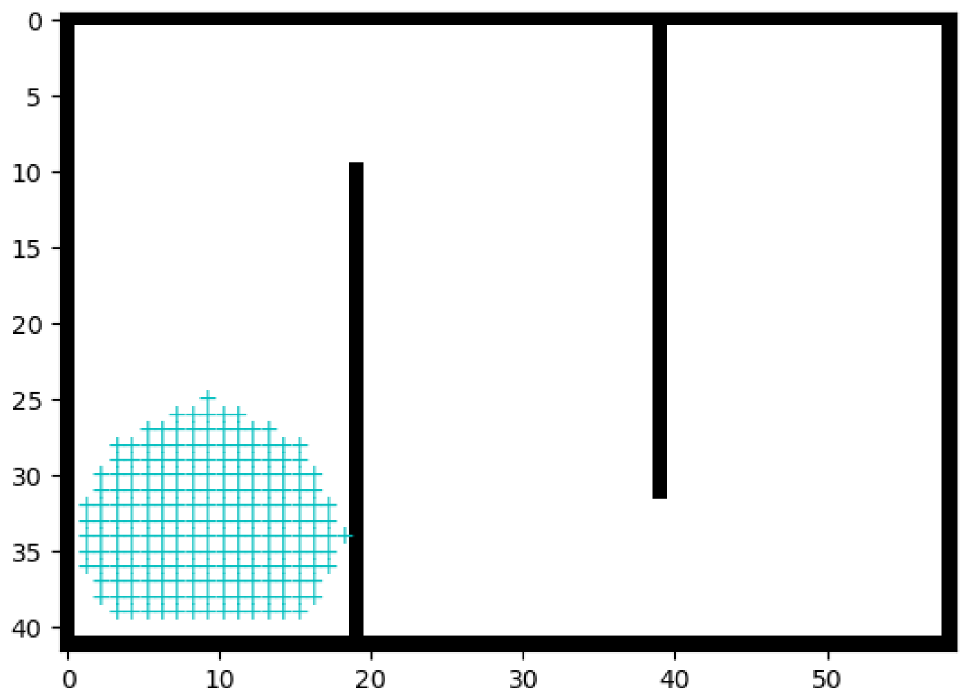
```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+                                      +                  +
+                                      +                  +
+                                      +                  +
+                                      +                  +
+                                      +                  +
+                                      +                  +
+                                      +                  +
+                                      +                  +
+                                      +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                   +                  +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+                  +                                      +
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
"""
```

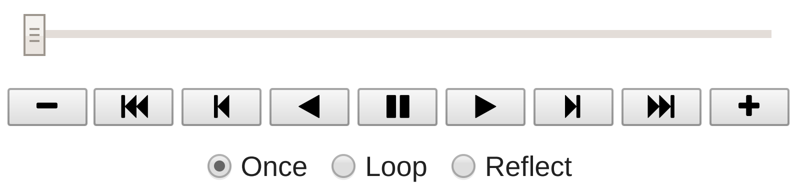
In [17]:

