Assignment – 1

Some use of NumPy:

- **Code 1: To check whether all elements in a vector are non-zero or Zero using np.all()?**

```
import numpy as np
v1 = np.array([4, 2, 7])
print("Vector :", v1)
print(np.all(v1))
v2 = np.array([0, 4, 2])
print("Vector :", v2)
print(np.all(v2))
```

- **Code 2: Vector addition, subtraction, multiplication**

```
import numpy as np
# Define two vectors
v1 = np.array([4, 2, 7])
v2 = np.array([1, 3, 5])
# Vector Addition
print("Vector Addition:", np.add(v1, v2))
# Vector Subtraction
print("Vector Subtraction:", np.subtract(v1, v2))
# Element-wise Multiplication
print("Vector Multiplication (Element-wise):", np.multiply(v1, v2))
```

- **Code 3: Matrix addition, subtraction, multiplication**

```
import numpy as np
# Define two 2x2 matrices
m1 = np.array([[2, 4], [3, 1]])
m2 = np.array([[5, 6], [7, 8]])
# Matrix Addition
m_add = np.add(m1, m2)  # or m1 + m2
print("Matrix Addition:\n", m_add)
# Matrix Subtraction
m_sub = np.subtract(m1, m2)  # or m1 - m2
print("Matrix Subtraction:\n", m_sub)
# Matrix Multiplication (Dot Product)
m_mul = np.dot(m1, m2)  # or m1 @ m2 in Python 3.5+
```

```
print("Matrix Multiplication (Dot Product):\n", m_mul)
```

- **Code 4: Create an array A of size 10 with each element of it set to value 3.**

```
import numpy as np
v1 = np.full(10,3)
print (v1)
```

- **Code 5: Find the memory size of this array and its individual elements.**

```
import numpy as np
a = np.array([2,3,5,5,9])
print(a.size)
print(a.itemsize)
print(a.nbytes)
```

- **Code 6: Create an array of size 10 with values ranging from 0 to 90 evenly spaced.**

```
import numpy as np
arr = np.linspace(0, 90, 10)
print(arr)
```

- **Code 7: Reverse the elements of an array**

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
reversed_arr = arr[::-1]
print(reversed_arr)
```

- **Code 8 : Add arrays a and b, store the result in array c.**

```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = a + b
print(c)
```

Assignment – 2

- ## Code 1: BFS

```
def bfs(graph, start):
    visited = set()
    queue = [start]
    while queue:
        node = queue[0]
        queue = queue[1:]
        if node not in visited:
            print(node, end=' ')
            visited.add(node)
            # Enqueue unvisited neighbors
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)
# Example graph as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
print("Breadth-First Search starting from node A:")
bfs(graph, 'A')
```

Output : Breadth-First Search starting from node A: A B C D E F

- ## Code 2: DFS

```
# Depth-First Search (DFS) using adjacency list and recursion
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=' ')
    # Recur for all adjacent nodes
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
```

```
# Example graph represented as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
# Call DFS starting from node 'A'
print("Depth-First Search starting from node A:")
dfs(graph, 'A')
```

Output : Depth-First Search starting from node A: A B D E F C

Assignment – 3

- **Code 1: Tic-tac-toe (one user another computer)**

```
def print_board(board):
    """Prints the Tic-Tac-Toe board."""
    print(f"\n{board[0]} | {board[1]} | {board[2]}")
    print("--+---+--")
    print(f"{board[3]} | {board[4]} | {board[5]}")
    print("--+---+--")
    print(f"{board[6]} | {board[7]} | {board[8]}\n")

def check_win(board, player):
    """Checks if the current player has won."""
    win_conditions = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],   # Rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8],   # Columns
        [0, 4, 8], [2, 4, 6]               # Diagonals
    ]
    for condition in win_conditions:
        if board[condition[0]] == board[condition[1]] == board[condition[2]] ==
player:
            return True
    return False
def check_draw(board):
    """Checks if the board is full and there is no winner."""
```

