



Assignment 2

CBS3004 Artificial Intelligence

Assignment 2

1) Implement Missionaries and cannibals problem in python with three Missionaries and three cannibals. You have to take care of the conditions to cross the river from one side to another side.

Exercise Date: 12-08-2025

Aim: To implement the Missionaries and Cannibals problem in Python with three missionaries and three cannibals, ensuring that at no point are missionaries outnumbered by cannibals on either side of the river.

Procedure:

1. Represent the state as (M_left, C_left, B, M_right, C_right) where
 - M_left and C_left = missionaries and cannibals on the left bank
 - M_right and C_right = missionaries and cannibals on the right bank
 - B indicates the boat position (left or right).
2. Define valid moves:
 - Move 1 missionary
 - Move 2 missionaries
 - Move 1 cannibal
 - Move 2 cannibals
 - Move 1 missionary and 1 cannibal
3. Ensure moves do not violate safety:
 - Missionaries are never outnumbered by cannibals on either bank.
4. Use **Breadth-First Search (BFS)** or **DFS** to explore possible states.
5. Stop when the goal state (0,0,right,3,3) is reached.
6. Print or display the sequence of valid steps.

Code:

```
from collections import deque

start_state = (3, 3, 0, 0, 'L')
end_state = (0, 0, 3, 3, 'R')

boat_moves = [(1, 0), (0, 1), (2, 0), (0, 2), (1, 1)]

def is_valid(state):
    M_left, C_left, M_right, C_right, _ = state

    if M_left < 0 or C_left < 0 or M_right < 0 or C_right < 0:
        return False
```

```
if M_left > 0 and C_left > M_left:
    return False
```

```
if M_right > 0 and C_right > M_right:
    return False
```

```
return True
```

```
def get_successors(state):
    successors = []
    M_left, C_left, M_right, C_right, boat = state

    for m, c in boat_moves:
        if boat == 'L':
            new_state = (M_left - m, C_left - c,
                        M_right + m, C_right + c,
                        'R')
        else:
            new_state = (M_left + m, C_left + c,
                        M_right - m, C_right - c,
                        'L')

        if is_valid(new_state):
            successors.append(new_state)

    return successors
```

```
def bfs(start, goal):
    queue = deque([(start, [start])])
    visited = set()

    while queue:
        state, path = queue.popleft()

        if state == goal:
            return path

        if state in visited:
            continue

        visited.add(state)

        for succ in get_successors(state):
            queue.append((succ, path + [succ]))

    return None
```

```
solution = bfs(start_state, end_state)
```

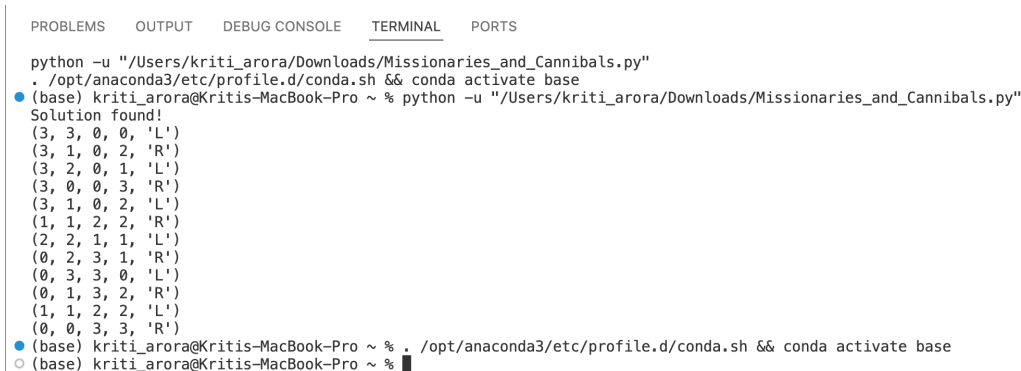
```
if solution:
```

```

print("Solution found!")
for step in solution:
    print(step)
else:
    print("No solution exists.")