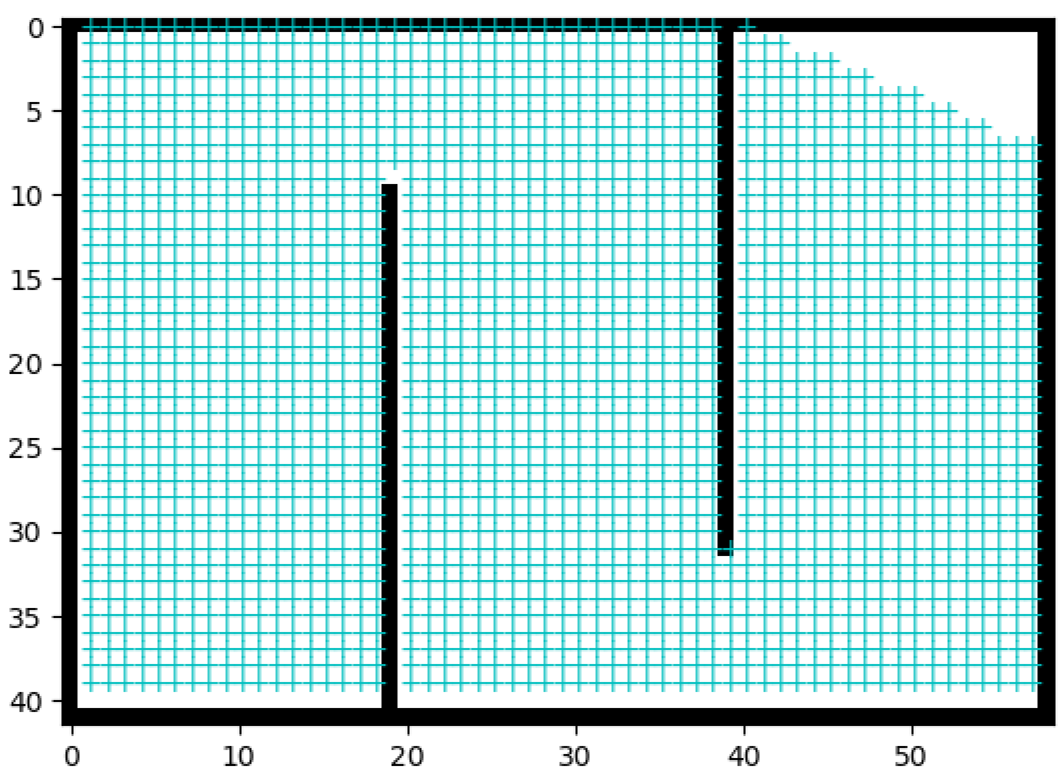
```python
import math
from itertools import islice

class Maze8(MazeD):
    def get(self, node, default):
        (r, c) = node
        rmax = len(self.maze_lines)
        cmax = len(self.maze_lines[0])
        m_row = self.maze_lines[r]
        possible_nbrs = [
            ((r, c-1), 1),
            ((r, c+1), 1),
```

```
            ((r-1, c), 1),
            ((r+1, c), 1),
            ((r-1, c-1), math.sqrt(2)),
            ((r-1, c+1), math.sqrt(2)),
            ((r+1, c-1), math.sqrt(2)),
            ((r+1, c+1), math.sqrt(2))
        ]
        free_nbrs = []
        for (ri, ci), dist in possible_nbrs:
            if (ri >= 0 and ci >= 0 and ri < rmax and ci < cmax
                   and self.maze_lines[ri][ci] == self.FREEPATH):
                free_nbrs.append(((ri, ci), dist))
        return free_nbrs if len(free_nbrs) else default

    def _plot_path(self, path, char='+', color='c'):
        return [self.ax.text(c-0.5, r-0.5, char, color=color)
               for (r, c) in path]

    def plot_path(self, path, **kw):
        self.plot_maze()
        return self._plot_path(path, **kw)

    def animate(self, path):
        self.init_plots()
        anim = animation.FuncAnimation(self.fig, self._plot_path, frames=bat
                                       init_func=self.plot_maze, blit=True, r
        return anim

maze = Maze8(maze_str)
success, search_path, node2parent, node2dist = dijkstra(maze, (35, 9), (5, 5
#print(success, search_path)
assert success
maze.animate(search_path)
```

