



Department of Computer Science and Engineering (Data Science)

Subject: Artificial Intelligence (DJ19DSC502)

AY: 2023-24

Experiment 1

60009210033

Jhanvi Parekh

D11

(Problem Solving)

Aim: Implement domain specific functions for given problems required for problem solving.

Theory:

There are two domain specific functions required in all problem solving methods.

1. GoalTest Function:

goalTest(State) Returns *true* if the input state is the goal state and *false* otherwise.

goalTest(State, Goal) Returns *true* if *State* matches *Goal*, and *false* otherwise.

2. MoveGen function:

Initialize set of successors C to empty set.

Add M to the complement of given state N to get new state S.

If given state has Left, then add Right to S, else add Left.

If legal(S) then add S to set of successors C.

For each other-entity E in N

make a copy S' of S,

add E to S',

If legal (S'), then add S' to C.

Return (C) .

Lab Assignment to do:

Create MoveGen and GoalTest Functions for the given problems

1. Water Jug Problem



Department of Computer Science and Engineering (Data Science)

There are two jugs available of different volumes such as a 3 litres and a 7 litres and you have to measure a different volume such as 6 litre.

```
class State:
    def __init__(self, jug1, jug2):
        self.jug1 = jug1
        self.jug2 = jug2

    def __eq__(self, other):
        return self.jug1 == other.jug1 and self.jug2 == other.jug2

    def __hash__(self):
        return hash((self.jug1, self.jug2))

def move_gen(state):
    moves = []

    # Fill jug 1
    moves.append(State(3, state.jug2))

    # Fill jug 2
    moves.append(State(state.jug1, 7))

    # Empty jug 1
    moves.append(State(0, state.jug2))

    # Empty jug 2
    moves.append(State(state.jug1, 0))

    # Pour jug 1 into jug 2
    pour_amount = min(state.jug1, 7 - state.jug2)
    moves.append(State(state.jug1 - pour_amount, state.jug2 + pour_amount))

    # Pour jug 2 into jug 1
```



Department of Computer Science and Engineering (Data Science)

60009210033_Jhanvi Parekh_AI_Exp01.ipynb

```
# Pour jug 2 into jug 1
pour_amount = min(3 - state.jug1, state.jug2)
moves.append(State(state.jug1 + pour_amount, state.jug2 - pour_amount))

return moves

def goal_test(state):
    return state.jug1 == 6 or state.jug2 == 6

def bfs(initial_state):
    queue = [(initial_state, [])]
    visited = set()

    while queue:
        state, path = queue.pop(0)
        visited.add(state)

        if goal_test(state):
            return path

        for next_state in move_gen(state):
            if next_state not in visited:
```

Step 1: Jug 1 = 3, Jug 2 = 0
Step 2: Jug 1 = 0, Jug 2 = 3
Step 3: Jug 1 = 3, Jug 2 = 3
Step 4: Jug 1 = 0, Jug 2 = 6

```
[ ] jugs = []
for i in range(3):
    jugs.append(int(input(f"Enter capacity of jug {i+1} ")))
```



Department of Computer Science and Engineering (Data Science)

The screenshot displays a Google Colab notebook titled "60009210033_Jhanvi Parekh_AI_Exp01.ipynb". The notebook contains a Python program for the Travelling Salesman Problem. The program uses a recursive approach to find the minimum cost path visiting all cities. The input shows capacities for three jugs: 5, 3, and 8. The output shows the minimum cost path: [[0, 0, 8], [5, 0, 3], [0, 3, 5], [5, 3, 0]].

```
jugs = []
for i in range(3):
    jugs.append(int(input(f"Enter capacity of jug {i+1} ")))
temp = jugs[2]
ans = []
lst = [0, 0, temp]
ans.append(lst)
lst = [min(temp, jugs[0]), 0, temp - min(jugs[0], temp)]
ans.append(lst)
lst = [0, min(temp, jugs[1]), temp - min(jugs[1], temp)]
ans.append(lst)
lst = [min(temp, jugs[0]), min(jugs[1], temp - min(temp, jugs[0]))]
lst.append(temp - (lst[0] + lst[1]))
if lst not in ans:
    ans.append(lst)
lst = [min(jugs[0], temp - min(temp, jugs[1]), min(temp, jugs[1]))]
lst.append(temp - (lst[0] + lst[1]))
if lst not in ans:
    ans.append(lst)
print(ans)
```

