University of Puerto Rico
Mayagüez Campus
Electrical & Computer Engineering Department

Ex. 3.9 & 3.11 Report

- Group A -
Daniel Mestres Piñero
Natanael Santiago Morales
Leonel Osoria Toledo
ICOM5015 - 001D
March 10, 2023

Exercises

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

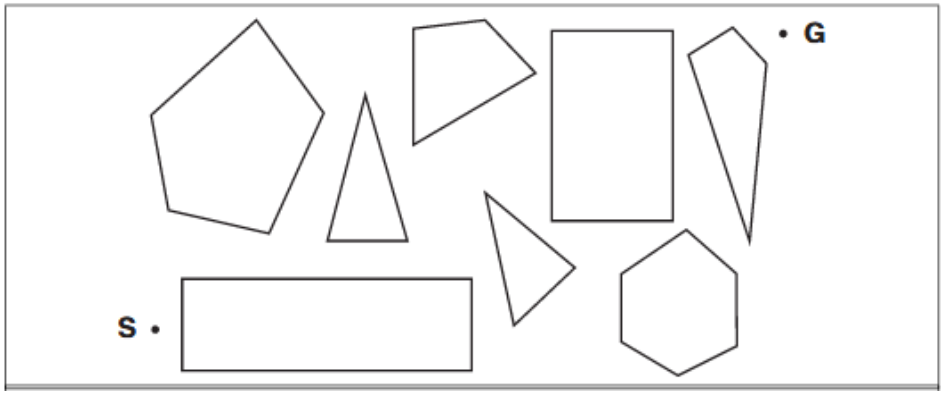

Figure 1: A scene with polygonal obstacles. S and G are the start and goal states. [1, page 114]

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?

2. 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?

3. Define the necessary functions to implement the search problem, including a 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.

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

3.11 - The 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.

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

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

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

Procedure

3.9.1 - If the state space consists of all the positions (x,y) in the plane, the number of states and paths to the goal is infinite.

3.9.2 - The shortest path from one polygon vertex to any other in the scene must consist of straight-line segments as the shortest distance between two points is a straight line. If the state space consists of all the polygonal vertices, with the path from one another being a straight line, the state space is the set of all vertices and start and goal state, adding up to 35 states in total.

3.9.3 - The graph problem implementation [2] was used to demonstrate the necessary functions to implement the search problem, which can be seen in the figure below:

```
class GraphProblem(Problem):
    """The problem of searching a graph from one node to another."""

    def __init__(self, initial, goal, graph):
        super().__init__(initial, goal)
        self.graph = graph

    def actions(self, A):
        """The actions at a graph node are just its neighbors."""
        return list(self.graph.get(A).keys())

    def result(self, state, action):
        """The result of going to a neighbor is just that neighbor."""
        return action

    def path_cost(self, cost_so_far, A, action, B):
        return cost_so_far + (self.graph.get(A, B) or np.inf)

    def find_min_edge(self):
        """Find minimum value of edges."""
        m = np.inf
        for d in self.graph.graph_dict.values():
            local_min = min(d.values())
            m = min(m, local_min)

        return m

    def h(self, node):
        """h function is straight-line distance from a node's state to goal."""
        locs = getattr(self.graph, 'locations', None)
        if locs:
            if type(node) is str:
                return int(distance(locs[node], locs[self.goal]))

            return int(distance(locs[node.state], locs[self.goal]))
        else:
            return np.inf
```

Figure 2: Graph Problem Implementation [2]

The actions function takes a node A and returns a list of its immediate neighbors, in other words the possible paths to take from that node. The function path cost adds up the already taken paths cost and the cost from Node A to Node B. The function find_min_edge returns the immediate node with the least straight-line path cost and the function h implements the heuristic function utilizing the straight-line distance from a Node to the goal.

