**Appendix**

**A.1.1. Problem 2 – Grid-Search State Space, Get Neighbors, & Utilities**

**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 = ' ')

**def** **is\_goal\_state**(self, state):

"""Function to check if a given state is the goal state."""

return state == self.goal\_state

# 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. Problems 2, 4, 5 – A\* Search**

**from** queue **import** PriorityQueue  
  
**def** **a\_star**(problem, heuristic):  
 """A\* Seach  
   
 Args:  
 problem: The problem we are running A\* on, includes start & goal states,

get\_neighbors, printing.  
 """  
 frontier = PriorityQueue()  
 frontier.put((heuristic(problem.start\_state, problem.goal\_state),

id(problem.start\_state), problem.start\_state))  
 explored = [] # set({})  
 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(id(problem.start\_state)) # Dictionaries are unhashable and can't

# be added to Hashset, but can add id  
   
 expanded = []  
 **while** (frontier):  
 queue\_entry = frontier.get()   
 curr\_state = queue\_entry[2]  
 explored.append(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** (id(neighbor) **not** **in** frontier\_set)):  
 frontier.put((heuristic(neighbor, problem.goal\_state), id(neighbor),

neighbor))  
 frontier\_set.add(id(neighbor))

**A.1.6. Problem 4 – N-Puzzle State Space, Get Neighbors, & Utilities**

**class** **NPuzzle**():  
 """Class to define our state space and n-puzzle problem, custizable."""  
 puzzle\_x\_size = 1  
 puzzle\_y\_size = 1  
   
 # STATE DEFINITION: Here, a state is (configuration, gap\_index) where

# configuration is the puzzle arrangement and gap\_index is the

# index of the empty block (having it separately makes things

# easier).  
 start\_state = ({}, (0, 0))  
 goal\_state = ({}, (0, 0))  
   
   
 **def** **\_\_init\_\_**(self, puzzle\_x\_size, puzzle\_y\_size, start\_state, goal\_state):  
 self.puzzle\_x\_size = puzzle\_x\_size  
 self.puzzle\_y\_size = puzzle\_y\_size  
   
 self.start\_state = start\_state  
 self.goal\_state = goal\_state  
   
 **def** **print\_puzzle**(self, state):  
 """Function to display the puzzle in the state passed in.

Not the prettiest, but does its job.

"""  
 configuration = state[0]  
 **for** y **in** range(self.puzzle\_y\_size - 1, -1, -1):  
 **for** x **in** range(self.puzzle\_x\_size):  
 coordinate = (x, y)  
 **if** (x == self.puzzle\_x\_size - 1):  
 print(configuration[coordinate])  
 **else**:  
 print(configuration[coordinate], end = ' ')

**def** **is\_goal\_state**(self, state):

"""Function to check if a given state is the goal state."""

return state == self.goal\_state

# 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, {...} of which we need to return valid

neighbors (other n-puzzle configurations).  
   
 Returns:  
 neighbors: A list of accessible neighbors from state, this is a list

dictionaries, which are puzzle configs.  
 """  
 configuration = state[0]  
 gap\_index = state[1]  
   
 transforms = [[-1, 0], [0, -1], [1, 0], [0, 1]]  
 neighbors = []  
 **for** transform **in** transforms:  
 new\_x = gap\_index[0] + transform[0]  
 new\_y = gap\_index[1] + transform[1]  
 **if** (0 <= new\_x **and** new\_x < self.puzzle\_x\_size **and**  
 0 <= new\_y **and** new\_y < self.puzzle\_y\_size):  
 # This is the index of the piece we're moving into the empty space

# (gap\_index)  
 neighbor\_index = (new\_x, new\_y)  
   
 # We make a copy of the current configuration and switch the two

# numbers  
 configuration\_copy = configuration.copy()  
 gap\_symbol = configuration\_copy[gap\_index]  
 configuration\_copy[gap\_index] = configuration\_copy[neighbor\_index]  
 configuration\_copy[neighbor\_index] = gap\_symbol  
   
 # We append this new configuration to our set of neighbors, note that

# neighbor\_index is now the gap index  
 neighbors.append((configuration\_copy, neighbor\_index))   
   
 **return** neighbors

**A.1.6. Problem 4 – Heuristic Functions**

**import** math  
   
**def** **count\_num\_misplaced\_tiles**(curr\_state, goal\_state):  
 curr\_configuration = curr\_state[0]  
 goal\_configuration = goal\_state[0]  
 count = 0  
 **for** index **in** curr\_configuration:  
 **if** (curr\_configuration[index] != goal\_configuration[index]) **and**

