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 [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, 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 [4]:
```
# 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],
```

```
    [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 [5]:
```
# Exercise 1

# Write a function that converts a graph in adjacency list format to adjacer
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 [53]:
```
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 = [] # List for seen nodes (contains both frontier and dead states)
    # 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.append(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.append(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 [54]:

```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 [55]:
```python
graph = {
  's' : ['w', 'r'],
  'r' : ['v'],
  'w' : ['t', 'x'],
  'x' : ['y' ],
  't' : ['u'],
  'u' : ['y']
}

def dfs(graph, start, debug=False):
    seen = [] # List for seen nodes (contains both frontier and dead states)
    # 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 = []
    seen.append(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.append(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 [56]:
```python
# 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 [38]:
```
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):
    if self.fig is None:
        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_path(self, path):
    self.plot_maze()
    return [self.ax.text(c, r, '%d' % (i+1))
            for i, (r, c) in enumerate(path)]
```

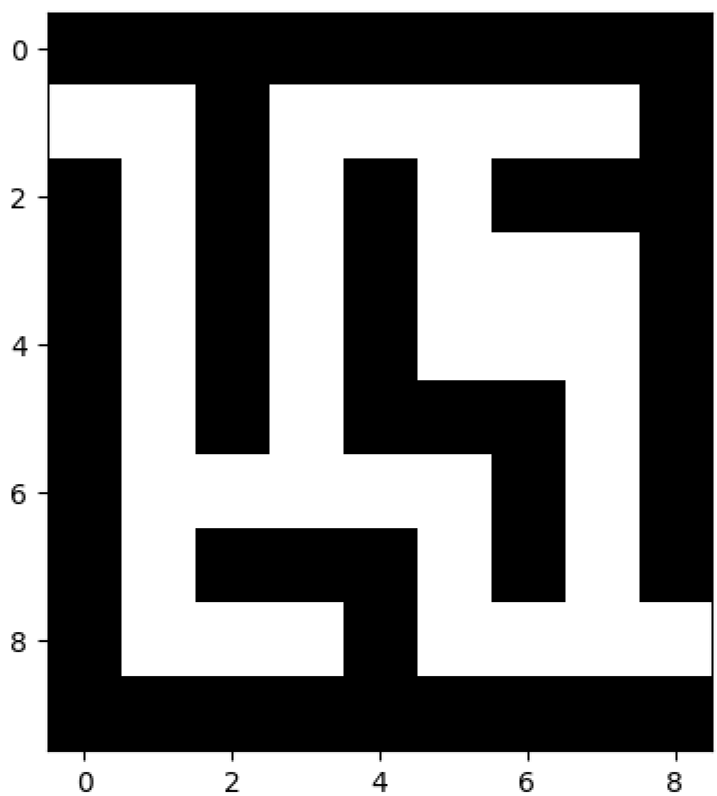
In [39]:
```
import matplotlib.pyplot as plt
import numpy as np
maze_str = \
"""
+++++++++
  +     +
+ + + +++
+ + +   +
+ + +   +
+ + +++ +
+     + +
+ +++ + +
+   +
+++++++++
"""

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))
```

```python
        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_path = plot_path
```

In [58]:
```python
maze = Maze(maze_str)
print(bfs(maze, (1, 0))) # prints the order of search all the searched nodes
maze.plot_maze()
```

```
([(1, 0), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (6, 2), (7, 1),
(6, 3), (8, 1), (6, 4), (5, 3), (8, 2), (6, 5), (4, 3), (8, 3), (7, 5), (3,
3), (8, 5), (2, 3), (8, 6), (1, 3), (8, 7), (1, 4), (8, 8), (7, 7), (1, 5),
(6, 7), (1, 6), (2, 5), (5, 7), (1, 7), (3, 5), (4, 7), (3, 6), (4, 5), (4,
6), (3, 7)], {(1, 0): 0, (1, 1): 1, (2, 1): 2, (3, 1): 3, (4, 1): 4, (5,
1): 5, (6, 1): 6, (6, 2): 7, (7, 1): 7, (6, 3): 8, (8, 1): 8, (6, 4): 9,
(5, 3): 9, (8, 2): 9, (6, 5): 10, (4, 3): 10, (8, 3): 10, (7, 5): 11, (3,
3): 11, (8, 5): 12, (2, 3): 12, (8, 6): 13, (1, 3): 13, (8, 7): 14, (1, 4):
14, (8, 8): 15, (7, 7): 15, (1, 5): 15, (6, 7): 16, (1, 6): 16, (2, 5): 16,
(5, 7): 17, (1, 7): 17, (3, 5): 17, (4, 7): 18, (3, 6): 18, (4, 5): 18, (4,
6): 19, (3, 7): 19})
```