```
    return all(cell in ['X', 'O'] for cell in board)
def play_game():
    """Runs the Tic-Tac-Toe game."""
    board = [' ' for _ in range(9)]
    current_player = 'X'
    while True:
        print_board(board)
        try:
            move = int(input(f"Player {current_player}, enter your move (1-9): ")) -
1
            if move < 0 or move >= 9 or board[move] != ' ':
                print("Invalid move. Try again.")
                continue
        except ValueError:
            print("Invalid input. Please enter a number between 1 and 9.")
            continue
        board[move] = current_player
        if check_win(board, current_player):
            print_board(board)
            print(f"Player {current_player} wins!")
            break
        elif check_draw(board):
            print_board(board)
            print("It's a draw!")
            break
        current_player = 'O' if current_player == 'X' else 'X'
# For Jupyter Notebook, just call the function directly
play_game()
```

- **Code 1: Tic-tac-toe (both user)**

```
import math
def print_board(board):
    """Prints the Tic-Tac-Toe board."""
    print(f"\n{board[0]} | {board[1]} | {board[2]}")
    print("--+---+--")
    print(f"{board[3]} | {board[4]} | {board[5]}")
    print("--+---+--")
    print(f"{board[6]} | {board[7]} | {board[8]}\n")
def check_win(board, player):
    """Checks if the current player has won."""
    win_conditions = [
```

```
        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # Rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # Columns
        [0, 4, 8], [2, 4, 6]              # Diagonals
    ]
    for condition in win_conditions:
        if board[condition[0]] == board[condition[1]] == board[condition[2]] ==
player:
            return True
    return False
def check_draw(board):
    """Checks if the board is full and there is no winner."""
    return all(cell in ['X', 'O'] for cell in board)
def minimax(board, depth, is_maximizing):
    """The Minimax algorithm to find the best move for the computer."""
    if check_win(board, 'O'):
        return 1  # AI wins
    if check_win(board, 'X'):
        return -1  # Human wins
    if check_draw(board):
        return 0  # Draw
    if is_maximizing:
        best_score = -math.inf
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'O'
                score = minimax(board, depth + 1, False)
                board[i] = ' '
                best_score = max(best_score, score)
        return best_score
    else:
        best_score = math.inf
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'X'
                score = minimax(board, depth + 1, True)
                board[i] = ' '
                best_score = min(best_score, score)
        return best_score
def find_best_move(board):
    """Finds the best move for the computer using the Minimax algorithm."""
    best_move = None
```

```
    best_score = -math.inf
    for i in range(9):
        if board[i] == ' ':
            board[i] = 'O'
            move_score = minimax(board, 0, False)
            board[i] = ' '
            if move_score > best_score:
                best_score = move_score
                best_move = i
    return best_move
def play_game():
    """Runs the Tic-Tac-Toe game."""
    board = [' ' for _ in range(9)]  # Initialize the board
    current_player = 'X'
    while True:
        print_board(board)

        if current_player == 'X':
            # Player 1's move
            try:
                move = int(input(f"Player {current_player}, enter your move (1-9):
")) - 1
                if move < 0 or move >= 9 or board[move] != ' ':
                    print("Invalid move. Try again.")
                    continue
            except ValueError:
                print("Invalid input. Please enter a number between 1 and 9.")
                continue
        else:
            # Computer's move
            move = find_best_move(board)
            print(f"Computer chooses position {move + 1}")

        board[move] = current_player
        if check_win(board, current_player):
            print_board(board)
            print(f"Player {current_player} wins!")
            break
        elif check_draw(board):
            print_board(board)
            print("It's a draw!")
```

```
            break
        # Switch players
        current_player = 'O' if current_player == 'X' else 'X'
if _name_ == "_main_":
    play_game()
```

Assignment – 4

- **Code 1: A* algorithm using matplotlib.pyplot & numpy**

