ai

1) non ai

1a tictactoe:

def print\_board(board):

"""Prints the current state of the board."""

print(" 0 1 2")

for i, row in enumerate(board):

print(f"{i} {' | '.join(row)}")

if i < 2:

print(" ---+---+---")

def check\_winner(board, player):

"""Checks if the given player has won."""

for i in range(3):

if all([cell == player for cell in board[i]]):

return True

if all([board[j][i] == player for j in range(3)]):

return True

if all([board[i][i] == player for i in range(3)]):

return True

if all([board[i][2 - i] == player for i in range(3)]):

return True

return False

def is\_full(board):

"""Checks if the board is full."""

return all([cell != ' ' for row in board for cell in row])

def main():

"""Main function to play Tic Tac Toe."""

board = [[' ' for \_ in range(3)] for \_ in range(3)]

current\_player = 'X'

print("Welcome to Tic Tac Toe!")

print\_board(board)

while True:

try:

print(f"Player {current\_player}, it's your turn.")

row = int(input("Enter the row (0-2): "))

col = int(input("Enter the column (0-2): "))

if board[row][col] == ' ':

board[row][col] = current\_player

print\_board(board)

if check\_winner(board, current\_player):

print(f"Player {current\_player} wins!")

break

if is\_full(board):

print("It's a draw!")

break

current\_player = 'O' if current\_player == 'X' else 'X'

else:

print("Cell already taken. Choose a different cell.")

except (ValueError, IndexError):

print("Invalid input. Please enter a row and column between 0 and 2.")

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

main()

1b) nqueen:

def print\_board(board):

"""Prints the N-Queens board."""

for row in board:

print(" ".join("Q" if cell else "." for cell in row))

print("\n")

def is\_safe(board, row, col, n):

"""Checks if placing a queen at (row, col) is safe."""

# Check the column

for i in range(row):

if board[i][col]:

return False

# Check the upper-left diagonal

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j]:

return False

# Check the upper-right diagonal

for i, j in zip(range(row, -1, -1), range(col, n)):

if board[i][j]:

return False

return True

def solve\_nqueens(board, row, n):

"""Solves the N-Queens problem using backtracking."""

if row == n:

print\_board(board)

return True

success = False

for col in range(n):

if is\_safe(board, row, col, n):

board[row][col] = 1 # Place the queen

success = solve\_nqueens(board, row + 1, n) or success

board[row][col] = 0 # Backtrack

return success

def n\_queens(n):

"""Main function to initialize the board and solve the problem."""

board = [[0 for \_ in range(n)] for \_ in range(n)]

if not solve\_nqueens(board, 0, n):

print("No solution exists.")

# Example usage:

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

N = int(input("Enter the value of N: "))

n\_queens(N)

1c magicsquare

P:

def print\_board(board):

"""Prints the N-Queens board."""

for row in board:

print(" ".join("Q" if cell else "." for cell in row))

print("\n")

def is\_safe(board, row, col, n):

"""Checks if placing a queen at (row, col) is safe."""

# Check the column

for i in range(row):

if board[i][col]:

return False

# Check the upper-left diagonal

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j]:

return False

# Check the upper-right diagonal

for i, j in zip(range(row, -1, -1), range(col, n)):

if board[i][j]:

return False

return True

def solve\_nqueens(board, row, n):

"""Solves the N-Queens problem using backtracking."""

if row == n:

print\_board(board)

return True

success = False

for col in range(n):

if is\_safe(board, row, col, n):

board[row][col] = 1 # Place the queen

success = solve\_nqueens(board, row + 1, n) or success

board[row][col] = 0 # Backtrack

return success

def n\_queens(n):

"""Main function to initialize the board and solve the problem."""

board = [[0 for \_ in range(n)] for \_ in range(n)]

if not solve\_nqueens(board, 0, n):

print("No solution exists.")

