Indian Institute of Information Technology Surat

Lab Report on
Artificial Intelligence (CS 701) Practical

Submitted by

[RAHUL KUMAR SINGH] (UI21CS44)

Course Faculty

Dr. Ritesh Kumar

Mrs. Archana Balmik

Department of Computer Science and Engineering

Indian Institute of Information Technology Surat

Gujarat-394190, India

Aug-2024

Lab No: 5

Aim:

To formulate state space representations and solve sequence problems for the Farmer-Wolf-Goat-Cabbage crossing, Water Jugs problem, and 8 Tiles puzzle

Description:

Give State Space Representation for following

1. Farmer, wolf, goat, and cabbage problem: A farmer with his wolf, goat, and cabbage come to the edge of a river they wish to cross. There is a boat at the river's edge, but, of course, only the farmer can row. The boat also can carry only two things, including the rower, at a time. If the wolf is ever left alone with the goat, the wolf will eat the goat; similarly if the goat is left alone with the cabbage, the goat will eat the cabbage. Devise a sequence of crossings of the river so that all four characters arrive safely on the other side of the river.
2. Water jugs problem: There are two jugs, one holding 3 and the other 5 gallons of water. A number of things can be done with the jugs: they can be filled, emptied, and dumped one into the other either until the poured-into jug is full or until the poured-out-of jug is empty. Devise a sequence of actions that will produce 4 gallons of water in the larger jug.
3. 8 tiles puzzle problem
   Initial state:
   [2,8,3],[1,6,4],[7,_,5]
   Goal state:
   [1,2,3],[8,_,4],[7,6,5]

State Space Representation:

Q1 Farmer, Wolf, Goat, and Cabbage Problem

- **Initial State**: All on the left bank.
- **Goal State**: All safely on the right bank.
- **Path**:
  1. Farmer takes the goat across.
  2. Returns alone.
  3. Take the wolf across.
  4. Bring the goat back.
  5. Take cabbage across.
  6. Returns alone.
  7. Take the goat across.
- **Solution**:
  (L, L, L, L) -> (R, L, R, L) -> (L, L, L, L) -> (R, R, L, L) -> (L, R, L, L) -> (R, R, L, R) -> (L, R, L, R) -> (R, R, R, R)

Q2 Water Jugs Problem

- **Initial State: (0, 0)**
- **Goal State: (x, 4) in the 5-gallon jug.**
- **Path:**
  1. **Fill a 5-gallon jug.**

2. Pour into a 3-gallon jug.
3. Empty 3-gallon jug.
4. Pour 2 gallons into a 3-gallon jug.
5. Fill a 5-gallon jug again.
6. Pour into a 3-gallon jug.

- **Solution:**
  (0, 0) -> (0, 5) -> (3, 2) -> (0, 2) -> (2, 0) -> (2, 5) -> (3, 4)

Q3 8-Tiles Puzzle Problem

Initial State:

```
[2, 8, 3]
[1, 6, 4]
[7, _, 5]
```

Goal State:

```
[1, 2, 3]
[8, _, 4]
[7, 6, 5]
```

Path:

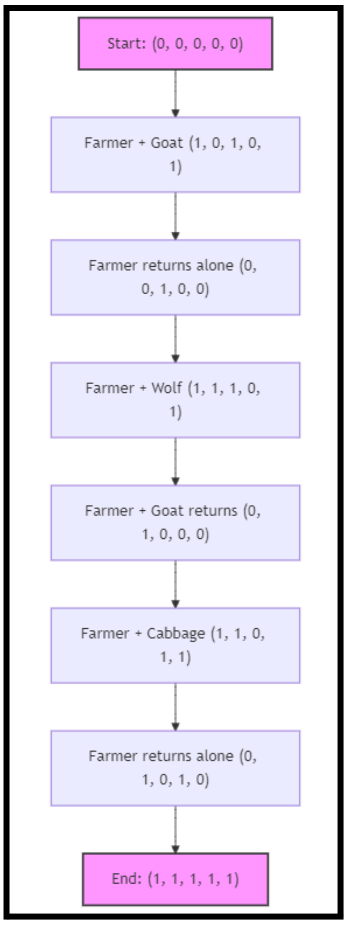
1. Move tile 6 down.
2. Move tile 8 down.
3. Move tile 2 left.
4. Move tile 1 up.
5. Move tile 8 left.

