Department of Computer Science and Engineering

National Institute of Technology Srinagar

Hazratbal, Srinagar, Jammu and Kashmir - 190006, India.

LAB MANUAL

Course Name: Artificial Intelligence

Submitted by

AAYUSH SONI

2020BCSE020

III B.Tech. CSE (6th Semester)

Submitted to

Dr. Ranjeet Kumar Rout

Department of Computer Science and Engineering

National Institute of Technology Srinagar

Hazratbal, Srinagar, Jammu and Kashmir - 190006, India.

AUTUMN 2023

INDEX

Exercise 1

Implement *BFS* (Breadth First Search) and *DFS* (Depth First Search) using any graph.

CODE

```
def bfs(graph, start):
    visited = set()
    queue = [start]
    visited.add(start)

    while queue:
        vertex = queue.pop(0)
        print(vertex, end=" ")

        for neighbor in graph[vertex]:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)

graph = {
    1: [2, 3],
    2: [1, 4, 5],
    3: [1, 6],
    4: [2],
    5: [2, 6],
    6: [3, 5]
}

start_vertex = 1
print("DFS traversal:")
bfs(graph, start_vertex)

def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)
    print(start, end=" ")

    for neighbor in graph[start]:
        if neighbor not in visited:
```

```
            dfs(graph, neighbor, visited)

graph = {
    1: [2, 3],
    2: [1, 4, 5],
    3: [1, 6],
    4: [2],
    5: [2, 6],
    6: [3, 5]
}

start_vertex = 1
print("DFS traversal: ")
dfs(graph, start_vertex)
```

Output

```
BFS traversal:
1 2 3 4 5 6
DFS traversal:
1 2 4 5 6 3
```

Exercise 2

Implement *Floyd-Warshall* algorithm using any graph.

CODE

```
def floyd_warshall(graph):
    num_vertices = len(graph)

    dist = [[float('inf')] * num_vertices for _ in range(num_vertices)]
    for i in range(num_vertices):
        dist[i][i] = 0
        for j in graph[i]:
            dist[i][j] = graph[i][j]

    for k in range(num_vertices):
        for i in range(num_vertices):
            for j in range(num_vertices):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist

graph = {
    0: {0: 0, 1: 5, 3: 2},
    1: {1: 0, 2: 1},
    2: {2: 0, 3: 4},
    3: {3: 0}
}

distances = floyd_warshall(graph)
print("Distance matrix:")
for row in distances:
    print(row)
```

OUTPUT

```
Distance matrix:
[0,   5,   6,    2]
[inf, 0,   1,    5]
[inf, inf, 0,    4]
[inf, inf, inf,  0]
```

Exercise 3

Implement $A*$ algorithm using *Maze problem*. Approximate the value of *h* using following approximate heuristics:
1. Manhattan distance
2. Euclidean distance

Generate particular use cases where one is applicable and not the other.