# Example usage:

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

N = int(input("Enter the value of N: "))

n\_queens(N)

2) waterjug :

1a)dfs

def water\_jug\_dfs(capacity1, capacity2, target):

visited = set()

parent = {}

def dfs(jug1, jug2):

if (jug1, jug2) in visited:

return False

visited.add((jug1, jug2))

if jug1 == target or jug2 == target:

return True

next\_states = [

(capacity1, jug2),

(jug1, capacity2),

(0, jug2),

(jug1, 0),

(max(0, jug1 - (capacity2 - jug2)), min(capacity2, jug1 + jug2)),

(min(capacity1, jug1 + jug2), max(0, jug2 - (capacity1 - jug1)))

]

for state in next\_states:

if state not in visited:

parent[state] = (jug1, jug2)

if dfs(\*state):

return True

return False

start = (0, 0)

if dfs(\*start):

path = []

current = next((s for s in visited if s[0] == target or s[1] == target), None)

while current in parent:

path.append(current)

current = parent[current]

path.append(start)

return path[::-1]

else:

return None

# Take user input

try:

capacity1 = int(input("Enter capacity of Jug 1: "))

capacity2 = int(input("Enter capacity of Jug 2: "))

target = int(input("Enter target amount: "))

except ValueError:

print("Invalid input. Please enter integers only.")

exit()

# Solve

solution = water\_jug\_dfs(capacity1, capacity2, target)

# Display

if solution:

print("\nSteps to reach the target:")

for step in solution:

print(f"Jug1: {step[0]}, Jug2: {step[1]}")

else:

print("No solution found.")

input:

Enter capacity of Jug 1: 4

Enter capacity of Jug 2: 3

Enter target amount: 2

2b) bfs:

from collections import deque

def water\_jug\_bfs(capacity1, capacity2, target):

visited = set()

queue = deque([(0, 0)])

parent = {}

while queue:

jug1, jug2 = queue.popleft()

if jug1 == target or jug2 == target:

path = []

while (jug1, jug2) in parent:

path.append((jug1, jug2))

jug1, jug2 = parent[(jug1, jug2)]

path.append((0, 0))

return path[::-1]

if (jug1, jug2) in visited:

continue

visited.add((jug1, jug2))

next\_states = [

(capacity1, jug2),

(jug1, capacity2),

(0, jug2),

(jug1, 0),

(max(0, jug1 - (capacity2 - jug2)), min(capacity2, jug1 + jug2)),

(min(capacity1, jug1 + jug2), max(0, jug2 - (capacity1 - jug1)))

]

for state in next\_states:

if state not in visited:

queue.append(state)

parent[state] = (jug1, jug2)

return None

# Take user input

try:

capacity1 = int(input("Enter capacity of Jug 1: "))

capacity2 = int(input("Enter capacity of Jug 2: "))

target = int(input("Enter target amount: "))

except ValueError:

print("Invalid input. Please enter integers only.")

exit()

# Solve

solution = water\_jug\_bfs(capacity1, capacity2, target)

# Display

if solution:

print("\nSteps to reach the target:")

for step in solution:

print(f"Jug1: {step[0]}, Jug2: {step[1]}")

else:

print("No solution found.")

3) hillclimbing:

import copy

import random

GOAL\_STATE = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

def input\_puzzle():

print("Enter the 8-puzzle initial state (use 0 for blank):")

state = []

for i in range(3):

row = list(map(int, input(f"Row {i + 1} (space-separated): ").split()))

state.append(row)

return state

def print\_state(state):

for row in state:

print(' '.join(str(x) for x in row))

print()

def get\_blank\_position(state):

for i in range(3):

for j in range(3):

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

return i, j

def heuristic(state):

"""Number of misplaced tiles."""

count = 0

for i in range(3):

for j in range(3):

if state[i][j] != 0 and state[i][j] != GOAL\_STATE[i][j]:

count += 1

return count

def get\_neighbors(state):

x, y = get\_blank\_position(state)

neighbors = []

moves = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Up Down Left Right

for dx, dy in moves:

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

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

new\_state = copy.deepcopy(state)

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

neighbors.append(new\_state)

return neighbors

def hill\_climbing(start):

current = start

current\_h = heuristic(current)

steps = 0

while True:

neighbors = get\_neighbors(current)

next\_state = None

next\_h = current\_h

for neighbor in neighbors:

h = heuristic(neighbor)

if h < next\_h:

next\_state = neighbor

next\_h = h

if next\_state is None:

break # Local minimum

current = next\_state

current\_h = next\_h

steps += 1

print(f"Step {steps}: (Heuristic = {current\_h})")

print\_state(current)

if current\_h == 0:

print("Goal reached!")

return

print("Stuck at local minimum, solution not found.")