Solution:

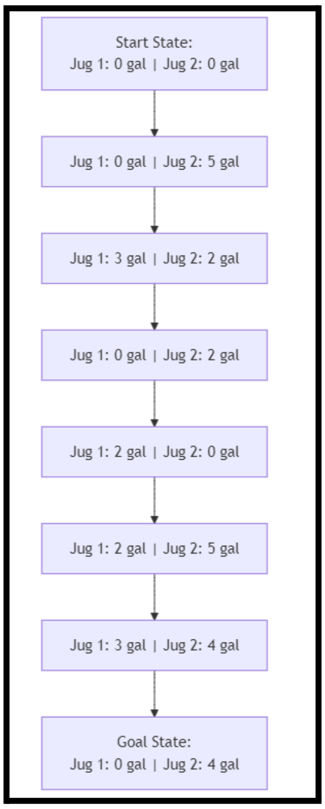
```
(2, 8, 3, 1, 6, 4, 7, _, 5)
-> (2, 8, 3, 1, _, 4, 7, 6, 5)
-> (2, _, 3, 1, 8, 4, 7, 6, 5)
-> (_, 2, 3, 1, 8, 4, 7, 6, 5)
-> (1, 2, 3, _, 8, 4, 7, 6, 5)
-> (1, 2, 3, 8, _, 4, 7, 6, 5)
```

Flowchart:

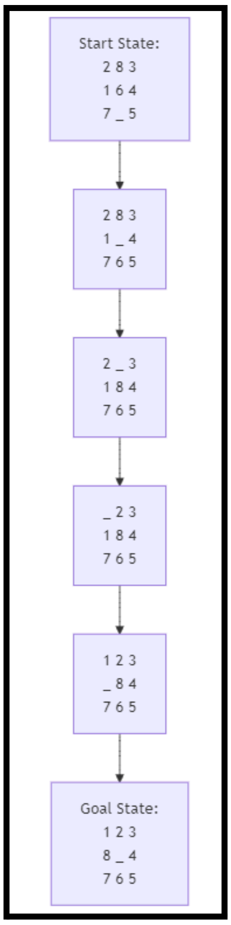
Q1


Start: (0, 0, 0, 0, 0)
Farmer + Goat (1, 0, 1, 0, 1)
Farmer returns alone (0, 0, 1, 0, 0)
Farmer + Wolf (1, 1, 1, 0, 1)
Farmer + Goat returns (0, 1, 0, 0, 0)
Farmer + Cabbage (1, 1, 0, 1, 1)
Farmer returns alone (0, 1, 0, 1, 0)
End: (1, 1, 1, 1, 1)

Q2


Start State:
Jug 1: 0 gal | Jug 2: 0 gal
Jug 1: 0 gal | Jug 2: 5 gal
Jug 1: 3 gal | Jug 2: 2 gal
Jug 1: 0 gal | Jug 2: 2 gal
Jug 1: 2 gal | Jug 2: 0 gal
Jug 1: 2 gal | Jug 2: 5 gal
Jug 1: 3 gal | Jug 2: 4 gal
Goal State:
Jug 1: 0 gal | Jug 2: 4 gal

Q3



Code:

Q1

```
from collections import deque

def is_valid_state(state):
    farmer, wolf, goat, cabbage = state
    if wolf == goat and wolf != farmer:
        return False
    if goat == cabbage and goat != farmer:
        return False
    return True

def get_neighbors(state):
    farmer, wolf, goat, cabbage = state
    neighbors = []
    moves = [('wolf', wolf), ('goat', goat), ('cabbage', cabbage), ('none', None)]

    for item, current in moves:
        if item == 'none':
            new_state = (1 - farmer, wolf, goat, cabbage)
        else:
            new_farm = 1 - farmer
            if item == 'wolf':
                new_state = (new_farm, 1 - wolf, goat, cabbage)
            elif item == 'goat':
                new_state = (new_farm, wolf, 1 - goat, cabbage)
            elif item == 'cabbage':
                new_state = (new_farm, wolf, goat, 1 - cabbage)

        if is_valid_state(new_state):
            neighbors.append(new_state)
    return neighbors

def format_state(state):
    bank = ["Left", "Right"]
    f, w, g, c = state
    f, w, g, c = bank[f], bank[w], bank[g], bank[c]
    return f"{f:<6} {w:<6} {g:<6} {c:<6}"

def solve_farm_problem():
    initial_state = (0, 0, 0, 0)
    goal_state = (1, 1, 1, 1)
    queue = deque([(initial_state, [])])
    visited = set()

    while queue:
        (state, path) = queue.popleft()
        if state == goal_state:
            return path + [state]

        if state in visited:
            continue
        visited.add(state)

        for neighbor in get_neighbors(state):
            queue.append((neighbor, path + [state]))

    return None

solution = solve_farm_problem()
if solution:
    print("Solution Path:")
    print(f"{'Farmer':<6} {'Wolf':<6} {'Goat':<6} {'Cabbage':<6}")
    for step in solution:
        print(format_state(step))
else:
    print("No solution found.")
```

Q2

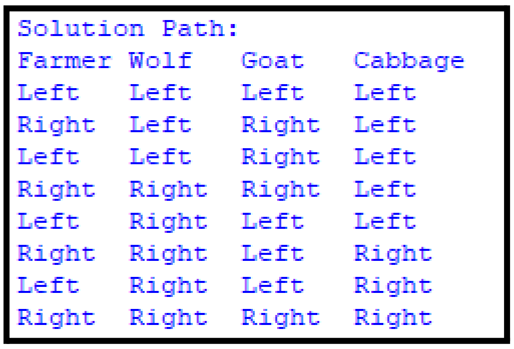
```
from collections import deque

def get_neighbors(state):
    x, y = state
    neighbors = []

    neighbors.append((3, y))
    neighbors.append((x, 5))
    neighbors.append((0, y))
    neighbors.append((x, 0))

    pour_to_2 = min(x, 5 - y)
    neighbors.append((x - pour_to_2, y + pour_to_2))
    pour_to_1 = min(y, 3 - x)
    neighbors.append((x + pour_to_1, y - pour_to_1))

    return neighbors

def format_state(state):
    return f"Jug 1: {state[0]} gal | Jug 2: {state[1]} gal"

def solve_water_jugs_problem():
    initial_state = (0, 0)
    goal_state = (0, 4)
    queue = deque([(initial_state, [])])
    visited = set()
    while queue:
        (state, path) = queue.popleft()
        if state == goal_state:
            return path + [state]

        if state in visited:
            continue
        visited.add(state)

        for neighbor in get_neighbors(state):
            queue.append((neighbor, path + [state]))

    return None

solution = solve_water_jugs_problem()
if solution:
    print("Solution Path:")
    for step in solution:
        print(format_state(step))
else:
    print("No solution found.")
```

