Assignment 2

CBS3004 Artificial Intelligence

Eshlok Agarwal

23BBS0096

**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).

1. Define valid moves:

* Move 1 missionary
* Move 2 missionaries
* Move 1 cannibal
* Move 2 cannibals
* Move 1 missionary and 1 cannibal

1. Ensure moves do not violate safety:

* Missionaries are never outnumbered by cannibals on either bank.

1. Use **Breadth-First Search (BFS)** or **DFS** to explore possible states.
2. Stop when the goal state (0,0,right,3,3) is reached.
3. 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!")

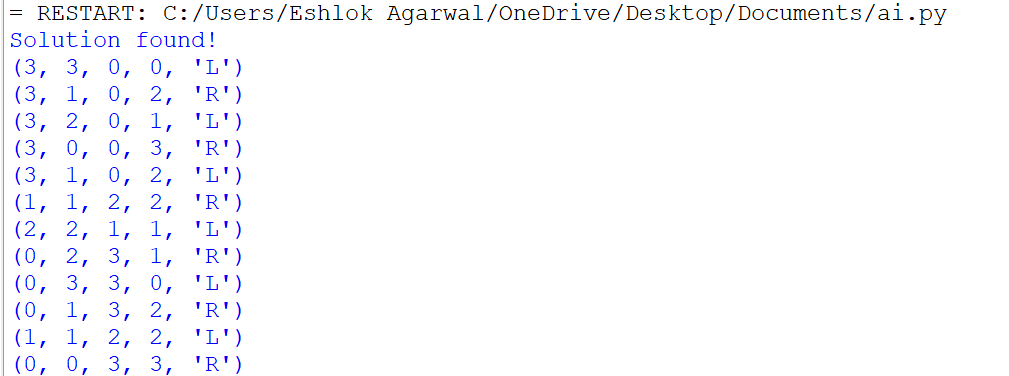
for step in solution:

print(step)

else:

print("No solution exists.")

**Output:**

****

**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.

1. **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}

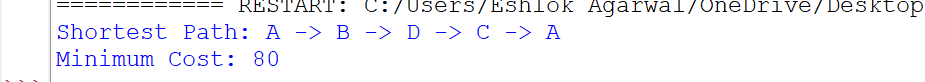
}

path, cost = travelling\_salesman(graph, 'A')

print("Shortest Path:", " -> ".join(path))

print("Minimum Cost:", cost)

**Output:**



**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(AONode(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 = AONode('A')

print("\nAO\* Algorithm:")

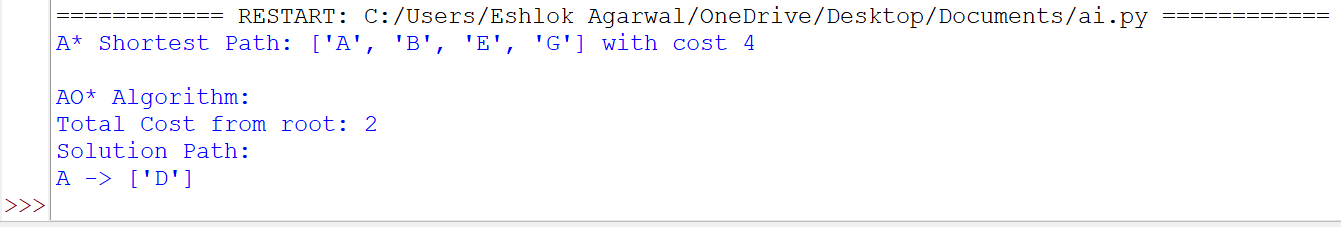
total\_cost = ao\_star(root, graph\_aostar, heuristic\_aostar)

print("Total Cost from root:", total\_cost)

print("Solution Path:")

print\_solution(root)

**Output:**

****

**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