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

initial\_state = input\_puzzle()

print("\nInitial State:")

print\_state(initial\_state)

hill\_climbing(initial\_state)

input:

Enter the 8-puzzle initial state (use 0 for blank):

Row 1 (space-separated): 1 2 3

Row 2 (space-separated): 4 0 6

Row 3 (space-separated): 7 5 8

4) bfs to solve :

4a)8 puzzle

import heapq # Import the heapq module to use a priority queue (min-heap)

import copy # Import the copy module for deep copying lists

# Define the goal state of the puzzle

goal\_state = [[1, 2, 3],

[4, 5, 6],

[7, 8, 0]] # 0 represents the empty tile

# Define the possible moves for the empty tile (right, down, left, up)

moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]

def manhattan(state):

"""Calculate the Manhattan distance heuristic for the current state."""

dist = 0 # Initialize the total Manhattan distance

for i in range(3): # Loop over each row

for j in range(3): # Loop over each column

val = state[i][j] # Get the value at position (i, j)

if val != 0: # If it's not the empty tile (0)

# Calculate the goal position of the tile

goal\_x, goal\_y = divmod(val - 1, 3) # divmod gives row and column in the goal state

# Add the Manhattan distance to the total

dist += abs(i - goal\_x) + abs(j - goal\_y)

return dist # Return the total Manhattan distance

def get\_neighbors(state):

"""Generate all valid neighboring states by moving the empty tile."""

neighbors = [] # List to store all valid neighboring states

# Find the position of the blank tile (0)

for i in range(3):

for j in range(3):

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

x, y = i, j # Store the coordinates of the blank tile

# Loop through the possible moves (right, down, left, up)

for dx, dy in moves:

nx, ny = x + dx, y + dy # Calculate the new position of the blank tile

# Check if the new position is within bounds (0 <= nx, ny < 3)

if 0 <= nx < 3 and 0 <= ny < 3:

new\_state = copy.deepcopy(state) # Make a deep copy of the current state

# Swap the blank tile with the adjacent tile

new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]

neighbors.append(new\_state) # Add the new state to the list of neighbors

return neighbors # Return the list of neighbors

def print\_state(state):

"""Print the current state in a readable format."""

for row in state:

print(row) # Print each row of the state

print() # Print a blank line for separation

def best\_first\_search(start):

"""Perform the best-first search algorithm to find the solution."""

visited = set() # Set to keep track of visited states

pq = [] # Priority queue to store states based on their heuristic value (Manhattan distance)

# Push the initial state to the priority queue with its heuristic value (Manhattan distance)

heapq.heappush(pq, (manhattan(start), start))

steps = 0 # Initialize step counter for tracking the number of steps

while pq: # While there are still states in the priority queue

# Pop the state with the lowest heuristic (Manhattan distance) from the queue

cost, state = heapq.heappop(pq)

# Convert the state into a tuple (to be hashable) for easy lookup in the visited set

state\_tuple = tuple(tuple(row) for row in state)

# If the state has already been visited, skip it

if state\_tuple in visited:

continue

visited.add(state\_tuple) # Mark this state as visited

# Print the current step and the cost (Manhattan distance) for this state

print(f"Step {steps}, Cost (h) = {cost}")

print\_state(state) # Print the current state

steps += 1 # Increment the step counter

# If the goal state is reached, print success message and exit

if state == goal\_state:

print(" Goal Reached!")

return

# Loop through all the valid neighboring states

for neighbor in get\_neighbors(state):

# Convert the neighbor state into a tuple for easy lookup in the visited set

neighbor\_tuple = tuple(tuple(row) for row in neighbor)

# If the neighbor hasn't been visited, push it to the priority queue

if neighbor\_tuple not in visited:

heapq.heappush(pq, (manhattan(neighbor), neighbor))

# If the priority queue is empty and the goal hasn't been found, print failure message

print("No solution found.")

# Function to take user input for the initial state

def get\_user\_input():

print("Enter the initial state of the puzzle:")

puzzle = []

for i in range(3):

row = list(map(int, input(f"Enter row {i + 1} (3 numbers separated by space): ").split()))

puzzle.append(row)

return puzzle

# Get the initial state from the user

initial\_state = get\_user\_input()

# Start the best-first search with the initial state

best\_first\_search(initial\_state)

input:

"""Enter row 1 (3 numbers separated by space): 1 2 3

Enter row 2 (3 numbers separated by space): 4 5 6

Enter row 3 (3 numbers separated by space): 0 7 8"""