CODE

```
import math
import heapq

class Cell:
    def __init__(self, row, col, is_wall):
        self.row = row
        self.col = col
        self.is_wall = is_wall
        self.g = float('inf')
        self.h = float('inf')
        self.f = float('inf')
        self.parent = None

    def __lt__(self, other):
        return self.f < other.f

def create_maze(rows, cols):
    maze = []
    for row in range(rows):
        maze.append([Cell(row, col, False) for col in range(cols)])
    return maze

def is_valid_cell(maze, row, col):
    return row >= 0 and row < len(maze) and col >= 0 and col < len(maze[0]) and not
maze[row][col].is_wall

def calculate_manhattan_distance(row1, col1, row2, col2):
    return abs(row1 - row2) + abs(col1 - col2)

def calculate_euclidean_distance(row1, col1, row2, col2):
    return math.sqrt((row1 - row2)**2 + (col1 - col2)**2)

def a_star(maze, start_row, start_col, end_row, end_col, heuristic):
    rows = len(maze)
    cols = len(maze[0])

    open_list = []
    closed_set = set()

    start_cell = maze[start_row][start_col]
```

```
    start_cell.g = 0
    start_cell.h = heuristic(start_row, start_col, end_row, end_col)
    start_cell.f = start_cell.h
    heapq.heappush(open_list, start_cell)

    while open_list:
        current_cell = heapq.heappop(open_list)
        closed_set.add(current_cell)

        if current_cell.row == end_row and current_cell.col == end_col:
            return construct_path(current_cell)

        for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            neighbor_row = current_cell.row + dr
            neighbor_col = current_cell.col + dc

            if is_valid_cell(maze, neighbor_row, neighbor_col):
                neighbor_cell = maze[neighbor_row][neighbor_col]

                if neighbor_cell in closed_set:
                    continue

                new_g = current_cell.g + 1

                if new_g < neighbor_cell.g:
                    neighbor_cell.g = new_g
                    neighbor_cell.parent = current_cell

                    if neighbor_cell not in open_list:
                        neighbor_cell.h = heuristic(neighbor_row, neighbor_col, end_row, end_col)
                        neighbor_cell.f = neighbor_cell.g + neighbor_cell.h
                        heapq.heappush(open_list, neighbor_cell)

    return None

def construct_path(cell):
    path = []
    while cell is not None:
        path.append((cell.row, cell.col))
        cell = cell.parent
    return list(reversed(path))

def print_maze(maze):
    for row in maze:
        for cell in row:
            if cell.is_wall:
                print('#', end=' ')
            elif cell.parent is not None:
```

```
                print('.', end=' ')
            else:
                print(' ', end=' ')
        print()

# Testing the algorithm
maze = create_maze(6, 6)
maze[0][2].is_wall = True
maze[1][2].is_wall = True
maze[2][2].is_wall = True
maze[3][2].is_wall = True
maze[4][2].is_wall = True

start_row, start_col = 0, 0
end_row, end_col = 5, 5

print("Maze:")
print_maze(maze)
print("")

print("A* Algorithm (Manhattan Distance):")
path_manhattan = a_star(maze, start_row, start_col, end_row, end_col,
calculate_manhattan_distance)
if path_manhattan:
    print("Path found!")
    for row, col in path_manhattan:
        maze[row][col].is_wall = False
    print_maze(maze)
else:
    print("Path not found.")

print("")

print("A* Algorithm (Euclidean Distance):")
path_euclidean = a_star(maze, start_row, start_col, end_row, end_col,
calculate_euclidean_distance)
if path_euclidean:
    print("Path found!")
    for row, col in path_euclidean:
        maze[row][col].is_wall = False
    print_maze(maze)
else:
    print("Path not found.")
```

OUTPUT

```
Maze:
    #
    #
    #
    #
    #


A* Algorithm (Manhattan Distance):
Path found!
  . #
. . #
. . #
. . #
. . # . .
. . . . . .

A* Algorithm (Euclidean Distance):
Path not found.
```

Exercise 4

Implement *Particle swarm optimization* algorithm using any *objective* function

CODE

```
import random
import math

class Particle:
    def __init__(self, position):
        self.position = position
        self.velocity = [random.uniform(-1, 1) for _ in range(len(position))]
        self.best_position = position
        self.best_fitness = float('inf')

def sphere_function(x, y):
    return x**2 + y**2

def pso(objective_function, num_particles, num_dimensions, num_iterations):
    # Initialize particles
    particles = []
    global_best_position = [0] * num_dimensions
    global_best_fitness = float('inf')

    for _ in range(num_particles):
        position = [random.uniform(-5, 5) for _ in range(num_dimensions)]
        particle = Particle(position)
        particles.append(particle)

        fitness = objective_function(*position)
        if fitness < particle.best_fitness:
            particle.best_fitness = fitness
            particle.best_position = position

        if fitness < global_best_fitness:
            global_best_fitness = fitness
            global_best_position = position

    # Perform iterations
    for _ in range(num_iterations):
        for particle in particles:
            for i in range(num_dimensions):
                # Update velocity
                r1 = random.random()
                r2 = random.random()

                cognitive_component = 2.0 * r1 * (particle.best_position[i] - particle.position[i])
                social_component = 2.0 * r2 * (global_best_position[i] - particle.position[i])
                particle.velocity[i] += cognitive_component + social_component
```

```
            # Update position
            particle.position[i] += particle.velocity[i]

        # Evaluate fitness
        fitness = objective_function(*particle.position)

        # Update personal best
        if fitness < particle.best_fitness:
            particle.best_fitness = fitness
            particle.best_position = particle.position

        # Update global best
        if fitness < global_best_fitness:
            global_best_fitness = fitness
            global_best_position = particle.position

    return global_best_position, global_best_fitness

# Testing the PSO algorithm
best_position, best_fitness = pso(sphere_function, num_particles=30, num_dimensions=2, num_iterations=100)

print("Best position:", best_position)
print("Best fitness:", best_fitness)
```