```
import matplotlib.pyplot as plt
import numpy as np
class Node:
    def __init__(self, position, parent=None):
        self.position = position
        self.parent = parent
        self.g = 0
        self.h = 0
        self.f = 0
    def __lt__(self, other):
        return self.f < other.f
def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
def astar(grid, start, end):
    open_list = []
    closed_set = set()
    start_node = Node(start)
    goal_node = Node(end)
    heapq.heappush(open_list, start_node)
    while open_list:
        current_node = heapq.heappop(open_list)
        closed_set.add(current_node.position)
```

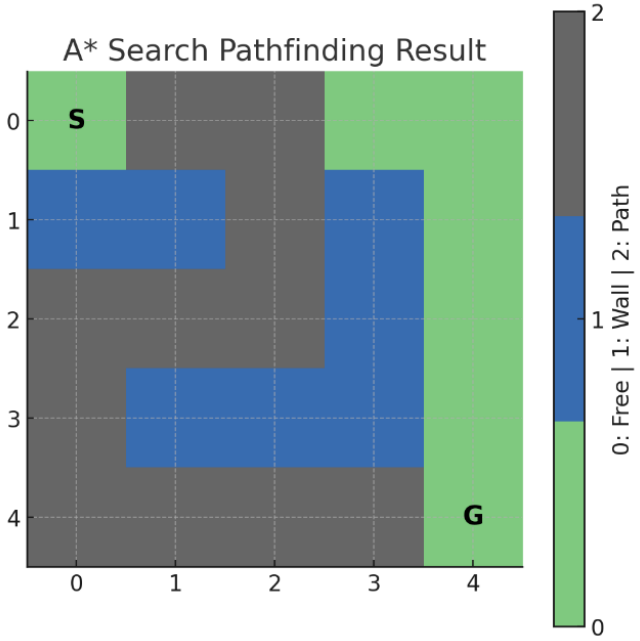
```
        if current_node.position == goal_node.position:
            path = []
            total_cost = current_node.g
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1], total_cost
        (x, y) = current_node.position
        neighbors = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]
        for next_pos in neighbors:
            (nx, ny) = next_pos
            if nx < 0 or ny < 0 or nx >= len(grid) or ny >= len(grid[0]):
                continue
            if grid[nx][ny] != 0:
                continue
            if next_pos in closed_set:
                continue
            neighbor_node = Node(next_pos, current_node)
            neighbor_node.g = current_node.g + 1
            neighbor_node.h = heuristic(next_pos, goal_node.position)
            neighbor_node.f = neighbor_node.g + neighbor_node.h
            if any(open_node.position == neighbor_node.position and open_node.f
<= neighbor_node.f for open_node in open_list):
                continue
            heapq.heappush(open_list, neighbor_node)
    return None, float('inf')
def plot_grid(grid, path, start, end):
    grid_display = np.array(grid)
```

```
    for (x, y) in path:
        if (x, y) != start and (x, y) != end:
            grid_display[x][y] = 2  # mark path as 2
    plt.figure(figsize=(6,6))
    plt.imshow(grid_display, cmap=plt.cm.get_cmap('Accent', 3))
    plt.xticks(range(len(grid[0])))
    plt.yticks(range(len(grid)))
    plt.grid(True)
    plt.text(start[1], start[0], 'S', ha='center', va='center', color='black',
fontsize=14, weight='bold')
    plt.text(end[1], end[0], 'G', ha='center', va='center', color='black', fontsize=14,
weight='bold')
    plt.title('A* Pathfinding Visualization')
    plt.colorbar(ticks=[0, 1, 2], label='0: Free, 1: Wall, 2: Path')
    plt.show()
# Define grid: 0 = free, 1 = wall
grid = [
    [0, 0, 0, 0, 0],
    [1, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0]
]
start = (0, 0)
end = (4, 4)
path, cost = astar(grid, start, end)
if path:
    print("Path found:", path)
    print("Minimum cost:", cost)
```

```
    plot_grid(grid, path, start, end)
else:
    print("No path found.")
```

Output : Path found: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (4, 4)]

Minimum cost from start to goal: 8



- **Code 2: A* algorithm using heapq**

```
import heapq
class Node:
    def __init__(self, position, parent=None):
        self.position = position
        self.parent = parent
        self.g = 0  # Cost from start to current node
        self.h = 0  # Heuristic cost to goal
        self.f = 0  # Total cost (g + h)
    def __lt__(self, other):
        return self.f < other.f
def heuristic(a, b):
```