4b)robot navigation:

import heapq

def heuristic(a, b):

return abs(a[0] - b[0]) + abs(a[1] - b[1]) # Manhattan distance

def print\_grid(grid, path\_positions):

for i in range(len(grid)):

for j in range(len(grid[0])):

if (i, j) in path\_positions:

print("R", end=" ") # Mark the robot's path

elif grid[i][j] == 1:

print("X", end=" ") # Mark obstacles

else:

print(".", end=" ") # Open space

print()

print("\n" + "-" \* 20 + "\n")

def best\_first\_search\_robot(grid, start, goal):

pq = []

heapq.heappush(pq, (0, start))

visited = set()

parent = {start: None}

while pq:

\_, current = heapq.heappop(pq)

if current == goal:

path = []

while current:

path.append(current)

current = parent[current]

return path[::-1] # Reverse the path to start from initial position

visited.add(current)

x, y = current

moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]

for dx, dy in moves:

nx, ny = x + dx, y + dy

if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]) and grid[nx][ny] == 0 and (nx, ny) not in visited:

heapq.heappush(pq, (heuristic((nx, ny), goal), (nx, ny)))

parent[(nx, ny)] = current

return None

# User Input

print("Enter grid size (rows and cols):")

rows, cols = map(int, input().split())

grid = []

print("Enter grid (0 for free space, 1 for obstacle):")

for \_ in range(rows):

grid.append(list(map(int, input().split())))

print("Enter start position (x y):")

sx, sy = map(int, input().split())

print("Enter goal position (x y):")

gx, gy = map(int, input().split())

path = best\_first\_search\_robot(grid, (sx, sy), (gx, gy))

if path:

print("\nStep-by-step robot navigation:\n")

for step in path:

print(f"Robot moves to: {step}")

print\_grid(grid, path[:path.index(step) + 1])

print("Goal Reached!")

else:

print("No path found!")

input:

Enter grid size (rows and cols):

3 3

Enter grid (0 for free space, 1 for obstacle):

0 0 0

1 1 0

0 0 0

Enter start position (x y):

0 0

Enter goal position (x y):

2 2

4c)cities

import heapq

def best\_first\_search\_cities(graph, start, goal):

pq = []

heapq.heappush(pq, (0, start))

visited = set()

parent = {start: None}

steps = [] # To store step-by-step traversal

#pop city with lowest cost

while pq:

cost, current = heapq.heappop(pq)

steps.append(f"Visiting: {current} (Cost: {cost})")

#if reached goal go to parent

if current == goal:

path = []

while current:

path.append(current)

current = parent[current]

return path[::-1], steps

visited.add(current)

for neighbor, dist in graph[current].items():

if neighbor not in visited:

heapq.heappush(pq, (dist, neighbor))

parent[neighbor] = current

return None, steps

# User Input

print("Enter number of cities:")

n = int(input())

graph = {}

for \_ in range(n):

city = input("Enter city name: ")

graph[city] = {}

print("Enter roads (city1 city2 distance), type 'done' to stop:")

while True:

data = input()

if data.lower() == "done":

break

city1, city2, dist = data.split()

dist = int(dist)

graph[city1][city2] = dist

graph[city2][city1] = dist # Assuming bidirectional roads

print("Enter start city:")

start = input()

print("Enter goal city:")

goal = input()

path, steps = best\_first\_search\_cities(graph, start, goal)

# Step-by-Step Output

print("\nStep-by-step traversal:")

for step in steps:

print(step)

# Final Path Output

if path:

print("\nShortest Path Found:", " → ".join(path))

else:

print("\nNo path found!")

input:

Enter number of cities:

4

Enter city name: A

Enter city name: B

Enter city name: C

Enter city name: D

Enter roads (city1 city2 distance), type 'done' to stop:

A B 1

A C 4

B D 2

C D 1

done

Enter start city:

A

Enter goal city:

D

5a) a\*

npuzzle

a:

from copy import deepcopy

from queue import PriorityQueue

import math

class Node:

def \_\_init\_\_(self, state, parent, move, depth, cost):

self.state = state

self.parent = parent

self.move = move

self.depth = depth

self.cost = cost

def \_\_lt\_import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

class Node:

def \_\_init\_\_(self, state, parent, move, depth, cost):

self.state = state

self.parent = parent

self.move = move

self.depth = depth

self.cost = cost

def \_\_lt\_\_(self, other):

return self.cost < other.cost

class NPuzzle:

def \_\_init\_\_(self, start, goal):

self.start = start

self.goal = goal

self.size = len(start)

self.moves = []

self.directions = {

'UP': (-1, 0),

'DOWN': (1, 0),

'LEFT': (0, -1),

'RIGHT': (0, 1)