OUTPUT

```
Best position: [-308.5137909801842, -52.07985203952131]
Best fitness: 0.2313743571599284
```

Exercise 5

Implement $A*$ algorithm using *8-puzzle* problem

CODE

```
from heapq import heappop, heappush

class PuzzleState:
    def __init__(self, board, parent=None, action=None, cost=0):
        self.board = board
        self.parent = parent
        self.action = action
        self.cost = cost
        self.heuristic = self.calculate_heuristic()

    def __lt__(self, other):
        return self.cost + self.heuristic < other.cost + other.heuristic

    def calculate_heuristic(self):
        # Calculate the Manhattan distance heuristic
        distance = 0
        for i in range(3):
            for j in range(3):
                value = self.board[i][j]
                if value != 0:
                    target_row = (value - 1) // 3
                    target_col = (value - 1) % 3
                    distance += abs(i - target_row) + abs(j - target_col)
        return distance
    def is_goal_state(self):
        return self.board == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    def get_successors(self):
        successors = []
        zero_row, zero_col = self.find_zero_position()
        for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            new_row, new_col = zero_row + dr, zero_col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = [row[:] for row in self.board]
                new_board[zero_row][zero_col] = new_board[new_row][new_col]
                new_board[new_row][new_col] = 0
                successors.append(PuzzleState(new_board, self, (new_row, new_col), self.cost + 1))
        return successors
    def find_zero_position(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return i, j
    def get_path(self):
        path = []
```

```
        current = self
        while current.parent is not None:
            path.append(current.action)
            current = current.parent
        path.reverse()
        return path

def solve_puzzle(initial_board):
    initial_state = PuzzleState(initial_board)
    open_set = [initial_state]
    closed_set = set()
    while open_set:
        current_state = heappop(open_set)
        if current_state.is_goal_state():
            return current_state.get_path()
        closed_set.add(tuple(map(tuple, current_state.board)))
        for successor in current_state.get_successors():
            if tuple(map(tuple, successor.board)) in closed_set:
                continue

            existing_state = next((state for state in open_set if tuple(map(tuple, state.board)) ==
tuple(map(tuple, successor.board))), None)
            if existing_state is not None and successor.cost < existing_state.cost:
                open_set.remove(existing_state)

            heappush(open_set, successor)

    return None

# Testing the algorithm
initial_board = [[1, 2, 3], [4, 8, 0], [7, 6, 5]]
solution = solve_puzzle(initial_board)

if solution:
    print("Solution found!")
    for action in solution:
        print(f"Move zero to {action}")
else:
    print("No solution found.")
```

OUTPUT

```
Solution found!
Move zero to (2, 2)
Move zero to (2, 1)
Move zero to (1, 1)
Move zero to (1, 2)
Move zero to (2, 2)
```

Exercise 6

Implement $AO*$ algorithm using any graph.

CODE

```
def Cost(H, condition, weight = 1):
    cost = {}
    if 'AND' in condition:
        AND_nodes = condition['AND']
        Path_A = ' AND '.join(AND_nodes)
        PathA = sum(H[node]+weight for node in AND_nodes)
        cost[Path_A] = PathA

    if 'OR' in condition:
        OR_nodes = condition['OR']
        Path_B =' OR '.join(OR_nodes)
        PathB = min(H[node]+weight for node in OR_nodes)
        cost[Path_B] = PathB
    return cost

# Update the cost
def update_cost(H, Conditions, weight=1):
    Main_nodes = list(Conditions.keys())
    Main_nodes.reverse()
    least_cost= {}
    for key in Main_nodes:
        condition = Conditions[key]
        print(key,':', Conditions[key],'>>>', Cost(H, condition, weight))
        c = Cost(H, condition, weight)
        H[key] = min(c.values())
        least_cost[key] = Cost(H, condition, weight)
    return least_cost

# Print the shortest path
def shortest_path(Start,Updated_cost, H):
    Path = Start
    if Start in Updated_cost.keys():
        Min_cost = min(Updated_cost[Start].values())
        key = list(Updated_cost[Start].keys())
        values = list(Updated_cost[Start].values())
        Index = values.index(Min_cost)

        # FIND MINIMIMUM PATH KEY
        Next = key[Index].split()
        # ADD TO PATH FOR OR PATH
        if len(Next) == 1:

            Start =Next[0]
            Path += '<--' +shortest_path(Start, Updated_cost, H)
```

```
            # ADD TO PATH FOR AND PATH
            else:
                Path +='<--('+key[Index]+') '

                Start = Next[0]
                Path += '[' +shortest_path(Start, Updated_cost, H) + ' + '

                Start = Next[-1]
                Path += shortest_path(Start, Updated_cost, H) + ']'

    return Path



H = {'A': -1, 'B': 5, 'C': 2, 'D': 4, 'E': 7, 'F': 9, 'G': 3, 'H': 0, 'I':0, 'J':0}

Conditions = {
'A': {'OR': ['B'], 'AND': ['C', 'D']},
'B': {'OR': ['E', 'F']},
'C': {'OR': ['G'], 'AND': ['H', 'I']},
'D': {'OR': ['J']}
}
# weight
weight = 1
# Updated cost
print('Updated Cost :')
Updated_cost = update_cost(H, Conditions, weight=1)
print('*'*75)
print('Shortest Path :\n',shortest_path('A', Updated_cost,H))
```

