## 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)

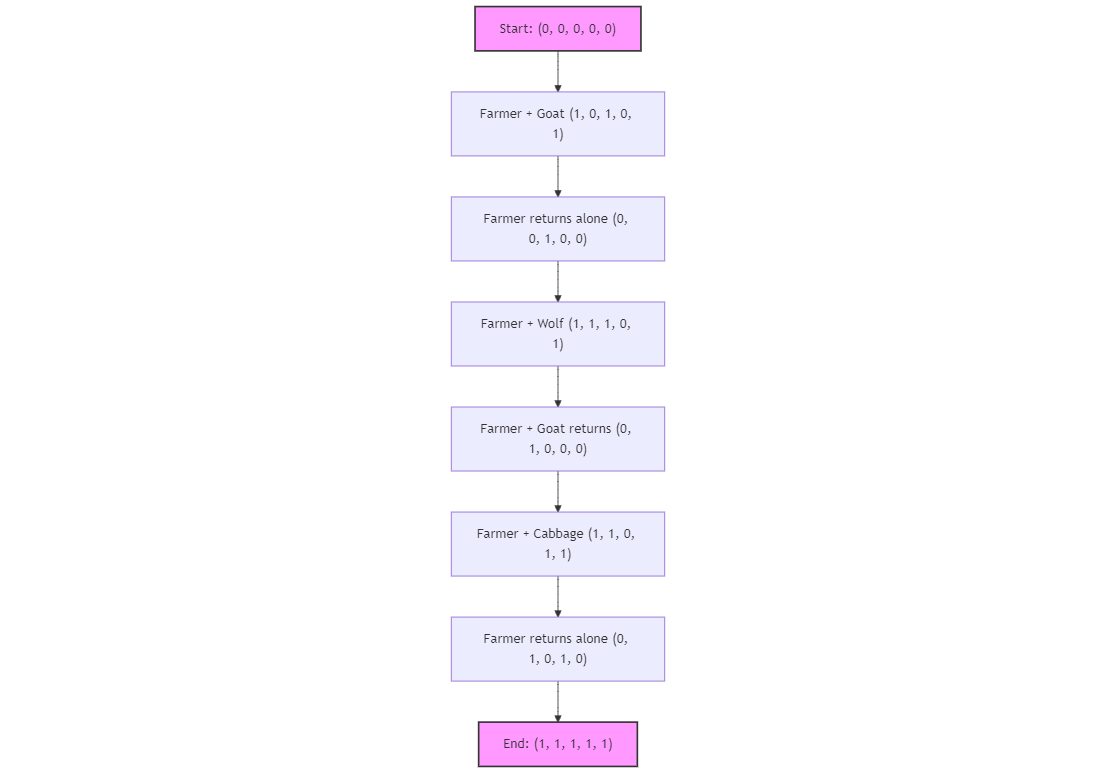
-> (\_, 2, 3, 1, 8, 4, 7, 6, 5)

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

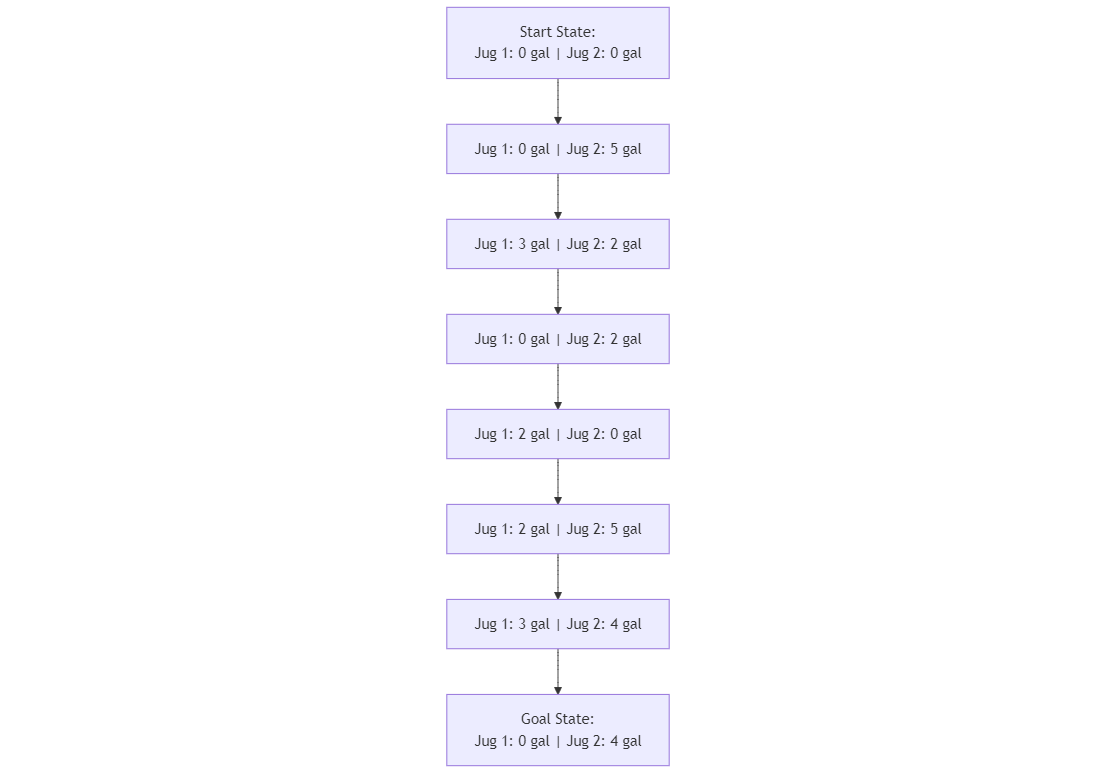
-> (1, 2, 3, 8, \_, 4, 7, 6, 5)

