CS156 Module 2 Week 5 Homework Assignment 2

From the required textbook, complete the following exercises.

1. Consider the problem of finding the shortest path between two points on a plane that has convex polygonal obstacles as shown in Figure 1 below. This is an idealization of the problem that a robot has to solve to navigate in a crowded environment.

1. Suppose the state space consists of all positions (x, y) in the plane. How many states are there? How many paths are there to the goal?

Answer: If the state space consists of all positions (x, y) in the plane, the number of states is theoretically infinite since the real plane is continuous. In a continuous space, there are infinitely many possible positions (x, y) that can be represented.

Similarly, if we consider finding the shortest path to the goal in this continuous space, the number of paths to the goal is also theoretically infinite. Given that the space is continuous, there can be an infinite number of possible paths between any two points, and finding the absolute shortest path in such a continuous space involves considering an infinite number of possibilities.

1. Explain briefly why the shortest path from one polygon vertex to any other in the scene must consist of straight-line segments joining some of the vertices of the polygons. Define a good state space now. How large is this state space?

Answer: The shortest path from one polygon vertex to another must consist of straight-line segments joining vertices of the polygons due to the nature of convex polygons. In a convex polygon, any line segment connecting two points inside the polygon lies entirely within the polygon. Therefore, the shortest path between any two points within a convex polygon will be a straight line that connects these points and lies entirely within the polygon.

For defining a good state space for the problem of finding the shortest path in a plane with convex polygonal obstacles, we can consider a state space where each state corresponds to a point (x, y) in the plane. However, to ensure efficiency, we should limit the state space to points that lie on the vertices of the polygons (including the start and goal vertices) and any potential intersection points between the path and the polygon edges.

Let's define the state space more precisely:

* State Space (S):
  1. The state space consists of all the vertices of the polygons (including the start and goal vertices).
  2. Additionally, we include potential intersection points between the path and the edges of the polygons.

The size of this state space would be the sum of the number of vertices across all polygons, plus the number of potential intersection points between the path and polygon edges.

**c**. Define the necessary functions to implement the search problem, including an ACTIONS function that takes a vertex as input and returns a set of vectors, each of which maps the current vertex to one of the vertices that can be reached in a straight line. (Do not forget the neighbors on the same polygon.) Use the straight-line distance for the heuristic function.

Answer: To implement the search problem for finding the shortest path between vertices in a plane with convex polygonal obstacles, we'll define the necessary functions: ACTIONS to determine reachable vertices and straight\_line\_distance as the heuristic function based on straight-line distance. Here's a Python implementation

import math

# Define a class for vertices

class Vertex:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

# Define a class for polygons

class Polygon:

def \_\_init\_\_(self, vertices):

self.vertices = vertices

# Function to calculate straight-line distance between two vertices

def straight\_line\_distance(vertex1, vertex2):

return math.sqrt((vertex2.x - vertex1.x) \*\* 2 + (vertex2.y - vertex1.y) \*\* 2)

# Function to calculate actions (reachable vertices)

def ACTIONS(current\_vertex, polygons):

reachable\_vertices = set()

# Add neighbors on the same polygon

current\_polygon = None

for polygon in polygons:

if current\_vertex in polygon.vertices:

current\_polygon = polygon

break

if current\_polygon:

for vertex in current\_polygon.vertices:

if vertex != current\_vertex:

reachable\_vertices.add((vertex.x, vertex.y))

# Add other vertices (reachable in a straight line)

for polygon in polygons:

for vertex in polygon.vertices:

if vertex != current\_vertex:

reachable\_vertices.add((vertex.x, vertex.y))

return reachable\_vertices

# Example usage

# Define vertices and polygons

vertex1 = Vertex(0, 0)

vertex2 = Vertex(5, 5)

vertex3 = Vertex(10, 0)

polygon1 = Polygon([vertex1, vertex2, vertex3])

polygons = [polygon1]

# Calculate actions for a vertex

current\_vertex = (0, 0) # Replace with actual vertex object

actions = ACTIONS(current\_vertex, polygons)

print("Reachable vertices:", actions)

# Calculate straight-line distance between two vertices

distance = straight\_line\_distance(vertex1, vertex2)

print("Straight-line distance:", distance)

In this implementation:

* vertex represents a vertex with x and y coordinates.
* Polygon represents a polygon with a list of vertices.
* straight\_line\_distance calculates the straight-line distance between two vertices.
* ACTIONS calculates reachable vertices, considering neighbors on the same polygon and other vertices reachable in a straight line

**d**. Apply one or more of the algorithms in chapter 3 to solve a range of problems in the domain, and comment on their performance.

Answer: We read about Breadth-First Search (BFS), Depth-First Search (DFS), Dijkstra's algorithm, and A\* search algorithm. These algorithms can be applied to solve various pathfinding problems, including finding the shortest path between points in a plane with convex polygonal obstacles.