OUTPUT

```
Updated Cost :
D : {'OR': ['J']} >>> {'J': 1}
C : {'OR': ['G'], 'AND': ['H', 'I']} >>> {'H AND I': 2, 'G':
4}
B : {'OR': ['E', 'F']} >>> {'E OR F': 8}
A : {'OR': ['B'], 'AND': ['C', 'D']} >>> {'C AND D': 5, 'B':
9}
***************************************************************
************
Shortest Path :
 A<--(C AND D) [C<--(H AND I) [H + I] + D<--J]
```

Exercise 7

Implement *CSP* on *8-queen* problem

CODE

```
class CSP:
    def __init__(self, n):
        self.n = n  # Number of queens
        self.board = [[0 for _ in range(n)] for _ in range(n)]
        self.solution = None

    def solve(self):
        if self.backtrack(0):
            return self.solution
        else:
            return []

    def backtrack(self, col):
        if col == self.n:
            self.solution = self.board.copy()
            return True

        for row in range(self.n):
            if self.is_safe(row, col):
                self.board[row][col] = 1

                if self.backtrack(col + 1):
                    return True

                self.board[row][col] = 0

        return False

    def is_safe(self, row, col):
        # Check if placing a queen at the given position is safe
        for c in range(col):
            # Check same row
            if self.board[row][c] == 1:
                return False

            # Check upper diagonal
            if row - (col - c) >= 0 and self.board[row - (col - c)][c] == 1:
                return False

            # Check lower diagonal
            if row + (col - c) < self.n and self.board[row + (col - c)][c] == 1:
                return False

        return True
```

```
    def print_solution(self):
        for row in self.solution:
            print(row)


# Example usage
csp = CSP(8)
solution = csp.solve()

if solution:
    print("Solution found:")
    csp.print_solution()
else:
    print("No solution found.")
```

OUTPUT

```
Solution found:
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
```

Exercise 8

Implement *Travelling Salesman Problem* using *Best first search* informed-search algorithm.

CODE

```
import heapq
def tsp_best_first(graph, start):
    # Heuristic function: Nearest Neighbor
    def nearest_neighbor(current_node, remaining_nodes):
        return min(remaining_nodes, key=lambda x: graph[current_node][x])

    priority_queue = []
    heapq.heappush(priority_queue, (0, [start]))  # (cost, path)

    while priority_queue:
        cost, path = heapq.heappop(priority_queue)
        current_node = path[-1]

        if len(path) == len(graph):
            return path, cost

        remaining_nodes = set(graph.keys()) - set(path)
        for node in remaining_nodes:
            new_cost = cost + graph[current_node][node]
            new_path = path + [node]
            heapq.heappush(priority_queue, (new_cost, new_path))

    return None  # No solution found

graph = {
    'A': {'B': 5, 'C': 3, 'D': 2},
    'B': {'A': 5, 'C': 2, 'D': 6},
    'C': {'A': 3, 'B': 2, 'D': 4},
    'D': {'A': 2, 'B': 6, 'C': 4}
}
start_node = 'A'
path, cost = tsp_best_first(graph, start_node)

if path:
    print("Optimal Path:", ' -> '.join(path))
    print("Total Cost:", cost)
else:
    print("No solution found.")
```

OUTPUT

```
Optimal Path: A -> D -> C -> B
Total Cost: 8
```

Exercise 9

Suppose we are working with the following knowledge base:

wizard(ron).
hasWand(harry).
quidditchPlayer(harry).
Wizard/1.(X) :- hasBroom(X),hasWand(X).
hasBroom(X) :- quidditchPlayer(X).