## Flowchart:

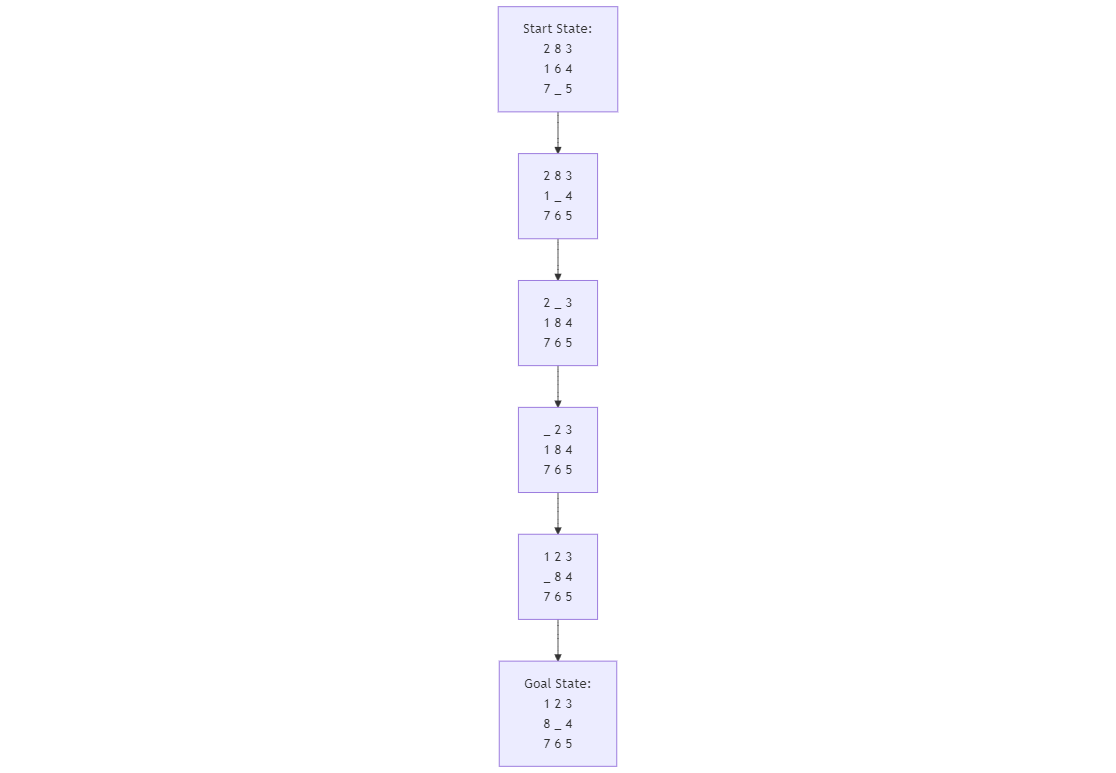
**Q1**

****

**Q2**

****

**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))

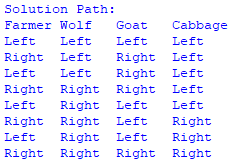
print()

else:

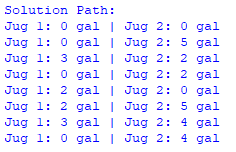
print("No solution found.")

## Output:

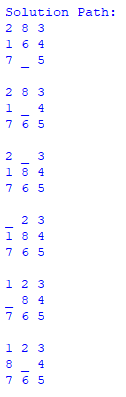
**Q1**

****

**Q2**

****

**Q3**

****

## 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.