**Appendix**

**A.1.1. Problem 2 – State Space**

**import** copy  
  
**class** **MazeGrid**():  
 """Class to define our state space and grid maze problem, custizable."""  
   
 **def** **\_\_init\_\_**(self, grid\_x\_size, grid\_y\_size, obstacle\_indices, start\_state, goal\_state):  
 self.grid\_x\_size = grid\_x\_size;  
 self.grid\_y\_size = grid\_y\_size;  
   
 # Building the grid indices (without obstacles for now)  
 self.grid\_indices = [(x, y) **for** x **in** range(grid\_x\_size) **for** y **in** range(grid\_y\_size)]  
 self.obstacle\_indices = obstacle\_indices;  
   
 # Building the grid as a dictionary of indices to a text value

# (either "\_" or a block if obstacle)  
 self.build\_grid()  
   
 self.start\_state = start\_state  
 self.goal\_state = goal\_state  
   
 self.grid[start\_state] = 'A'  
 self.grid[goal\_state] = 'B'

**def** **build\_grid**(self):  
 """Function to build the grid once the dimensions and obstacles are defined"""  
 self.grid = {}  
 **for** entry **in** self.grid\_indices:  
 self.grid[entry] = '\_'  
 **if** entry **in** self.obstacle\_indices:  
 self.grid[entry] = '\u2588'

**def** **print\_grid**(self):  
 """Function to display the grid. Not the prettiest, but does its job."""  
 **for** y **in** range(self.grid\_y\_size - 1, -1, -1):  
 **for** x **in** range(self.grid\_x\_size):  
 coordinate = (x, y)  
 **if** (x == self.grid\_x\_size - 1):  
 print(self.grid[coordinate])  
 **else**:  
 print(self.grid[coordinate], end = ' ')

# Next Page #

**def** **find\_neighbors**(self, state):  
 """Function to find neighboring states.  
 We don't define specific actions, so this is effectively our transition function.  
   
 Args:  
 state: The current state, (x, y), of which we need to returnvalid neighbors

(not obstacles).  
   
 Returns:  
 neighbors: A list of accessible neighbors from state, this is a lit of (x, y)

coordinates.  
 """  
 neighbor\_transforms = [(-1, 0), (0, -1), (1, 0), (0, 1)]  
 neighbors = []  
 **for** transform **in** neighbor\_transforms:  
 new\_x = state[0] + transform[0]  
 new\_y = state[1] + transform[1]  
 **if** ((0 <= new\_x) **and** (new\_x < self.grid\_x\_size) **and**   
 (0 <= new\_y) **and** (new\_y < self.grid\_y\_size) **and**   
 ((new\_x, new\_y) **not** **in** self.obstacle\_indices)):  
 neighbors.append((new\_x, new\_y))  
   
 **return** neighbors

**def** **print\_path**(self, states):  
 """Function to add x's on the grid along the set of states defined by the input.  
 Note that we don't check if any of those states are obstacles, it is assumed that the

given path is valid.  
   
 Args:  
 states: A list of states, (x, y), travelled along the grid.  
 """  
 grid\_copy = copy.deepcopy(self)  
   
 **for** state **in** states:  
 grid\_copy.grid[state] = 'x'  
   
 grid\_copy.print\_grid()

**A.1.2. Problem 2 – Depth-First Search**

We first implement DFS and BFS as building blocks (and learning experience) for A\*.

def dfs(problem):  
 """Depth-First Seach  
   
 Args:  
 problem: The problem we are running DFS on, includes start &

goal states, get\_neighbors, printing.  
   
 Returns:  
 expanded: If a path to the goal state of given problem is

found, return the path, otherwise None is returned.  
 """  
 frontier = [problem.start\_state]

frontier\_set = set({}) # Making a set of frontier because

# checking if state is in a set is

# faster than in a stack

frontier\_set.add(problem.start\_state)  
 explored = set({})  
   
 expanded = []  
 **while** (frontier):  
 curr\_state = frontier.pop()  
 explored.add(curr\_state)  
 expanded.append(curr\_state)  
   
 neighbors = problem.find\_neighbors(curr\_state)  
 **for** neighbor **in** neighbors:  
 **if** (neighbor == problem.goal\_state):  
 return expanded  
 **if** ((neighbor **not** **in** explored) **and**

(neighbor **not** **in** frontier\_set)):  
 frontier.append(neighbor)

frontier\_set.add(neighbor)  
   
 return None

**A.1.3. Problem 2 – Breadth-First Search**

**from** queue import Queue  
   
def bfs(problem):  
 """Depth-First Seach  
   
 Args:  
 problem: The problem we are running BFS on, includes start &

goal states, get\_neighbors, printing.

"""  
 frontier = Queue()  
 frontier.put(problem.start\_state)

frontier\_set = set({}) # Making a set of frontier because

# checking if state is in a set is

# faster than in a stack

frontier\_set.add(problem.start\_state)  
 explored = set({})  
   
 expanded = []  
 **while** (frontier):  
 curr\_state = frontier.get()  
 explored.add(curr\_state)  
 expanded.append(curr\_state)  
   
 neighbors = problem.find\_neighbors(curr\_state)  
 **for** neighbor **in** neighbors:  
 **if** (neighbor == problem.goal\_state):  
 return expanded  
 **if** ((neighbor **not** **in** explored) **and**

(neighbor **not** **in** frontier\_set)):   
 frontier.put(neighbor)

frontier\_set.add(neighbor)

return None

**A.1.4. Problem 2 – Heuristic Functions for Grid Search**

**import** math  
  
**def** **manhattan\_distance**(point1, point2):  
 """Computes the Manhattan distance between two points (x, y)."""  
 **return** abs(point1[0] - point2[0]) + abs(point1[1] - point2[1])  
  
**def** **euclidean\_distance**(point1, point2):  
 """Computes the Euclidean distance between two points (x, y)."""  
 **return** math.sqrt((point1[0] - point2[0]) \*\* 2) +

math.sqrt((point1[1] - point2[1]) \*\* 2)

**A.1.5. Problem 2 – A\* Search**