```
    # Manhattan distance
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
def astar(grid, start, end):
    open_list = []
    closed_set = set()
    start_node = Node(start)
    goal_node = Node(end)
    heapq.heappush(open_list, start_node)
    while open_list:
        current_node = heapq.heappop(open_list)
        closed_set.add(current_node.position)
        if current_node.position == goal_node.position:
            path = []
            total_cost = current_node.g
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1], total_cost  # Return reversed path and cost
        (x, y) = current_node.position
        neighbors = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]

        for next_pos in neighbors:
            (nx, ny) = next_pos
            if nx < 0 or ny < 0 or nx >= len(grid) or ny >= len(grid[0]):
                continue
            if grid[nx][ny] != 0:
                continue
            if next_pos in closed_set:
                continue
```

```
            neighbor_node = Node(next_pos, current_node)
            neighbor_node.g = current_node.g + 1
            neighbor_node.h = heuristic(next_pos, goal_node.position)
            neighbor_node.f = neighbor_node.g + neighbor_node.h
            if any(open_node.position == neighbor_node.position and open_node.f
<= neighbor_node.f for open_node in open_list):
                continue
            heapq.heappush(open_list, neighbor_node)
    return None, float('inf')  # No path found
# Example grid: 0 = free, 1 = obstacle
grid = [
    [0, 0, 0, 0, 0],
    [1, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0]
]

start = (0, 0)
end = (4, 4)

path, cost = astar(grid, start, end)
if path:
    print("Path found:", path)
    print("Minimum cost from start to goal:", cost)
else:
    print("No path found.")
```

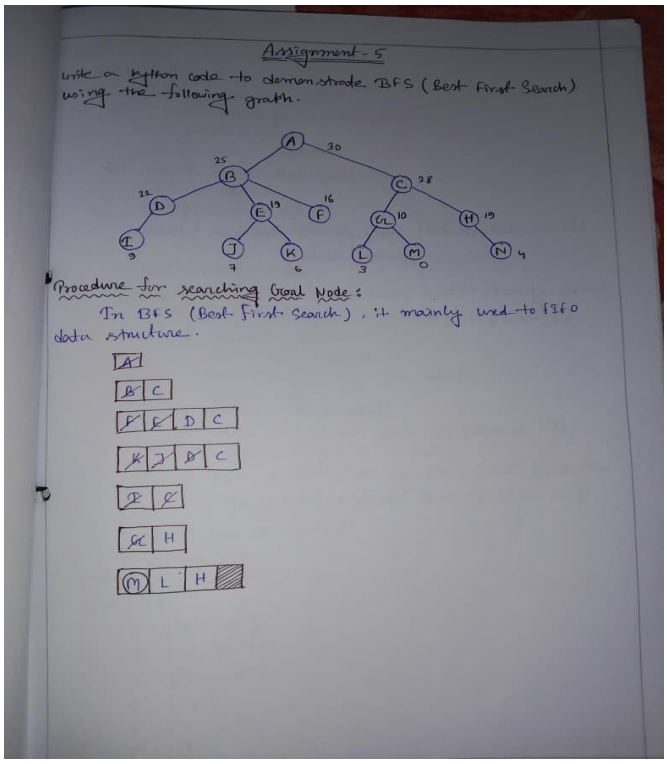
OUTPUT:

Path found: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (4, 4)]

Minimum cost from start to goal: 8

Assignment – 5

- **Code 1: Best First Search**




```
import heapq
class BestFirstSearch:
    def __init__(self, graph, heuristic):
        self.graph = graph              # Graph as adjacency list
        self.heuristic = heuristic      # Heuristic values {node: h(node)}
    def search(self, start, goal):
        visited = set()
        priority_queue = []             # Min-heap based on heuristic value
        heapq.heappush(priority_queue, (self.heuristic[start], start, [start]))
        while priority_queue:
            h_val, current, path = heapq.heappop(priority_queue)
            print(f"Visiting Node: {current} (h={h_val})")

            if current == goal:
                print("Goal reached!")
                print("Path:", " -> ".join(path))
                return path
            visited.add(current)
            # Expand neighbors
```