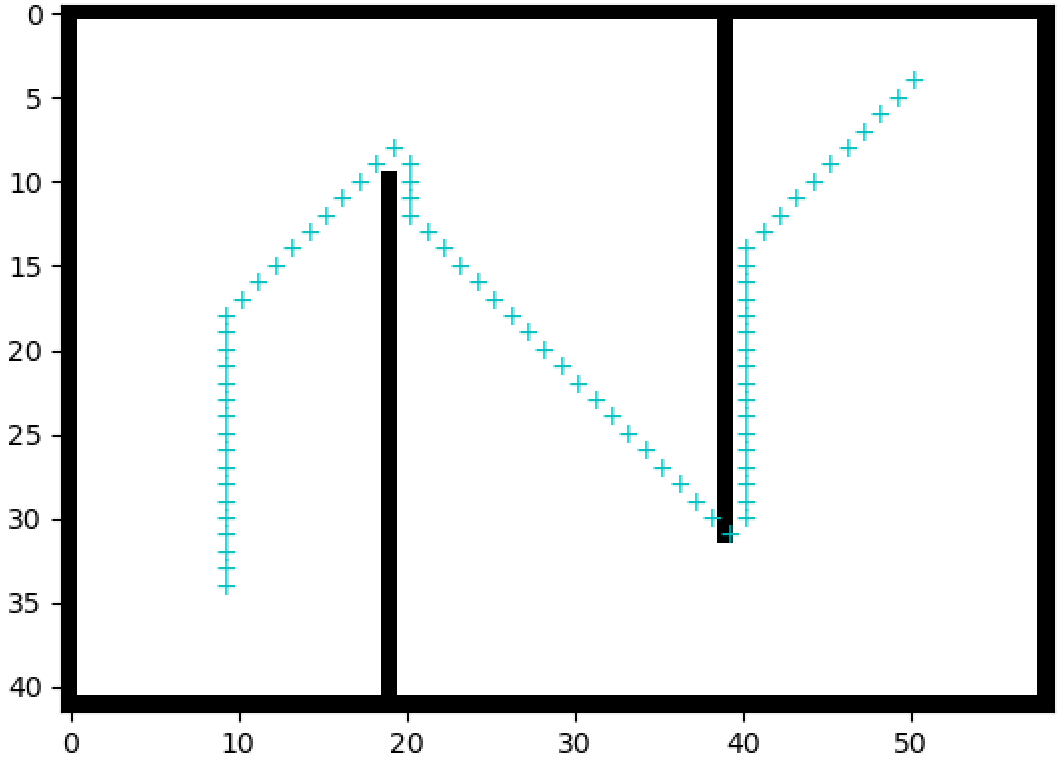
Out[17]:


0
5
10
15
20
25
30
35
40
0
10
20
30
40
50


Once
Loop
Reflect

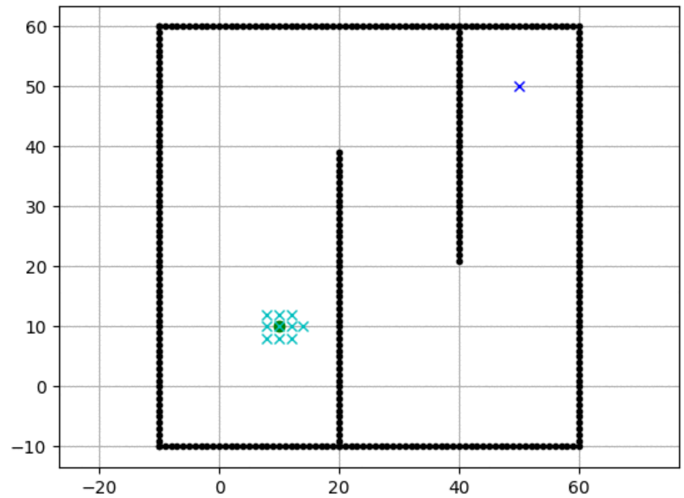

0
5
10
15
20
25
30
35
40
0
10
20
30
40
50

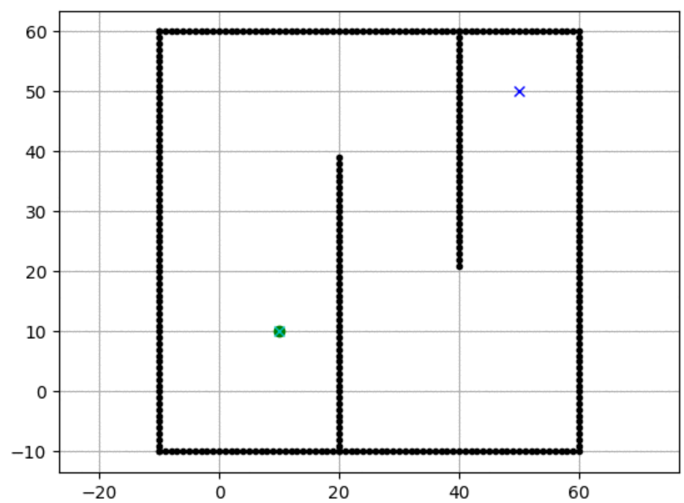
In [18]:
```python
path = backtrace_path(node2parent, (35, 9), (5, 50))
maze.init_plots(reinit=True)
maze.plot_path(path) # Draws the traced shortest path
plt.show()
```



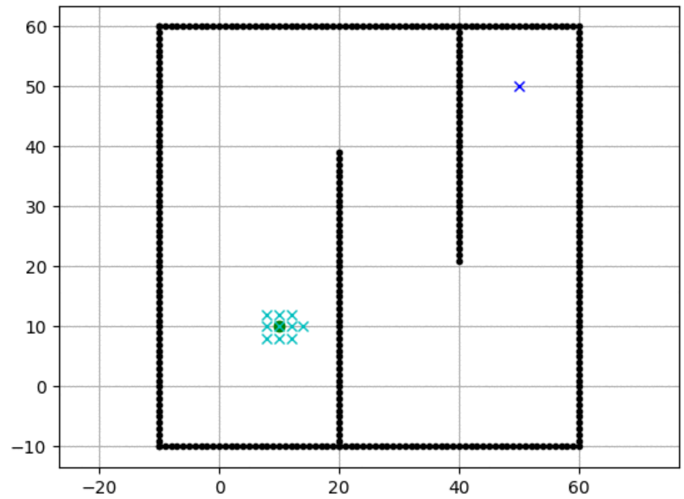
Search order in BFS vs DFS vs Dijkstra vs A*

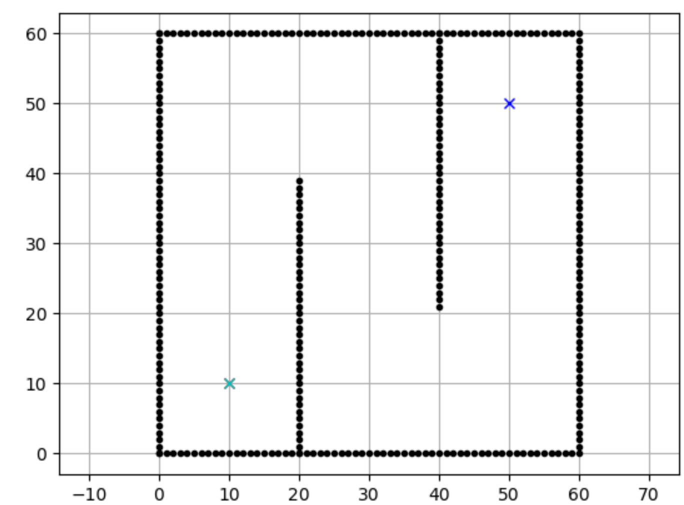
Breadth first search vs Depth first search





Breadth first search vs Dijkstra





Computational complexity of BFS

In [19]:
```
# 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)
```

Computational complexity of Dijkstra

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 dijkstra_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 = PriorityQueue() # Frontier of unvisited nodes as PriorityQueu
    frontier.put(start) # O(1)

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

        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| * ?)
```

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

Heap property

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

heap

Heap Insert

heap-insert

Heap runtimes

heap-runtimes