3.9.4 - Four algorithms, these being Breadth-first tree search, Depth-first graph search, Best first uniform cost search, and Depth limited search were used to observe on their performance with a similar problem to the Figure 3, where each possible move is a straight line from one node to another. The data provided by the aima-python repository was used to create the graph, seen in the figure below:

Figure 3: Visual representation of the nodes and their relation to each other

The following figures show the implementation of the 4 algorithms used, provided by the aima-python code repository [2].

```
def tree_breadth_search_for_vis(problem):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Don't worry about repeated paths to a state. [Figure 3.7]"""

    # we use these two variables at the time of visualisations
    iterations = 0
    all_node_colors = []
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    #Adding first node to the queue
    frontier = deque([Node(problem.initial)])

    node_colors[Node(problem.initial).state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    while frontier:
        #Popping first node of queue
        node = frontier.popleft()

        # modify the currently searching node to red
        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            # modify goal node to green after reaching the goal
            node_colors[node.state] = "green"
            iterations += 1
            all_node_colors.append(dict(node_colors))
            return(iterations, all_node_colors, node)

        frontier.extend(node.expand(problem))

        for n in node.expand(problem):
            node_colors[n.state] = "orange"
            iterations += 1
            all_node_colors.append(dict(node_colors))

        # modify the color of explored nodes to gray
        node_colors[node.state] = "gray"
        iterations += 1
        all_node_colors.append(dict(node_colors))

    return None
```

Figure 4: Breadth-first tree search [2]

```
def graph_search_for_vis(problem):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    If two paths reach a state, only use the first one. [Figure 3.7]"""
    # we use these two variables at the time of visualisations
    iterations = 0
    all_node_colors = []
    node_colors = {k : 'white' for k in problem.graph.nodes()}
    frontier = [(Node(problem.initial))]
    explored = set()

    # modify the color of frontier nodes to orange
    node_colors[Node(problem.initial).state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    while frontier:
        # Popping first node of stack
        node = frontier.pop()

        # modify the currently searching node to red
        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            # modify goal node to green after reaching the goal
            node_colors[node.state] = "green"
            iterations += 1
            all_node_colors.append(dict(node_colors))
            return(iterations, all_node_colors, node)

        explored.add(node.state)
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored and
                        child not in frontier)

        for n in frontier:
            # modify the color of frontier nodes to orange
            node_colors[n.state] = "orange"
            iterations += 1
            all_node_colors.append(dict(node_colors))

        # modify the color of explored nodes to gray
        node_colors[node.state] = "gray"
        iterations += 1
        all_node_colors.append(dict(node_colors))
    return None
```

Figure 5: Depth-first graph search [2]

```
def best_first_graph_search_for_vis(problem, f):
    """Search the nodes with the lowest f scores first.
    You specify the function f(node) that you want to minimize; for example,
    if f is a heuristic estimate to the goal, then we have greedy best
    first search; if f is node.depth then we have breadth-first search.
    There is a subtlety: the line "f = memoize(f, 'f')" means that the f
    values will be cached on the nodes as they are computed. So after doing
    a best first search you can examine the f values of the path returned."""

    # we use these two variables at the time of visualisations
    iterations = 0
    all_node_colors = []
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    f = memoize(f, 'f')
    node = Node(problem.initial)

    node_colors[node.state] = "red"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    if problem.goal_test(node.state):
        node_colors[node.state] = "green"
        iterations += 1
        all_node_colors.append(dict(node_colors))
        return(iterations, all_node_colors, node)

    frontier = PriorityQueue('min', f)
    frontier.append(node)

    node_colors[node.state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    explored = set()
    while frontier:
        node = frontier.pop()

        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            node_colors[node.state] = "green"
            iterations += 1
            all_node_colors.append(dict(node_colors))
            return(iterations, all_node_colors, node)
```

```
        explored.add(node.state)
        for child in node.expand(problem):
            if child.state not in explored and child not in frontier:
                frontier.append(child)
                node_colors[child.state] = "orange"
                iterations += 1
                all_node_colors.append(dict(node_colors))
            elif child in frontier:
                incumbent = frontier[child]
                if f(child) < incumbent:
                    del frontier[child]
                    frontier.append(child)
                    node_colors[child.state] = "orange"
                    iterations += 1
                    all_node_colors.append(dict(node_colors))

        node_colors[node.state] = "gray"
        iterations += 1
        all_node_colors.append(dict(node_colors))
    return None
```