```
            for neighbor in self.graph.get(current, []):
                if neighbor not in visited:
                    heapq.heappush(
                        priority_queue,
                        (self.heuristic[neighbor], neighbor, path + [neighbor])
                    )
        print("Goal not found.")
        return None
graph = {
    'A': ['B', 'C', 'D'],
    'B': ['E', 'F'],
    'C': ['G', 'H'],
    'D': ['I'],
    'E': [],
    'F': [],
    'G': [],
    'H': [],
    'I': []
}
heuristic = {
    'A': 5, 'B': 4, 'C': 3, 'D': 7,
    'E': 1, 'F': 4, 'G': 2, 'H': 6, 'I': 3
}
bfs = BestFirstSearch(graph, heuristic)
bfs.search('A', 'E')
```

Output :

```
Visiting Node: A (h=5)
Visiting Node: C (h=3)
Visiting Node: G (h=2)
Visiting Node: B (h=4)
Visiting Node: E (h=1)
Goal reached!
Path: A -> B -> E
['A', 'B', 'E']
```

Assignment – 6

- **Code 1: Uniform Cost Search (UCS)**

```
import heapq
```

```
# Graph as adjacency list (node: [(cost, neighbor)])
graph = {
    'A': [(1, 'B'), (4, 'C')],
    'B': [(2, 'D'), (5, 'E')],
    'C': [(1, 'F')],
    'D': [],
    'E': [(1, 'G')],
    'F': [(3, 'G')],
    'G': []
}
def uniform_cost_search(start, goal):
    pq = []  # priority queue (min-heap)
    heapq.heappush(pq, (0, start, []))  # (cost, node, path)
    visited = set()
    while pq:
        cost, node, path = heapq.heappop(pq)
        if node in visited:
            continue
        visited.add(node)
        new_path = path + [node]
        if node == goal:
            return cost, new_path

