EXP 2 BFS DFS

from collections import deque

def bfs(graph, start, goal):

queue = deque([(start, [start])]) # Queue stores (current\_node, path\_to\_node)

visited = set()

while queue:

node, path = queue.popleft() # Get the first added node (FIFO order)

if node == goal:

return path # Return path when goal is reached

if node not in visited:

visited.add(node)

for neighbor in graph.get(node, []):

if neighbor not in visited:

queue.append((neighbor, path + [neighbor]))

return None # No path found

# Define the directed graph as an adjacency list

graph = {

'A': ['B', 'D'],

'B': ['C', 'F', 'E'],

'C': ['G'],

'D': ['F'],

'F': ['E'],

'E': ['G'],

'G': []

}

# Start BFS from 'A' to find a path to 'E'

path = bfs(graph, 'A', 'E')

print("Path from A to E:", path)

EXP 3 A\* algo  
from heapq import heappush, heappop

def astar(graph, start, goal):

# Priority queue to store (f\_score, node)

open\_set = []

# Dictionary to store g\_scores (path costs from start)

g\_score = {node: float('inf') for node in graph}

g\_score[start] = 0

# Dictionary to store parent nodes for path reconstruction

came\_from = {}

# Initial node with f\_score = g\_score + h\_score

heappush(open\_set, (g\_score[start] + graph[start]['h'], start))

while open\_set:

# Get node with lowest f\_score

current\_f, current = heappop(open\_set)

# If goal is reached

if current == goal:

path = []

while current in came\_from:

path.append(current)

current = came\_from[current]

path.append(start)

path.reverse()

return path, g\_score[goal]

# Check all neighbors

for neighbor, cost in graph[current]['edges'].items():

# Calculate tentative g\_score

tentative\_g = g\_score[current] + cost

# If this path is better than previous one

if tentative\_g < g\_score[neighbor]:

# Update path

came\_from[neighbor] = current

g\_score[neighbor] = tentative\_g

f\_score = tentative\_g + graph[neighbor]['h']

heappush(open\_set, (f\_score, neighbor))

return None, None

graph = {

'S': {'h': 7, 'edges': {'A': 4, 'B': 10, 'C': 11}},

'A': {'h': 8, 'edges': {'B': 8, 'D': 5}},

'B': {'h': 6, 'edges': {'D': 15, 'C': 8}},

'C': {'h': 5, 'edges': {'E': 20, 'F': 2}},

'D': {'h': 5, 'edges': {'H': 16, 'I': 1, 'F': 1}},

'E': {'h': 3, 'edges': {'G': 19}},

'F': {'h': 3, 'edges': {'G': 13}},

'G': {'h': 0, 'edges': {}},

'H': {'h': 7, 'edges': {'I': 1, 'J': 2}},

'I': {'h': 4, 'edges': {'J': 5, 'K': 13, 'G': 5}},

'J': {'h': 5, 'edges': {'K': 7}},

'K': {'h': 3, 'edges': {'G': 16}}

}

path, cost = astar(graph, 'S', 'G')

print(f"Path: {' -> '.join(path)}")

print(f"Total cost: {cost}")

EXP 4 8-Puzzle with A\*  
import copy

from heapq import heappush, heappop

class PuzzleState:

    def \_\_init\_\_(self, state, parent=None, action=None, cost=0):

        self.state = state

        self.parent = parent

        self.action = action

        self.cost = cost

        self.heuristic = self.manhattan\_distance()

    def \_\_lt\_\_(self, other):

        return (self.cost + self.heuristic) < (other.cost + other.heuristic)

    def find\_blank(self):

        for i in range(3):

            for j in range(3):

                if self.state[i][j] == 0:

                    return i, j

    def manhattan\_distance(self):

        distance = 0

        for i in range(3):

            for j in range(3):

                if self.state[i][j] != 0:

                    x, y = divmod(self.state[i][j] - 1, 3)

                    distance += abs(x - i) + abs(y - j)

        return distance

    def is\_goal(self):

        goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

        return self.state == goal

    def get\_neighbors(self):

        neighbors = []

        moves = {

            'UP': (-1, 0),

            'DOWN': (1, 0),

            'LEFT': (0, -1),

            'RIGHT': (0, 1)