Q3

```
import heapq

def heuristic(state):
    goal_state = [1, 2, 3, 8, 0, 4, 7, 6, 5]
    distance = 0
    for i, value in enumerate(state):
        goal_index = goal_state.index(value)
        if value != 0:
            current_row, current_col = divmod(i, 3)
            goal_row, goal_col = divmod(goal_index, 3)
            distance += abs(current_row - goal_row) + abs(current_col - goal_col)
    return distance

def get_neighbors(state):
    neighbors = []
    blank_index = state.index(0)
    row, col = divmod(blank_index, 3)
    possible_moves = []

    if row > 0:
        possible_moves.append(blank_index - 3)
    if row < 2:
        possible_moves.append(blank_index + 3)
    if col > 0:
        possible_moves.append(blank_index - 1)
    if col < 2:
        possible_moves.append(blank_index + 1)

    for new_index in possible_moves:
        new_state = list(state)
        new_state[blank_index], new_state[new_index] = new_state[new_index],
new_state[blank_index]
        neighbors.append(tuple(new_state))

    return neighbors

def format_state(state):
    return "\n".join(" ".join(str(x) if x != 0 else "_" for x in state[i:i+3]) for i in range(0,
9, 3))

def solve_8_tiles_problem():
    initial_state = (2, 8, 3, 1, 6, 4, 7, 0, 5)
    goal_state = (1, 2, 3, 8, 0, 4, 7, 6, 5)

    open_set = []
    heapq.heappush(open_set, (heuristic(initial_state), 0, initial_state, []))
    visited = set()

    while open_set:
        _, cost, state, path = heapq.heappop(open_set)

        if state == goal_state:
            return path + [state]

        if state in visited:
            continue
        visited.add(state)

        for neighbor in get_neighbors(state):
            heapq.heappush(open_set, (cost + 1 + heuristic(neighbor), cost + 1, neighbor, path + 
[state]))

    return None

solution = solve_8_tiles_problem()
if solution:
    print("Solution Path:")
    for step in solution:
        print(format_state(step))
        print()
else:
    print("No solution found.")
```

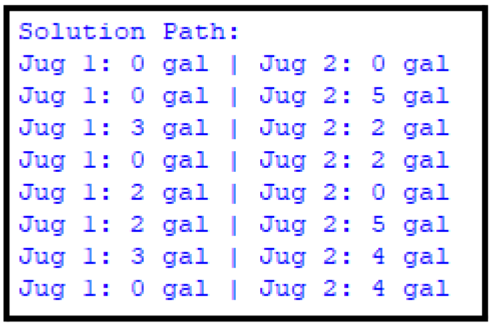
Output:

Q1



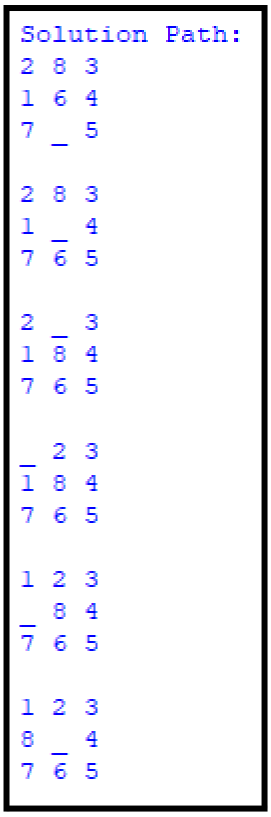
```
Solution Path:
Farmer Wolf   Goat   Cabbage
Left   Left   Left   Left
Right  Left   Right  Left
Left   Left   Right  Left
Right  Right  Right  Left
Left   Right  Left   Left
Right  Right  Left   Right
Left   Right  Left   Right
Right  Right  Right  Right
```

Q2



```
Solution Path:
Jug 1: 0 gal | Jug 2: 0 gal
Jug 1: 0 gal | Jug 2: 5 gal
Jug 1: 3 gal | Jug 2: 2 gal
Jug 1: 0 gal | Jug 2: 2 gal
Jug 1: 2 gal | Jug 2: 0 gal
Jug 1: 2 gal | Jug 2: 5 gal
Jug 1: 3 gal | Jug 2: 4 gal
Jug 1: 0 gal | Jug 2: 4 gal
```

Q3



```
Solution Path:
2 8 3
1 6 4
7 _ 5

2 8 3
1 _ 4
7 6 5

2 _ 3
1 8 4
7 6 5

_ 2 3
1 8 4
7 6 5

1 2 3
_ 8 4
7 6 5

1 2 3
8 _ 4
7 6 5
```

Conclusion:

- All problems involve exploring state spaces to find optimal solutions through defined transitions.
- Python code utilizes search algorithms like BFS and A* to solve each problem.
- The solutions effectively address the problem constraints and achieve the desired goals.