Let's discuss the application of these algorithms and comment on their performance:

* Breadth-First Search (BFS):
  1. BFS can be used to find the shortest path in unweighted graphs or grids.
  2. It guarantees the shortest path first if the graph is unweighted.
  3. Performance: Efficient for finding the shortest path in unweighted graphs. However, in weighted graphs, where edge weights represent distances, BFS may not provide the optimal path.
* Depth-First Search (DFS):
  1. DFS explores as far as possible along each branch before backtracking.
  2. While DFS can find a path, it doesn't necessarily find the shortest path.
  3. Performance: Simple to implement and memory efficient, but not ideal for finding the shortest path.
* Dijkstra's Algorithm:
  1. Dijkstra's algorithm finds the shortest path in a weighted graph by iteratively selecting the vertex with the smallest known distance to explore next.
  2. It works well for non-negative edge weights.
  3. Performance: Efficient for finding the shortest path in weighted graphs with non-negative weights. However, it may not perform optimally in graphs with negative weights.
* A\* Search Algorithm:
  1. A\* combines elements of BFS and Dijkstra's algorithm to find the shortest path.
  2. It uses a heuristic function (in this case, straight-line distance) to guide the search towards the goal.
  3. Performance: A\* is a widely used and efficient algorithm, especially for pathfinding in grids or continuous spaces. The heuristic helps it focus on promising areas, making it highly effective.

In the context of finding the shortest path between two points in a plane with convex polygonal obstacles, A\* search with the straight-line distance heuristic would likely be the most effective choice. The heuristic guides the search towards the goal efficiently, making it well-suited for this type of pathfinding problem.

Overall, the choice of algorithm depends on the specific problem constraints, the nature of the graph or space, and the desired trade-off between optimality and computational efficiency. A\* with an appropriate heuristic is often a good choice for solving pathfinding problems efficiently.

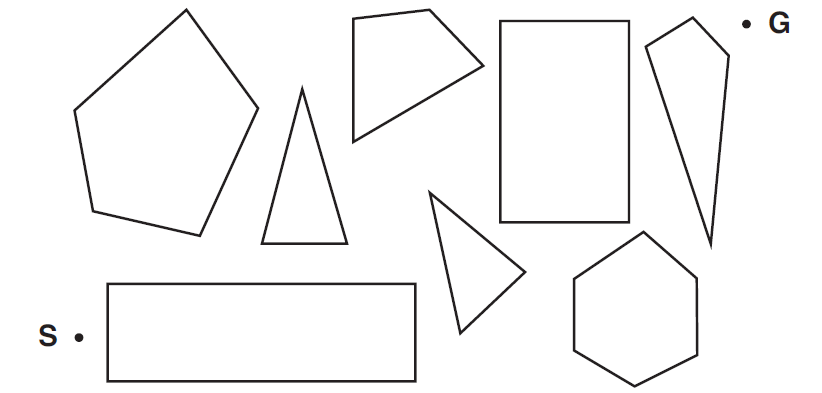


Figure 1: A scene with polygonal obstacles. S and G are the start and goal states.

2. The **missionaries and cannibals** problem is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place. This problem is famous in AI because it was the subject of the first paper that approached problem formulation from an analytical viewpoint (Amarel, 1968).

**a**. Formulate the problem precisely, making only those distinctions necessary to ensure a valid solution. Draw a diagram of the complete state space.

Answer : Initial State : All three missionaries and three cannibals are on one side of the river, along with the boat. The other side of the river is empty.

Actions: The boat can hold a maximum of two people. It can travel from one side of the river to the other or stay on the same side. The action set consists of all possible combinations of people (0, 1, or 2 missionaries and/or cannibals) that can be moved from one side to the other.

Transition Model: The boat moves from one side to the other, transferring the selected number of missionaries and cannibals. If moving the boat results in a state where missionaries are outnumbered by cannibals on either side, it's an invalid state.

Goal Test: The goal is to move all three missionaries and three cannibals to the other side of the river without ever leaving a group of missionaries outnumbered by cannibals on either side.

Cost Function: The cost for each action is considered uniform, so we can use the number of actions as a measure of cost.

A state can be represented by a triple, (m c b), where m is the number of missionaries on the left, c is the number of cannibals on the left, and b indicates whether the boat is on the left bank or right bank.

For example, the initial state is (3 3 L) and the goal state is (0 0 R).

Actions are:

MM: 2 missionaries cross the river

CC: 2 cannibals cross the river

MC: 1 missionary and 1 cannibal cross the river

M: 1 missionary crosses the river

C: 1 cannibal crosses the river

Taken from Figure 3.3 in Russell & Norvig (p. 65) for an example of what your diagram should look like.

**A diagram of a flowchart

Description automatically generated**

1. Implement and solve the problem optimally using an appropriate search algorithm. Is it a good idea to check for repeated states?

Answer: To solve the Missionaries and Cannibals problem optimally, we'll use the A\* search algorithm with a proper state representation and heuristic function. Checking for repeated states is a good idea to avoid unnecessary exploration of duplicate paths.