        }

        x, y = self.find\_blank()

        for action, (dx, dy) in moves.items():

            new\_x, new\_y = x + dx, y + dy

            if 0 <= new\_x < 3 and 0 <= new\_y < 3:

                new\_state = copy.deepcopy(self.state)

                new\_state[x][y], new\_state[new\_x][new\_y] = new\_state[new\_x][new\_y], new\_state[x][y]

                neighbors.append(PuzzleState(new\_state, self, action, self.cost + 1))

        return neighbors

def solve\_puzzle(initial\_state):

    start = PuzzleState(initial\_state)

    frontier = []

    heappush(frontier, start)

    explored = set()

    while frontier:

        current = heappop(frontier)

        if current.is\_goal():

            path = []

            while current.parent is not None:

                path.append((current.action, current.state))

                current = current.parent

            return path[::-1]

        state\_tuple = tuple(map(tuple, current.state))

        if state\_tuple in explored:

            continue

        explored.add(state\_tuple)

        for neighbor in current.get\_neighbors():

            heappush(frontier, neighbor)

    return None

def print\_state(state):

    for row in state:

        print(row)

    print()

def get\_user\_input():

    print("Enter the initial state of the 8-puzzle (use 0 for empty space)")

    print("Enter numbers row by row (space-separated)")

    initial\_state = []

    for i in range(3):

        row = list(map(int, input(f"Enter row {i+1}: ").strip().split()))

        initial\_state.append(row)

    return initial\_state

def main():

    print("Welcome to 8-Puzzle Solver!")

    initial\_state = get\_user\_input()

    print("\nInitial State:")

    print\_state(initial\_state)

    solution = solve\_puzzle(initial\_state)

    if solution:

        print("Solution found! Here are the steps:")

        current\_state = initial\_state

        for step, (action, new\_state) in enumerate(solution, 1):

            print(f"\nStep {step}: {action}")

            print\_state(new\_state)

            current\_state = new\_state

    else:

        print("No solution found!")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

EXP 5 Hanoi   
public class Tower {

    public static void towerOfHanoi(int n, String src, String helper, String des) {

        if(n == 1) {

            System.out.println("Transfer disk " + n  + " from tower " + src + " to " + des);

            return;

        }

        towerOfHanoi(n-1, src, des, helper);

        System.out.println("Transfer disk " + n + " from tower " + src + " to " + des);

        towerOfHanoi(n-1, helper, src, des);

    }

    public static void main(String args[]) {

        int n = 3;

        towerOfHanoi(n, "S", "H", "D");

    }

}

EXP 6 Knowledge base in prolog  
% pair(symbol,symbol)

iskiller(symbol,symbol)

male(symbol)

female(symbol)

isvictim(symbol)

not\_at\_bar(symbol,symbol)

not\_at\_beach(symbol,symbol)

not\_alone(symbol)

twin(symbol,symbol)

younger(symbol,symbol)

child(symbol)

clauses

male(husband).

male(brother).

male(son).

female(alice).

female(daughter).

twin(brother,alice).

twin(son,daughter).

child(son).

child(daughter).

Medical Dignosis

% Facts: Symptoms associated with patients

has\_symptom(patient1, fever).

has\_symptom(patient1, cough).

has\_symptom(patient2, headache).

has\_symptom(patient2, fatigue).

has\_symptom(patient3, cough).

has\_symptom(patient3, sore\_throat).

has\_symptom(patient4, headache).

has\_symptom(patient4, nausea).

has\_symptom(patient5, fever).

has\_symptom(patient5, chills).

has\_symptom(patient6, sore\_throat).

has\_symptom(patient6, cough).

% Rules: Diagnoses based on symptoms

illness(flu) :- has\_symptom(X, fever), has\_symptom(X, cough), has\_symptom(X, chills).

illness(cold) :- has\_symptom(X, cough), has\_symptom(X, sore\_throat).

illness(migraine) :- has\_symptom(X, headache), has\_symptom(X, nausea).

illness(covid) :- has\_symptom(X, fever), has\_symptom(X, cough), has\_symptom(X, chills), has\_symptom(X, sore\_throat).