Enter capacity of jug 1 5
Enter capacity of jug 2 3
Enter capacity of jug 3 8
[[0, 0, 8], [5, 0, 3], [0, 3, 5], [5, 3, 0]]

Question 2

2. Travelling Salesman Problem

A salesman is travelling and selling his/her product to in different cities. The condition is that it has to travel each city just once.



Department of Computer Science and Engineering (Data Science)

60009210033_Jhanvi Parekh_AI_Exp01.ipynb

File Edit View Insert Runtime Tools Help [Cannot save changes](#)

+ Code + Text Copy to Drive

Question 2

```
[ ] class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, node, neighbors):
        self.graph[node] = neighbors

    def get_neighbors(self, node):
        return self.graph.get(node, [])

    def bfs_shortest_path(self, start_node, end_node):
        visited = set()
        queue = [(start_node, [start_node])] # Queue stores tuples (node, path)

        while queue:
            node, path = queue.pop(0)
            if node not in visited:
                visited.add(node)

                if node == end_node:
```

60009210033_Jhanvi Parekh_AI_Exp01.ipynb

File Edit View Insert Runtime Tools Help [Cannot save changes](#)

+ Code + Text Copy to Drive

```
node, path = queue.pop(0)
if node not in visited:
    visited.add(node)

    if node == end_node:
        return path

    for neighbor in self.get_neighbors(node):
        if neighbor not in visited:
            new_path = path + [neighbor]
            queue.append((neighbor, new_path))

    return None # No path found

# Function to input the graph from the user
def input_graph():
    graph = Graph()
    while True:
        node = input("Enter a node (or 'done' to finish): ")
        if node == 'done':
            break

        neighbors = input(f"Enter neighbors for node '{node}' (comma-separated, or 'none' for no neighbors): ").split(',')
```



Department of Computer Science and Engineering (Data Science)

The screenshot displays a Google Colab notebook titled "60009210033_Jhanvi Parekh_AI_Exp01.ipynb". The notebook contains a Python script for finding the shortest path in a graph using Breadth-First Search (BFS). The code defines a function to add edges to a graph, prompts the user for input, and then finds the shortest path from node 'A' to node 'G'. The execution output shows the user entering nodes and their neighbors, and the final shortest path found is A -> C -> E -> G.

```
neighbors = input(f"Enter neighbors for node '{node}': (comma-separated, or 'none' for no neighbors): ").split(',')
neighbors = [neighbor.strip() for neighbor in neighbors if neighbor.strip() != 'none']

graph.add_edge(node, neighbors)

return graph

# Example usage:
print("Input your graph:")
user_graph = input_graph()

start_node = 'A'
end_node = 'G'

shortest_path = user_graph.bfs_shortest_path(start_node, end_node)

if shortest_path:
    print(f"Shortest path from {start_node} to {end_node}: {' -> '.join(shortest_path)}")
else:
    print(f"No path found from {start_node} to {end_node}.")
```

