**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**

|  |  |  |  |
| --- | --- | --- | --- |
| S.NO | EXERCISE | DATE | PAGE No. |
| 1 | Implement *BFS* (Breadth First Search) and *DFS* (Depth First Search) using any graph. |  | 3 |
| 2 | Implement *Floyd-Warshall* algorithm using any graph. |  | 5 |
| 3 | Implement 𝐴∗ algorithm using *Maze problem*. |  | 6 |
| 4 | Implement *Particle swarm optimization* algorithm using any *objective* function |  | 10 |
| 5 | Implement 𝐴∗ algorithm using *8-puzzle* problem |  | 12 |
| 6 | Implement 𝐴𝑂∗ algorithm using any graph. |  | 14 |
| 7 | Implement *CSP* on *8-queen problem* |  | 16 |
| 8 | Implement *Travelling Salesman Problem* using *Best first search* informed-search algorithm. |  | 18 |
| 9 | Prolog responses on various queries |  | 19 |
| 10 | Implement *Simple linear regression ( with 1 independent variable)* on any small datasets. |  | 20 |
| 11 | Implement *Simple logistic regression* |  | 22 |

**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