Out[58]: <matplotlib.image.AxesImage at 0x7f84fc9fe710>

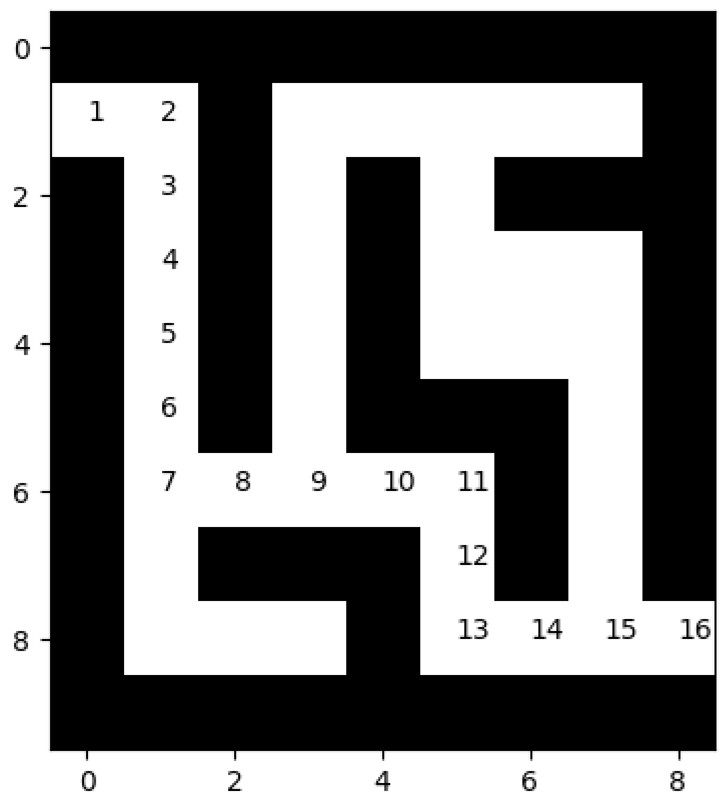


In [92]:
```python
def bfs_path(graph, start, goal):
    """
    Returns success and node2parent

    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 [93]:

```
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

In [94]:
```
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)]
}


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 = [start] # List for seen nodes.
    # Frontier is the boundary between seen and unseen
    frontier = PriorityQueue() # Frontier of unvisited nodes as FIFO
    node2parent = {start : None} # Keep track of nearest parent for each noc
    node2dist = {start: 0} # Keep track of cost to arrive at each node
    frontier.put(PItem(0, start))
    i = 0
    while not frontier.empty():          # Creating loop to visit each node
        dist_m = frontier.get() # Get the closest 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
        if goal is not None and m == goal:
            return True, 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.append(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
        i += 1
    if goal is not None:
        return False, {}, node2dist
    else:
        return True, node2parent, node2dist
```

In [95]:

```
success, node2parent, node2dist = dijkstra(graph, 's', None)
print(success, node2parent, node2dist)
```

```
True {'s': None, 'x': 's', 'u': 'x', 'v': 'u', 'y': 'x'} {'s': 0, 'x': 5,
'u': 8, 'v': 11, 'y': 7}
```

In [98]:

```
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, 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)}
```