```

Output:



```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
python -u "/Users/kriti_arora/Downloads/Missionaries_and_Cannibals.py"
. /opt/anaconda3/etc/profile.d/conda.sh && conda activate base
(base) kriti_arora@Kritis-MacBook-Pro ~ % python -u "/Users/kriti_arora/Downloads/Missionaries_and_Cannibals.py"
Solution found!
(3, 3, 0, 0, 'L')
(3, 1, 0, 2, 'R')
(3, 2, 0, 1, 'L')
(3, 0, 0, 3, 'R')
(3, 1, 0, 2, 'L')
(1, 1, 2, 2, 'R')
(2, 2, 1, 1, 'L')
(0, 2, 3, 1, 'R')
(0, 3, 3, 0, 'L')
(0, 1, 3, 2, 'R')
(1, 1, 2, 2, 'L')
(0, 0, 3, 3, 'R')
(base) kriti_arora@Kritis-MacBook-Pro ~ % . /opt/anaconda3/etc/profile.d/conda.sh && conda activate base
(base) kriti_arora@Kritis-MacBook-Pro ~ %

```

Result:

The Missionaries and Cannibals problem was successfully implemented in Python. The program generated a safe sequence of boat crossings that transferred all missionaries and cannibals across the river without violating constraints.

2) Implement Travelling sales man problem in python.

Exercise Date: 26-08-2025

Aim: To implement the Travelling Salesman Problem (TSP) in Python using brute force and heuristic approaches.

Procedure:

1. Represent the problem as a graph with cities as nodes and distances as weighted edges.
2. Input the distance matrix between cities.
3. Brute force approach:
 - Generate all permutations of cities.
 - Calculate the total travel distance for each permutation.
 - Select the minimum cost path.
4. Optimization: Use **Dynamic Programming (Held-Karp Algorithm)** to reduce time complexity.
5. Display the optimal route and its total cost.

Code:

```
import itertools

def travelling_salesman(graph, start):
    nodes = list(graph.keys())
    nodes.remove(start)
    shortest_path = None
    min_cost = float('inf')

    for perm in itertools.permutations(nodes):
        current_cost = 0
        current_path = [start] + list(perm) + [start]

        for i in range(len(current_path) - 1):
            current_cost += graph[current_path[i]][current_path[i + 1]]

        if current_cost < min_cost:
            min_cost = current_cost
            shortest_path = current_path

    return shortest_path, min_cost

graph = {
    'A': {'A': 0, 'B': 10, 'C': 15, 'D': 20},
    'B': {'A': 10, 'B': 0, 'C': 35, 'D': 25},
    'C': {'A': 15, 'B': 35, 'C': 0, 'D': 30},
    'D': {'A': 20, 'B': 25, 'C': 30, 'D': 0}
}

path, cost = travelling_salesman(graph, 'A')

print("Shortest Path:", " -> ".join(path))
print("Minimum Cost:", cost)
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

python -u "/Users/kriti_arora/Downloads/Travelling_Salesman.py"
. /opt/anaconda3/etc/profile.d/conda.sh && conda activate base
● (base) kriti_arora@Kritis-MacBook-Pro ~ % python -u "/Users/kriti_arora/Downloads/Travelling_Salesman.py"
Shortest Path: A -> B -> D -> C -> A
Minimum Cost: 80
● (base) kriti_arora@Kritis-MacBook-Pro ~ % . /opt/anaconda3/etc/profile.d/conda.sh && conda activate base
○ (base) kriti_arora@Kritis-MacBook-Pro ~ % █
```

Result:

The Travelling Salesman Problem was implemented in Python. The program successfully calculated the shortest possible route covering all cities exactly once and returning to the starting point, using both brute force and optimized dynamic programming approaches.

3) Implementation of A* and AO* Algorithms in python.

Exercise Date: 02-09-2025

Aim: To implement the A* and AO* search algorithms in Python for solving graph traversal and problem-solving tasks.

Procedure(A* Algorithm):

1. Represent the problem as a graph with nodes and edges.
2. Define:
 - $g(n)$ = cost from start node to current node.
 - $h(n)$ = heuristic estimate from current node to goal.
 - $f(n) = g(n) + h(n)$.
3. Use a priority queue (min-heap) to expand nodes with the lowest $f(n)$.
4. Continue until the goal node is reached.
5. Display the optimal path and cost.