Input your graph:
Enter a node (or 'done' to finish): A
Enter neighbors for node 'A' (comma-separated, or 'none' for no neighbors): B,C
Enter a node (or 'done' to finish): B
Enter neighbors for node 'B' (comma-separated, or 'none' for no neighbors): C,D
Enter a node (or 'done' to finish): C
Enter neighbors for node 'C' (comma-separated, or 'none' for no neighbors): E,F
Enter a node (or 'done' to finish): D
Enter neighbors for node 'D' (comma-separated, or 'none' for no neighbors): H,F
Enter a node (or 'done' to finish): E
Enter neighbors for node 'E' (comma-separated, or 'none' for no neighbors): G
Enter a node (or 'done' to finish): F
Enter neighbors for node 'F' (comma-separated, or 'none' for no neighbors): I
Enter a node (or 'done' to finish): H
Enter neighbors for node 'H' (comma-separated, or 'none' for no neighbors): F,I
Enter a node (or 'done' to finish): I
Enter neighbors for node 'I' (comma-separated, or 'none' for no neighbors): J
Enter a node (or 'done' to finish): J
Enter neighbors for node 'J' (comma-separated, or 'none' for no neighbors): G
Enter a node (or 'done' to finish): done
Shortest path from A to G: A -> C -> E -> G

3. 8 Puzzle Problem

An initial state is given in a 8 puzzle where one place is blank out of 9 places. You can shift this blank space and get a different state to reach to a given goal state.



Department of Computer Science and Engineering (Data Science)

60009210033_Jhanvi Parekh_AI_Exp01.ipynb

File Edit View Insert Runtime Tools Help [Cannot save changes](#)

+ Code + Text Copy to Drive Connect

Question3

```
[ ] def gridPrint(blankX, blankY, tempX, tempY, board):
    for i in range(3):
        for j in range(3):
            if(i== blankX and j== blankY):
                print(f"{board[tempX][tempY]}", end= " ")
                continue
            if(i== tempX and j== tempY):
                print(f"{board[blankX][blankY]}", end= " ")
                continue
            print(f"{board[i][j]}", end= " ")
        print("")
    board= []

    posX= -1
    posY= -1
    count= 1
```

posY= -1
count= 1
solved= True

```
for i in range(3):
    lst= []
    for j in range(3):
        temp= int(input(f"Enter for [{i+1}], [{j+1}] number "))

        lst.append(temp)
        if(count!= temp and count!= 9):
            solved= False
            count+= 1

        if(temp== 0):
            posX= i
            posY= j
            board.append(lst)
```

28°C Haze



Department of Computer Science and Engineering (Data Science)

60009210033_Jhanvi Parekh_AI_Exp01.ipynb

```
posY = j
board.append(1st)
if(solved):
    print("Already solved ")
else:
    for i in range(3):
        for j in range(3):
            print(board[i][j], end= " ")
            print()
        ans = []
        if(posX != 0):
            ans.append(board[posX-1][posY])
            print("If we move the blank space up ")
            gridPrint(posX, posY, posX-1, posY, board)
        if(posY != 0):
            ans.append(board[posX][posY-1])
```

60009210033_Jhanvi Parekh_AI_Exp01.ipynb

```
ans.append(board[posX][posY-1])
print("If we move the blank space left ")
gridPrint(posX, posY, posX, posY-1, board)
if(posX != 2):
    ans.append(board[posX+1][posY])
    print("If we move the blank space down ")
    gridPrint(posX, posY, posX+1, posY, board)
if(posY != 2):
    ans.append(board[posX][posY+1])
    print("If we move the blank space right ")
    gridPrint(posX, posY, posX, posY+1, board)
```




Department of Computer Science and Engineering (Data Science)

```
Enter for [1], [1] number 1
Enter for [1], [2] number 5
Enter for [1], [3] number 4
Enter for [2], [1] number 0
Enter for [2], [2] number 8
Enter for [2], [3] number 6
Enter for [3], [1] number 3
Enter for [3], [2] number 7
Enter for [3], [3] number 2
1 5 4
0 8 6
3 7 2
If we move the blank space up
0 5 4
1 8 6
3 7 2
If we move the blank space down
1 5 4
3 8 6
0 7 2
If we move the blank space right
1 5 4
8 0 6
3 7 2
```

Link:

<https://colab.research.google.com/drive/1KDRKX3Cx3cAjzXph0p7J-9b4vT0mmtJc?usp=sharing>