        for edge_cost, neighbor in graph[node]:
            if neighbor not in visited:
                heapq.heappush(pq, (cost + edge_cost, neighbor, new_path))
    return None  # if goal not found
# Example run
start, goal = 'A', 'G'
cost, path = uniform_cost_search(start, goal)
print("Minimum cost from", start, "to", goal, ":", cost)
print("Path:", " -> ".join(path))
```

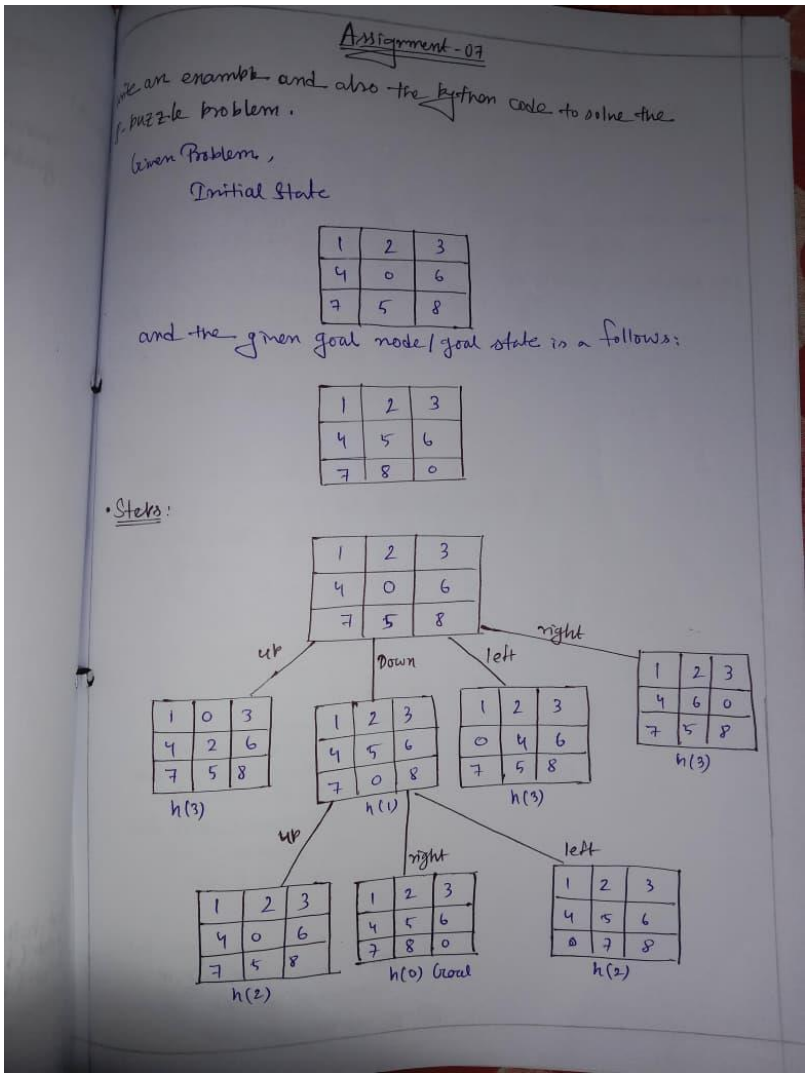
Output :

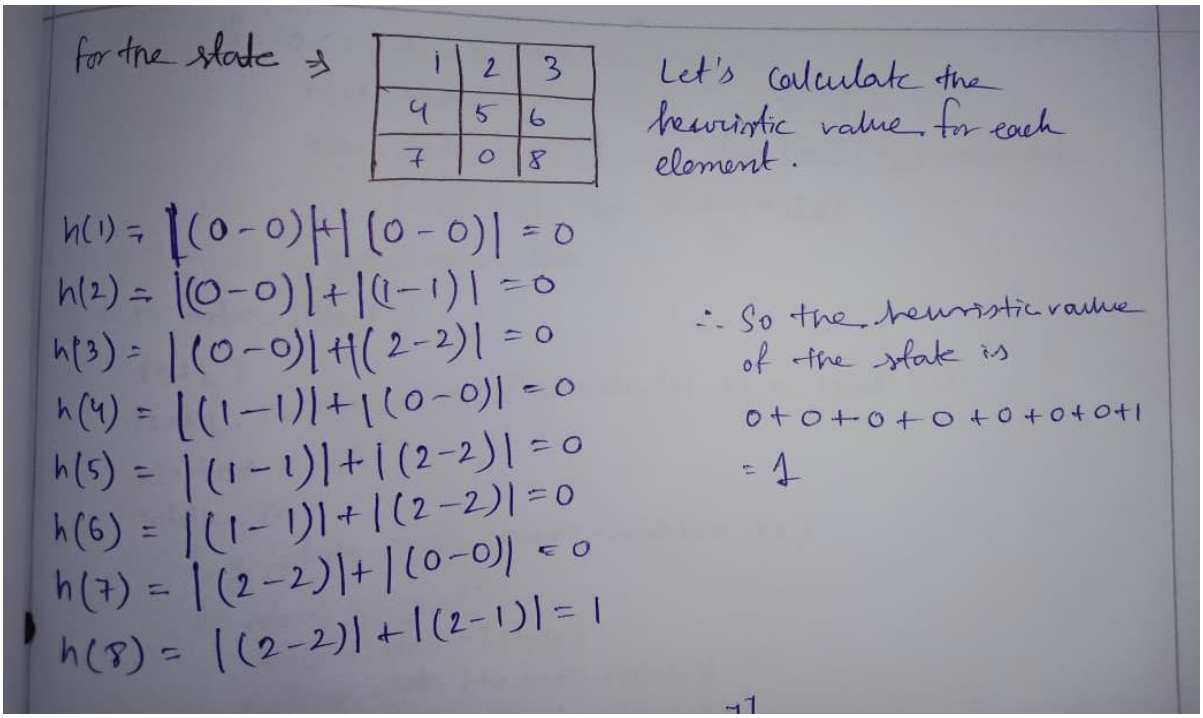
Minimum cost from A to G : 7

Path: A -> B -> E -> G

Assignment – 7

- **Code 1: 8 puzzle algorithm**