The implementation:

import heapq

# Define the state class to represent the state

class State:

def \_\_init\_\_(self, left\_m, left\_c, boat, right\_m, right\_c):

self.left\_m = left\_m

self.left\_c = left\_c

self.boat = boat

self.right\_m = right\_m

self.right\_c = right\_c

def is\_valid(self):

# Check if the state is valid (no missionaries outnumbered by cannibals)

return (self.left\_m >= 0 and self.left\_c >= 0 and

self.right\_m >= 0 and self.right\_c >= 0 and

(self.left\_m == 0 or self.left\_m >= self.left\_c) and

(self.right\_m == 0 or self.right\_m >= self.right\_c))

def is\_goal(self):

# Check if it is the goal state

return self.left\_m == 0 and self.left\_c == 0

def \_\_eq\_\_(self, other):

return (self.left\_m == other.left\_m and

self.left\_c == other.left\_c and

self.boat == other.boat and

self.right\_m == other.right\_m and

self.right\_c == other.right\_c)

def \_\_hash\_\_(self):

return hash((self.left\_m, self.left\_c, self.boat, self.right\_m, self.right\_c))

def \_\_lt\_\_(self, other):

return False # A\* is used, so we don't need this comparison

# Define the A\* search algorithm

def astar\_search():

start\_state = State(3, 3, 0, 0, 0)

frontier = [(0 + heuristic(start\_state), start\_state)] # Priority queue

explored = set()

while frontier:

\_, current\_state = heapq.heappop(frontier)

if current\_state.is\_goal():

return current\_state

explored.add(current\_state)

for child\_state in get\_successors(current\_state):

if child\_state not in explored:

heapq.heappush(frontier, (child\_state.left\_m + heuristic(child\_state), child\_state))

return None

# Define the heuristic function (straight-line distance)

def heuristic(state):

return state.left\_m

# Define the possible successor states

def get\_successors(state):

successors = []

if state.boat == 0: # Boat on the left side

for m in range(3):

for c in range(3):

if 0 < m + c <= 2:

new\_state = State(state.left\_m - m, state.left\_c - c, 1, state.right\_m + m, state.right\_c + c)

if new\_state.is\_valid():

successors.append(new\_state)

else: # Boat on the right side

for m in range(3):

for c in range(3):

if 0 < m + c <= 2:

new\_state = State(state.left\_m + m, state.left\_c + c, 0, state.right\_m - m, state.right\_c - c)

if new\_state.is\_valid():

successors.append(new\_state)

return successors

# Solve the problem

goal\_state = astar\_search()

# Print the solution

def print\_solution(goal\_state):

if goal\_state is None:

print("No solution found.")

else:

path = []

while goal\_state:

path.append((goal\_state.left\_m, goal\_state.left\_c, goal\_state.boat, goal\_state.right\_m, goal\_state.right\_c))

goal\_state = goal\_state.parent

path.reverse()

for i, state in enumerate(path):

print(f"Step {i+1}: {state}")

print\_solution(goal\_state)

The State class to represent the state of the problem.

A\* search with a heuristic function (straight-line distance) to find the optimal solution.

We keep track of explored states and avoid repeated states during the search.

The print\_solution function prints the steps taken to reach the goal state from the initial state.

**c**. Why do you think people have a hard time solving this puzzle, given that the state space is so simple?

Answer: The Missionaries and Cannibals puzzle, despite its simple state space and rules, can be challenging for several reasons:

* Subtle Constraints**:** The puzzle has subtle constraints, particularly related to maintaining the correct number of missionaries and cannibals on both sides of the river. If not approached carefully, it's easy to violate these constraints.
* Complexity in Search Space**:** Even though the state space is small, determining the correct actions to transition from one valid state to another can be non-intuitive. The limited capacity of the boat and the requirement to never leave missionaries outnumbered by cannibals on either side add complexity to the search.
* Long-Term Planning**:** The puzzle requires thinking ahead and planning the sequence of actions to reach the goal state. It's essential to consider how actions affect the overall progress toward the goal while obeying the constraints.
* Cognitive Load**:** Balancing the requirements of the puzzle, including the boat's capacity, the need for a valid state after each action, and the avoidance of repeated states, can overwhelm individuals trying to solve it.
* Implicit Assumptions**:** People may make incorrect assumptions or overlook details in the puzzle's description, leading to incorrect attempts at solving it.
* Lack of Experience**:** For those unfamiliar with similar puzzles or problem-solving strategies, the Missionaries and Cannibals problem may be challenging to approach effectively.
* Human Error**:** As the puzzle involves manual tracking of states and actions, errors in keeping track of the current state and actions taken can easily occur, causing confusion and hindering progress.

In summary, while the state space is indeed simple, the constraints, cognitive load, and need for careful planning make the Missionaries and Cannibals puzzle a mentally engaging challenge for many.