(curr\_configuration[index] != '\_'):  
 count += 1  
 **return** count  
   
**def** **count\_sum\_misplaced\_distances**(curr\_state, goal\_state):  
 curr\_configuration = curr\_state[0]  
 goal\_configuration = goal\_state[0]  
 total\_misplaced\_distances = 0  
 **for** index **in** curr\_configuration:  
 **if** (curr\_configuration[index] != '\_'):  
 index\_of\_same\_num\_in\_goal = find\_key\_from\_value(goal\_configuration,

curr\_configuration[index])  
 current\_misplaced\_distance = manhattan\_distance(index, index\_of\_same\_num\_in\_goal)  
 total\_misplaced\_distances += current\_misplaced\_distance  
   
 **return** total\_misplaced\_distances

**def** **find\_key\_from\_value**(dictionary, search\_value):  
 """A hacky method to find the key corresponding to a given value in a dictionary.  
 Which is not at all how a HashMap is supposed to be used.

Actually it's literally the exact opposite, but we have to do it because of our state

space definitions.  
 """  
 **for** index **in** dictionary:  
 **if** dictionary[index] == search\_value:  
 **return** index

**A.1.7. Problem 5 – Color Geometry State Space, Get Neighbors, & Utilities**

**import** math  
**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**from** shapely.geometry **import** Polygon  
   
**class** **ColorGeometry**():  
 """Class to define our state space and color geometry problem, custizable."""  
   
 # STATE DEFINITION: Here, we represent a state as a tuple (vertices,

shapes\_available)  
   
   
 # **Note:** We can have the starting state be any single state provided that all

# shapes are part of the solution, which is the case here.  
 start\_state = ([], {})  
   
 **def** **\_\_init\_\_**(self):  
 self.shapes = {'square' : ([(0, 0), (0, 4), (4, 4), (4, 0)], 'red'),  
 'house' : ([(0, 0), (0, 2 \* math.sqrt(2)),

(math.sqrt(2), 3 \* math.sqrt(2)),

(2 \* math.sqrt(2), 2 \* math.sqrt(2)),

(2 \* math.sqrt(2), 0)], 'blue'),  
 'triangle\_yellow' : ([(0, 0), (math.sqrt(2), math.sqrt(2)),

(2 \* math.sqrt(2), 0)], 'yellow'),  
 'triangle\_purple' : ([(0, 0), (2, 2), (4, 0)], 'purple'),  
 'triangle\_green' : ([(0, 0),

(2 \* math.sqrt(2), 2 \* math.sqrt(2)),

(4 \* math.sqrt(2), 0)], 'green'),  
 'triangle\_orange' : ([(0, 0),

(2 \* math.sqrt(2), 2 \* math.sqrt(2)),

(4 \* math.sqrt(2), 0)], 'orange')  
 }  
   
 self.start\_state = (self.shapes[BLUE][0],

{BLUE, YELLOW, PURPLE, GREEN, ORANGE})  
   
   
 **def** **is\_goal\_state**(self, state):  
 """Function to check if a given state is the goal state."""  
 # TODO(capuanomat): Tough  
 # vertices = state[0]  
 # return vertices == self.goal\_state  
 **pass**

# Next Page #

**def** **print\_state**(self, state):  
 """Function to display the shape. Quite pretty (subjective)."""  
 # Extracting the shape and color information  
 coordinates = state[0]  
 color = state[1]  
 x = [x **for** (x, \_) **in** coordinates]  
 y = [y **for** (\_, y) **in** coordinates]  
   
 # Creating Figure and Specifying parameters  
 plt.figure(figsize=(8, 8))  
 plt.axis('equal')  
   
 # Plotthing the geometry and displaying the plot  
 plt.fill(x, y, color)  
 plt.show()  
   
   
 **def** **find\_angle**(self, v1, v2, v3):  
 """Function to find the angle between three points in Euclidean space,

with v2 at junction

"""  
 v1v2 = np.array(v1) - np.array(v2)  
 v3v2 = np.array(v3) - np.array(v2)  
   
 cosine\_angle = np.dot(v1v2, v3v2) /

(np.linalg.norm(v1v2) \* np.linalg.norm(v3v2))  
 angle = np.arccos(cosine\_angle)  
   
 **return** angle  
   
   
 **def** **rot\_matrix**(self, angle):  
 """Function to return the 2D rotation matrix given an angle \*in radians\*."""  
 **return** np.array([[math.cos(angle), math.sin(angle)],  
 [-math.sin(angle), math.cos(angle)]])  
   
   
 **def** **rotate\_shape\_if\_valid**(self, added\_vertices, rot, current\_vertices):  
 """Function to apply a rotation matrix to a given geometry, and then check

if it intersects with the existing shape.