```
import heapq
# Goal state
goal = [[1,2,3],[8,0,4],[7,6,5]]
# Possible moves (up, down, left, right)
moves = [(-1,0),(1,0),(0,-1),(0,1)]
# Manhattan distance heuristic
def heuristic(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            val = state[i][j]
            if val != 0:
                x, y = divmod(val-1, 3)   # goal position
                distance += abs(x-i) + abs(y-j)
    return distance
# Convert to tuple (for hashing)
def to_tuple(state):
    return tuple(tuple(row) for row in state)
# Find blank position
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
# Generate neighbors
def neighbors(state):
    x, y = find_blank(state)
    for dx, dy in moves:
        nx, ny = x+dx, y+dy
```

```
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [row[:] for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny],
new_state[x][y]
            yield new_state
# A* algorithm
def astar(start):
    pq = []
    heapq.heappush(pq, (heuristic(start), 0, start, []))
    visited = set()
    while pq:
        f, g, state, path = heapq.heappop(pq)
        if state == goal:
            return path + [state]
        visited.add(to_tuple(state))
        for neigh in neighbors(state):
            if to_tuple(neigh) not in visited:
                heapq.heappush(pq, (g+1+heuristic(neigh), g+1, neigh, path+[state]))
# Example run
start = [[2,8,3],[1,6,4],[7,0,5]]
solution = astar(start)
print("Solution found in", len(solution)-1, "moves:")
for s in solution:
    for row in s:
        print(row)
    print()
```

Output :

```
Solution found in 5 moves:
[2, 8, 3]
```

[1, 6, 4]

[7, 0, 5]

[2, 8, 3]

[1, 0, 4]

[7, 6, 5]

[2, 0, 3]

[1, 8, 4]

[7, 6, 5]

[0, 2, 3]

[1, 8, 4]

[7, 6, 5]

[1, 2, 3]

[0, 8, 4]

[7, 6, 5]

[1, 2, 3]

[8, 0, 4]

[7, 6, 5]

Assignment – 8

- **Code 1: Hill climbing**

```
import random
# Objective function: f(x) = -(x^2) + 10
def objective_function(x):
    return -(x**2) + 10
def hill_climbing(max_iterations=1000, step_size=0.1):
```

```
    # Start with a random solution
    current_x = random.uniform(-10, 10)
    current_value = objective_function(current_x)
    for _ in range(max_iterations):
        # Generate two neighbors (left and right)
        left_x = current_x - step_size
        right_x = current_x + step_size
        left_value = objective_function(left_x)
        right_value = objective_function(right_x)
        # Choose the better neighbor
        if left_value > current_value or right_value > current_value:
            if left_value > right_value:
                current_x, current_value = left_x, left_value
            else:
                current_x, current_value = right_x, right_value
        else:
            # No better neighbor found → local maximum
            break
    return current_x, current_value
# Run the algorithm
best_x, best_value = hill_climbing()
print(f"Best solution found: x = {best_x:.4f}, f(x) = {best_value:.4f}")
```

Output: Best solution found: x = -0.0421, f(x) = 9.9982

- ## Code 2: Water Jug Problem

```
def water_jug_dfs(jug1, jug2, target, visited=None, path=None):
    if visited is None:
        visited = set()
    if path is None:
        path = []
    # Current state
    state = (jug1, jug2)
    # If target is reached
    if jug1 == target or jug2 == target:
        path.append(state)
        print("Solution found:")
        for step in path:
            print(step)
        return True
    # If already visited, return
```