**Code 2:**

**% Facts about diseases**

**disease(flu).**

**disease(cold).**

**disease(pneumonia).**

**% Facts about symptoms**

**symptom(flu, fever).**

**symptom(flu, cough).**

**symptom(flu, fatigue).**

**symptom(cold, cough).**

**symptom(cold, sore\_throat).**

**symptom(pneumonia, fever).**

**symptom(pneumonia, cough).**

**symptom(pneumonia, difficulty\_breathing).**

**% Facts about patient symptoms**

**has\_symptom(john, fever).**

**has\_symptom(john, cough).**

**has\_symptom(john, fatigue).**

**has\_symptom(mary, cough).**

**has\_symptom(mary, sore\_throat).**

**% Rules for diagnosis**

**diagnosis(Patient, flu) :-**

**has\_symptom(Patient, fever),**

**has\_symptom(Patient, cough),**

**has\_symptom(Patient, fatigue).**

**diagnosis(Patient, cold) :-**

**has\_symptom(Patient, cough),**

**has\_symptom(Patient, sore\_throat).**

**diagnosis(Patient, pneumonia) :-**

**has\_symptom(Patient, fever),**

**has\_symptom(Patient, cough),**

**has\_symptom(Patient, difficulty\_breathing).**

EXP 7 fwd chaining for medical using prolog  
% Facts defining initial symptoms

symptom\_present(fever).

symptom\_present(cough).

% Forward chaining rules for medical diagnosis

diagnose :-

symptom\_present(fever),

symptom\_present(cough),

assert(disease(flu)),

write('The patient may have flu.'), nl.

diagnose :-

symptom\_present(sneezing),

symptom\_present(runny\_nose),

assert(disease(common\_cold)),

write('The patient may have common cold.'), nl.

diagnose :-

symptom\_present(fever),

symptom\_present(cough),

symptom\_present(loss\_of\_taste),

assert(disease(covid\_19)),

write('The patient may have COVID-19.'), nl.

% Sample Query

% ?- diagnose.

EXP 8 Bayesian reasoning for probabilistic inference for Weather prediction.  
import numpy as np

def bayesian\_update(prior, likelihood, evidence):

"""

Perform Bayesian update.

:param prior: Prior probability P(H)

:param likelihood: Likelihood P(E|H)

:param evidence: Total probability P(E)

:return: Posterior probability P(H|E)

"""

return (likelihood \* prior) / evidence

# Get user input for prior probabilities

prior\_sunny = float(input("Enter prior probability of a sunny day: "))

prior\_rainy = float(input("Enter prior probability of a rainy day: "))

# Get user input for likelihoods

likelihood\_sunny = float(input("Enter likelihood of seeing a clear sky given a sunny day: "))

likelihood\_rainy = float(input("Enter likelihood of seeing a clear sky given a rainy day: "))

# Compute total probability of the evidence (marginal likelihood)

evidence = (likelihood\_sunny \* prior\_sunny) + (likelihood\_rainy \* prior\_rainy)

# Compute posterior probabilities

posterior\_sunny = bayesian\_update(prior\_sunny, likelihood\_sunny, evidence)

posterior\_rainy = bayesian\_update(prior\_rainy, likelihood\_rainy, evidence)

print(f"Updated probability of a sunny day given clear sky: {posterior\_sunny:.2f}")

print(f"Updated probability of a rainy day given clear sky: {posterior\_rainy:.2f}")

EXP 9 Implement a rule-based AI to play a Tic-Tac-Toe game.

% Define winning conditions win(Player, Board) :-

(Board = [Player, Player, Player, \_, \_, \_, \_, \_, \_]);

(Board = [\_, \_, \_, Player, Player, Player, \_, \_, \_]);

(Board = [\_, \_, \_, \_, \_, \_, Player, Player, Player]);

(Board = [Player, \_, \_, Player, \_, \_, Player, \_, \_]);

(Board = [\_, Player, \_, \_, Player, \_, \_, Player, \_]);

(Board = [\_, \_, Player, \_, \_, Player, \_, \_, Player]);

(Board = [Player, \_, \_, \_, Player, \_, \_, \_, Player]);

(Board = [\_, \_, Player, \_, Player, \_, Player, \_, \_]).

% Check if position is free free(Position, Board) :-

nth0(Position, Board, empty).

% Find winning move for AI

winning\_move(Board, Player, Move) :- free(Move, Board),

nth0(Move, Board, Player, NewBoard), win(Player, NewBoard).

% Find blocking move to prevent opponent from winning blocking\_move(Board, Move) :-

winning\_move(Board, o, Move).

% Take the center if available center\_move(Board, Move) :-

nth0(4, Board, empty), Move = 4.

% Take a corner if available corner\_move(Board, Move) :-

member(Move, [0, 2, 6, 8]), nth0(Move, Board, empty).

% Pick first available move as fallback first\_available(Board, Move) :-

nth0(Move, Board, empty).

% AI move: Prioritize winning, blocking, center, corner, and fallback best\_move(Board, Move) :-

(winning\_move(Board, x, Move), !); (blocking\_move(Board, Move), !); (center\_move(Board, Move), !); (corner\_move(Board, Move), !); first\_available(Board, Move).