Procedure(AO* Algorithm):

1. Represent the problem as an **AND-OR graph**.
2. Initialize all nodes with heuristic estimates.
3. Traverse the graph:
 - If an OR node is expanded, choose the child with minimum cost.
 - If an AND node is expanded, combine the costs of all children.
4. Update parent nodes iteratively until the solution graph is formed.
5. Display the solution path.

Code:

```
import heapq

def a_star(graph, heuristics, start, goal):
    open_list = [(heuristics[start], 0, start, [start])] # (f, g, node, path)
    closed = set()

    while open_list:
        f, g, node, path = heapq.heappop(open_list)

        if node == goal:
            return path, g

        if node in closed:
```

```

        continue

    closed.add(node)

    for neighbor, cost in graph[node].items():
        if neighbor not in closed:
            g_new = g + cost
            f_new = g_new + heuristics[neighbor]
            heapq.heappush(open_list, (f_new, g_new, neighbor, path +
[neighbor]))

    return None, float('inf')

```

```

graph_astar = {
    'A': {'B': 1, 'C': 3},
    'B': {'D': 3, 'E': 1},
    'C': {'F': 5},
    'D': {},
    'E': {'G': 2},
    'F': {'G': 2},
    'G': {}
}

```

```

heuristics = {'A': 7, 'B': 6, 'C': 5, 'D': 4, 'E': 2, 'F': 1, 'G': 0}

```

```

path, cost = a_star(graph_astar, heuristics, 'A', 'G')
print("A* Shortest Path:", path, "with cost", cost)

```

```

class AONode:
    def __init__(self, name):
        self.name = name
        self.children = [] # (list of alternatives, each alternative is a list of
nodes)
        self.cost = float('inf')
        self.solved = False
        self.best_child = None

```

```

def ao_star(node, graph, heuristic):
    if node.solved:
        return node.cost

    if not graph.get(node.name):
        node.cost = heuristic.get(node.name, 0)
        node.solved = True
        return node.cost

```

```

    min_cost = float('inf')
    best_child = None

```

```

    for alternative in graph[node.name]:
        cost = 0
        for child, weight in alternative:
            child_node = AONode(child)

```

```

        cost += weight + ao_star(child_node, graph, heuristic)

    if cost < min_cost:
        min_cost = cost
        best_child = alternative

    node.cost = min_cost
    node.best_child = best_child
    node.solved = True
    return node.cost

def print_solution(node):
    if node.best_child:
        print(f"{node.name} -> {[child for child, _ in node.best_child]}")
        for child, _ in node.best_child:
            print_solution(A0Node(child))

graph_aostar = {
    'A': [(('B', 1), ('C', 1)), (('D', 1))], # A can go to (B AND C) or D
    'B': [(('E', 1))],
    'C': [(('F', 1))],
    'D': [(('G', 1))],
    'E': [],
    'F': [],
    'G': []
}

heuristic_aostar = {'A': 3, 'B': 2, 'C': 2, 'D': 1, 'E': 0, 'F': 0, 'G': 0}

root = A0Node('A')
print("\nA0* Algorithm:")
total_cost = ao_star(root, graph_aostar, heuristic_aostar)
print("Total Cost from root:", total_cost)
print("Solution Path:")
print_solution(root)

```

Output:

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

python -u "/Users/kriti_arora/Downloads/A_A0.py"
. /opt/anaconda3/etc/profile.d/conda.sh && conda activate base
• (base) kriti_arora@Kritis-MacBook-Pro ~ % python -u "/Users/kriti_arora/Downloads/A_A0.py"
A* Shortest Path: ['A', 'B', 'E', 'G'] with cost 4

A0* Algorithm:
Total Cost from root: 2
Solution Path:
A -> ['D']
• (base) kriti_arora@Kritis-MacBook-Pro ~ % . /opt/anaconda3/etc/profile.d/conda.sh && conda activate base
❖ (base) kriti_arora@Kritis-MacBook-Pro ~ %

```

Result:

Both A* and AO* algorithms were successfully implemented in Python.

- A* found the optimal path in a weighted graph using heuristic estimates.
- AO* solved AND-OR graph problems efficiently, producing the minimal cost solution tree