OUTPUT

wizard(ron)

true

witch(ron)

```
procedure `witch(A)' does not exist
```

wizard(hermoine)

false

witch(hermoine)

```
procedure `witch(A)' does not exist
```

wizard(harry)

false

wizard(Y)

Y = ron

witch(Y)

```
procedure `witch(A)' does not exist
```

Exercise 10

Implement *Simple linear regression (with 1 independent variable)* on any small datasets.

CODE

```
import numpy as np

# Dataset
X = np.array([1, 2, 3, 4, 5])
Y = np.array([2, 4, 6, 8, 10])

# Calculate the mean of X and Y
mean_x = np.mean(X)
mean_y = np.mean(Y)

# Calculate the differences from the mean
diff_x = X - mean_x
diff_y = Y - mean_y

# Calculate the slope (m)
m = np.sum(diff_x * diff_y) / np.sum(diff_x * diff_x)

# Calculate the y-intercept (c)
c = mean_y - m * mean_x

# Predict Y values for each X
Y_pred = m * X + c

# Print the slope and y-intercept
print("Slope (m):", m)
print("Y-intercept (c):", c)

# Print the predicted Y values
print("Predicted Y:", Y_pred)
```

OUTPUT

```
Slope (m): 2.0
Y-intercept (c): 0.0
Predicted Y: [ 2.  4.  6.  8. 10.]
```

CODE (using sklearn)

```
import numpy as np
from sklearn.linear_model import LinearRegression

# Dataset - House size in square feet (independent variable)
X = np.array([750, 900, 1200, 1500, 2000, 2500, 3000, 3500, 4000, 4500]).reshape(-1, 1)

# Target variable - House price in thousands of dollars (dependent variable)
y = np.array([100, 120, 170, 200, 250, 300, 350, 400, 450, 500])

# Create and fit the linear regression model
regression_model = LinearRegression()
regression_model.fit(X, y)

# Predict house prices for new data points
new_house_sizes = np.array([1000, 1800, 2800, 3800]).reshape(-1, 1)
predicted_prices = regression_model.predict(new_house_sizes)

print("Predicted house prices:")
for size, price in zip(new_house_sizes, predicted_prices):
    print(f"House size: {size[0]} sqft, Predicted price: {price:.2f} thousand dollars")
```

OUTPUT

```
Predicted house prices:
House size: 1000 sqft, Predicted price: 139.36 thousand dollars
House size: 1800 sqft, Predicted price: 222.91 thousand dollars
House size: 2800 sqft, Predicted price: 327.34 thousand dollars
House size: 3800 sqft, Predicted price: 431.77 thousand dollars
```

Exercise 11

Implement *logistic regression* on same dataset as in Exercise 10.

CODE

```
import numpy as np
from sklearn.linear_model import LogisticRegression

# Dataset - House size in square feet (independent variable)
X = np.array([750, 900, 1200, 1500, 2000, 2500, 3000, 3500, 4000, 4500]).reshape(-1, 1)

# Target variable - Binary labels: 0 (not buying) or 1 (buying)
y = np.array([0, 0, 0, 1, 1, 0, 1, 1, 1, 1])


logistic_model = LogisticRegression()
logistic_model.fit(X, y)

new_house_sizes = np.array([1000, 1800, 2800, 3800]).reshape(-1, 1)
probabilities = logistic_model.predict_proba(new_house_sizes)

print("Predicted probabilities of buying a house:")
for size, prob in zip(new_house_sizes, probabilities):
    print(f"House size: {size[0]} sqft, Probability: {prob[1]:.2f}")
```

OUTPUT

```
Predicted probabilities of buying a house:
House size: 1000 sqft, Probability: 0.17
House size: 1800 sqft, Probability: 0.48
House size: 2800 sqft, Probability: 0.85
House size: 3800 sqft, Probability: 0.97
```