Figure 6: Best-first uniform cost search [2]

```python
def depth_limited_search_graph(problem, limit = -1):
    '''
    Perform depth first search of graph g.
    if limit >= 0, that is the maximum depth of the search.
    '''
    # we use these two variables at the time of visualisations
    iterations = 0
    all_node_colors = []
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    frontier = [Node(problem.initial)]
    explored = set()

    cutoff_occurred = False
    node_colors[Node(problem.initial).state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    while frontier:
        # Popping first node of queue
        node = frontier.pop()

        # modify the currently searching node to red
        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            # modify goal node to green after reaching the goal
            node_colors[node.state] = "green"
            iterations += 1
            all_node_colors.append(dict(node_colors))
            return(iterations, all_node_colors, node)

        elif limit >= 0:
            cutoff_occurred = True
            limit += 1
            all_node_colors.pop()
            iterations -= 1
            node_colors[node.state] = "gray"


        explored.add(node.state)
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored and
                        child not in frontier)

        for n in frontier:
            limit -= 1
            # modify the color of frontier nodes to orange
            node_colors[n.state] = "orange"
            iterations += 1
            all_node_colors.append(dict(node_colors))

        # modify the color of explored nodes to gray
        node_colors[node.state] = "gray"
        iterations += 1
        all_node_colors.append(dict(node_colors))

    return 'cutoff' if cutoff_occurred else None


def depth_limited_search_for_vis(problem):
    """Search the deepest nodes in the search tree first."""
    iterations, all_node_colors, node = depth_limited_search_graph(problem)
    return(iterations, all_node_colors, node)
```

Figure 7: Depth limited search [2]

The four algorithms were used to move from the starting node “Arad” to the goal node “Bucharest”, the following table presents their amount of iterations taken to achieve their goal:

TABLE I
ALGORITHM PERFORMANCE COMPARISON

Algorithm	Iterations Needed
Breadth-first tree search	102
Depth-first graph search	40
Best first uniform cost search	41
Depth limited search	40

From table I, we can observe how the clear loser of the 4 algorithms is the Breadth-first tree search, the second is the Best first uniform cost search and tied for first are the Depth-first graph search and Depth limited search. Breadth-first tree search performs considerably worse than the others, taking more than double the iterations needed to reach the goal state.

3.11.1 - The state space is the number of missionaries and cannibals on the left side of the river and the position of the boat, represented by a 3 tuple of integers, composed of (M, C, B), where M = missionaries, C = Cannibals, and B = boat in left or right side of river. The following includes the initial state, possible actions and goal state of the problem:

- Initial State:
  - (3, 3, 0) - 3 missionaries and 3 cannibals on the left side of the river and the boat on the left side of the river.

- Possible Actions:
  - + (0, 1, 0) or (1, 0, 0) : One cannibal or missionary crossing from right to left. The boat is assigned 0 because it is on the left side.
  - + (0, 2, 0) or (2, 0, 0) : Two cannibals or missionaries crossing from right to left. The boat is assigned 0 because it is on the left side.
  - - (1, 0, 1) or (0, 1, 1) : One cannibal or missionary crossing from left to right. The boat is assigned 1 because it is on the right side.
  - - (2, 0, 1) or (0, 2, 1) : Two cannibals or missionaries crossing from left to right. The boat is assigned 1 because it is on the right side.
  - + (1, 1, 0) : One cannibal and one missionary crossing from right to left. The boat is assigned 0 because it is on the left side.

- (1, 1, 1) : One cannibal and one missionary crossing from left to right. The boat is assigned 1 because it is on the right side.

- Goal State:
  - (0, 0, 1) - Everyone crossed the river and boat on the right side of the river.