"""  
 rotated = []  
 **for** vertex **in** added\_vertices:  
 rotated\_vertex = np.matmul(rot, vertex)  
 rotated.append(tuple(rotated\_vertex))  
   
 curr\_polygon = Polygon(current\_vertices)  
 rotated\_polygon = Polygon(rotated)  
   
 # .touches checks that the polygons touch at at least one point but that the

# interiors do not intersect  
 **if** curr\_polygon.touches(rotated\_polygon):  
 **return** rotated  
   
 **return** None

**def** **combine\_geometries**(self, vertices, rotated\_shape):  
 """Function to combine two polygons.  
 The key challenge here is making sure that neighboring vertices remain next to

each other.  
   
 This approach actually does so INCORRECTLY, depending on when we find the

matching vertex  
 """  
 new\_vertices = []  
   
 **for** vertex **in** vertices:  
 **if** (vertex **in** rotated\_shape):  
 **for** rot\_vertex **in** rotated\_shape:  
 new\_vertices.append(rot\_vertex)  
 **else**:  
 new\_vertices.append(vertex)  
   
 **return** new\_vertices  
   
   
 **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, a list of vertices of the current shape.  
   
 Returns:  
 neighbors: A list of lists of vertices of neighboring shapes.  
 """  
 vertices = state[0]  
 shapes\_available = state[1]  
   
 neighbors = []  
   
 **for** shape **in** shapes\_available:  
 # The indices of the shape we are adding are:  
 added\_shape = self.shapes[shape][0].copy()  
 num\_added\_vertices = len(added\_shape)  
   
 # The remaining shapes we could use to add to the geometry (in later

# steps):  
 remaining\_shapes\_available = shapes\_available.copy()  
 remaining\_shapes\_available.remove(shape)  
   
 **for** i **in** range(len(vertices)):  
 # First, find the current vertex and the vertices neighboring this

# vertex  
 vertex1 = vertices[i]  
 vertex1\_prev = vertices[(i - 1) % len(vertices)]  
 vertex1\_next = vertices[(i + 1) % len(vertices)]  
   
 # Second, loop through all vertices of the shape we are adding and see

# how they can be aligned  
 **for** j **in** range(num\_added\_vertices):  
 vertex2 = added\_shape[i]  
 vertex2\_prev = added\_shape[(j - 1) % num\_added\_vertices]  
 vertex2\_next = added\_shape[(j + 1) % num\_added\_vertices]  
   
 # Now, we have to align the two primary vertices we are focusing

# on, such that we only have to rotate to align edges  
 xdiff = vertex1[0] - vertex2[0]  
 ydiff = vertex1[1] - vertex2[1]  
 **for** k **in** range(num\_added\_vertices):  
 added\_shape[k] = (added\_shape[k][0] + xdiff,

added\_shape[k][1] + ydiff)  
   
 # There are four pairs of vertices to find the angle for and try

# aligning (two will fail due to intersections). Note that at this

# point, vertex1 = vertex2 since we just aligned them.  
 # Case 1: v1n, v1, v2n  
 theta1 = self.find\_angle(vertex1\_next, vertex1, vertex2\_next)  
 R1 = self.rot\_matrix(theta1)  
 rotated\_shape1 = self.rotate\_shape\_if\_valid(added\_shape, R1,

vertices)  
   
 # Case 2: v1n, v1, v2p  
 theta2 = self.find\_angle(vertex1\_next, vertex1, vertex2\_prev)  
 R2 = self.rot\_matrix(theta2)  
 rotated\_shape1 = self.rotate\_shape\_if\_valid(added\_shape, R3,

vertices)  
   
 # Case 3: v1p, v1, v2n  
 theta3 = self.find\_angle(vertex1\_prev, vertex1, vertex2\_next)  
 R3 = self.rot\_matrix(theta3)  
 rotated\_shape3 = self.rotate\_shape\_if\_valid(added\_shape, R3,

vertices)  
   
 # Case 4: v1p, v1, v2p  
 theta4 = self.find\_angle(vertex1\_prev, vertex1, vertex2\_prev)  
 R4 = self.rot\_matrix(theta4)  
 rotated\_shape4 = self.rotate\_shape\_if\_valid(added\_shape, R4,

vertices)  
   
 rotated\_shapes = [rotated\_shape1, rotated\_shape2,

rotated\_shape3, rotated\_shape4]  
 # Finally, for the valid rotated shapes (do not overlap with the

# current geometry) we combine them with it  
 **for** rotated\_shape **in** rotated\_shapes:  
 **if** (rotated\_shape):  
 new\_vertices = self.combine\_geometries(vertices,

rotated\_shape)  
 neighbors.append((new\_vertices,

remaining\_shapes\_available))

**return** neighbors