```
    if state in visited:
        return False
    visited.add(state)
    path.append(state)
    # Generate possible next states
    possible_moves = [
        (4, jug2),  # Fill jug1 (capacity 4)
        (jug1, 3),  # Fill jug2 (capacity 3)
        (0, jug2),  # Empty jug1
        (jug1, 0),  # Empty jug2
        # Pour jug1 -> jug2
        (jug1 - min(jug1, 3 - jug2), jug2 + min(jug1, 3 - jug2)),
        # Pour jug2 -> jug1
        (jug1 + min(jug2, 4 - jug1), jug2 - min(jug2, 4 - jug1)),
    ]
    # Explore all possible moves
    for next_state in possible_moves:
        if water_jug_dfs(next_state[0], next_state[1], target, visited, path.copy()):
            return True
    return False
# Example usage
jug1_capacity = 4
jug2_capacity = 3
target = 2

if not water_jug_dfs(0, 0, target):
    print("No solution found.")
```

Output:

```
Solution found:
(0, 0)
(4, 0)
(4, 3)
(0, 3)
(3, 0)
(3, 3)
(4, 2)
```

Assignment – 9

- ## Code 1: Answer Extraction system

```
# Define Knowledge Base
knowledge_base = {
   "who is the president of india": "The President of India is Droupadi Murmu.",
   "what is the capital of india": "The capital of India is New Delhi.",
   "who is the prime minister of india": "The Prime Minister of India is
Narendra Modi.",
   "what is ai": "AI stands for Artificial Intelligence, the simulation of human
intelligence by machines.",
   "what is machine learning": "Machine Learning is a subset of AI that allows
systems to learn from data.",
   "what is the national animal of india": "The national animal of India is the
Bengal Tiger."
}
# user input
question = input("Ask a question: ").lower().strip()
# Search for answer
found = False
for key in knowledge_base.keys():
   if key in question or question in key:
      print("Answer:", knowledge_base[key])
      found = True
      break
# If not found
if not found:
   print("Answer not found in knowledge base.")
```

Output: Ask a question: What is the capital of INDIA
Answer: The capital of India is New Delhi.

- ## Code 2: Rule Based system

```
def diagnose(symptoms):
   if "fever" in symptoms and "cough" in symptoms and "tiredness" in
symptoms:
      return "You may have the Flu."
   elif "headache" in symptoms and "nausea" in symptoms:
      return "You may have a Migraine."
   elif "sneezing" in symptoms and "runny nose" in symptoms:
      return "You may have a Common Cold."
   else:
      return "Symptoms not recognized. Please consult a doctor."
```

```
# Input from user
symptoms = input("Enter your symptoms (comma separated): ")
symptoms = symptoms.lower().split(",")
symptoms = [s.strip() for s in symptoms]

print(diagnose(symptoms))
```

Output:

```
Enter your symptoms (comma separated):  fever, cough, tiredness
You may have the Flu.
```

- ## Code 3: KNN

```
# Simple dataset: [x, y, label]
dataset = [
    [2, 4, "A"],
    [4, 2, "A"],
    [4, 4, "A"],
    [6, 6, "B"],
    [6, 8, "B"],
    [8, 6, "B"]
]

new_point = [5, 5]

# Calculate Euclidean distances manually
distances = []
for point in dataset:
    x, y, label = point
    distance = ((x - new_point[0])**2 + (y - new_point[1])**2) ** 0.5
    distances.append([distance, label])

# Sort distances
distances.sort()

# k nearest neighbors
k = 3
neighbors = distances[:k]

# Count votes
votes = {}
```

```
for d, label in neighbors:
    if label in votes:
        votes[label] += 1
    else:
        votes[label] = 1

# Find the label with maximum votes
predicted_label = max(votes, key=votes.get)

print("Distances to dataset points:", distances)
print("Nearest neighbors:", neighbors)
print("Predicted class for point", new_point, "is", predicted_label)
```

Output:

```
Distances to dataset points: [[1.4142135623730951, 'A'],
[1.4142135623730951, 'B'], [3.1622776601683795, 'A'],
[3.1622776601683795, 'A'], [3.1622776601683795, 'B'],
[3.1622776601683795, 'B']]
Nearest neighbors: [[1.4142135623730951, 'A'], [1.4142135623730951, 'B'],
[3.1622776601683795, 'A']]
Predicted class for point [5, 5] is A
```