Given the established rules, $M \geq 0$, $C \geq 0$, and $M \geq C$ if $M > 0$.

3.11.2 - To implement and solve the problem, we chose the Breadth-First Search algorithm. The state space is represented in the same way as above and we need to check each state to see if it is valid, in other words if it does not cause cannibals to outnumber missionaries. Below is our implementation of the problem utilizing the BFS algorithm:

```
from queue import Queue
```

```
def is_valid_state(state):
    """Check if a state is valid."""
    M, C, B = state
    if 0 <= M <= 3 and 0 <= C <= 3 and 0 <= B <= 1:
        if M < C and M > 0:
            return False
        if M > C and M < 3:
            return False
        return True
    return False
```

```
def get_successors(state):
    """Return the valid successors of a state."""
    M, C, B = state
    successors = []
    if B == 0:
        for m in range(3):
            for c in range(3):
                if m + c >= 1 and m + c <= 2:
                    new_state = (M-m, C-c, 1)
                    if is_valid_state(new_state):
                        successors.append(new_state)
    else:
        for m in range(3):
            for c in range(3):
                if m + c >= 1 and m + c <= 2:
                    new_state = (M+m, C+c, 0)
                    if is_valid_state(new_state):
                        successors.append(new_state)
    return successors
```

```python
def bfs(initial_state, goal_state):
    """Apply Breadth-First Search to find the path from the initial state to the goal state."""
    frontier = Queue()
    frontier.put(initial_state)
    explored = set()
    parents = {initial_state: None}

    while not frontier.empty():
        current_state = frontier.get()

        for successor in get_successors(current_state):
            if successor not in explored:
                explored.add(successor)
                frontier.put(successor)
                parents[successor] = current_state

        if current_state == goal_state:
            # Reconstruct the path after getting to the goal state
            path = [current_state]
            while True:
                path.insert(0, parents[current_state])
                current_state = parents[current_state]
                if current_state == initial_state:
                    return path

    return None
```

```python
# Solve the problem
initial_state = (3, 3, 0)
goal_state = (0, 0, 1)
path = bfs(initial_state, goal_state)

if path is None:
    print("No solution found.")
else:
    print("Solution found with", len(path)-1, "moves:")
    for i in range(len(path)):
        print("Step", i, ":", path[i])
```

Figure 8: BFS Missionaries and Cannibals Problem

The function is_valid_state returns true or false whether the condition of the missionaries outnumbering the cannibals is maintained. The function get_successors utilizes the is_valid_state function to return the valid actions to take from the present state. Finally, the bfs function returns a list of the correct path taken to solve the problem, utilizing the Breadth-first search algorithm. It first creates a frontier queue, an explored set and a parents dictionary and inserts the initial state into frontier, then comenzing to search for paths from the initial state to the goal state.

Providing our implementation with an initial state of (3, 3, 0) and a goal state of (0, 0, 1), we obtain the following path:

```
Solution found with 11 moves:
Step 0 : (3, 3, 0)
Step 1 : (3, 1, 1)
Step 2 : (3, 2, 0)
Step 3 : (3, 0, 1)
Step 4 : (3, 1, 0)
Step 5 : (1, 1, 1)
Step 6 : (2, 2, 0)
Step 7 : (0, 2, 1)
Step 8 : (0, 3, 0)
Step 9 : (0, 1, 1)
Step 10 : (0, 2, 0)
Step 11 : (0, 0, 1)
```

Figure 9: BFS solution

Analyzing the above steps, we notice how the condition of always being more missionaries than cannibals is maintained while successfully navigating to the goal state.

3.11.3 - In our opinion, the reason why people have a hard time solving this puzzle is because it requires, or is made much more clearer, if one first draws each state and their possible actions. With 10 possible actions to take, it quickly becomes too complex for an average person to solve completely in their head.

References

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Hoboken, NJ: Pearson, 2021.

[2] aimacode (2016) aima-python [Source code]. https://github.com/aimacode/aima-python