}

def find\_blank(self, state):

for i in range(self.size):

for j in range(self.size):

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

return i, j

def move\_tile(self, state, direction):

x, y = self.find\_blank(state)

dx, dy = self.directions[direction]

nx, ny = x + dx, y + dy

if 0 <= nx < self.size and 0 <= ny < self.size:

new\_state = deepcopy(state)

new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]

return new\_state

return None

def state\_to\_tuple(self, state):

return tuple(tuple(row) for row in state)

def manhattan(self, state):

total = 0

for i in range(self.size):

for j in range(self.size):

val = state[i][j]

if val != 0:

goal\_x = (val - 1) // self.size

goal\_y = (val - 1) % self.size

total += abs(goal\_x - i) + abs(goal\_y - j)

return total

def heuristic(self, node):

return node.depth + self.manhattan(node.state)

def expand\_node(self, node, visited):

children = []

for move in self.directions.keys():

new\_state = self.move\_tile(node.state, move)

if new\_state and self.state\_to\_tuple(new\_state) not in visited:

child = Node(new\_state, node, move, node.depth + 1, 0)

children.append(child)

return children

def solve(self):

logging.info("Starting puzzle solving")

root = Node(self.start, None, None, 0, 0)

root.cost = self.heuristic(root)

frontier = PriorityQueue()

frontier.put(root)

visited = set()

visited.add(self.state\_to\_tuple(self.start))

while not frontier.empty():

current = frontier.get()

logging.info(f"Current node depth: {current.depth}, cost: {current.cost}")

if current.state == self.goal:

logging.info("Puzzle solved")

path = []

while current.parent:

path.append(current.move)

current = current.parent

path.reverse()

self.moves = path

return path

for child in self.expand\_node(current, visited):

child.cost = self.heuristic(child)

frontier.put(child)

visited.add(self.state\_to\_tuple(child.state))

logging.info("No solution found")

return None

def main():

print("N-PUZZLE SOLVER USING A\* SEARCH\n")

size = int(input("Enter puzzle size (e.g., 3 for 3x3): "))

print("Enter the start state row by row, use 0 for blank:")

start = [list(map(int, input().split())) for \_ in range(size)]

print("Enter the goal state row by row:")

goal = [list(map(int, input().split())) for \_ in range(size)]

puzzle = NPuzzle(start, goal)

solution = puzzle.solve()

if solution:

print("\nPuzzle solved in", len(solution), "moves:")

for i, move in enumerate(solution):

print(f"{i+1}. {move}")

else:

print("No solution found.")

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

main()\_(self, other):

return self.cost < other.cost

class NPuzzle:

def \_\_init\_\_(self, start, goal):

self.start = start

self.goal = goal

self.size = len(start)

self.moves = []

self.directions = {

'UP': (-1, 0),

'DOWN': (1, 0),

'LEFT': (0, -1),

'RIGHT': (0, 1)

}

def find\_blank(self, state):

for i in range(self.size):

for j in range(self.size):

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

return i, j

def move\_tile(self, state, direction):

x, y = self.find\_blank(state)

dx, dy = self.directions[direction]

nx, ny = x + dx, y + dy

if 0 <= nx < self.size and 0 <= ny < self.size:

new\_state = deepcopy(state)

new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]

return new\_state

return None

def state\_to\_tuple(self, state):

return tuple(tuple(row) for row in state)

def manhattan(self, state):

total = 0

for i in range(self.size):

for j in range(self.size):

val = state[i][j]

if val != 0:

goal\_x = (val - 1) // self.size

goal\_y = (val - 1) % self.size

total += abs(goal\_x - i) + abs(goal\_y - j)

return total

def heuristic(self, node):

return node.depth + self.manhattan(node.state)

def expand\_node(self, node, visited):

children = []

for move in self.directions.keys():

new\_state = self.move\_tile(node.state, move)

if new\_state and self.state\_to\_tuple(new\_state) not in visited:

child = Node(new\_state, node, move, node.depth + 1, 0)

children.append(child)

return children

def solve(self):

root = Node(self.start, None, None, 0, 0)

root.cost = self.heuristic(root)

frontier = PriorityQueue()

frontier.put(root)

visited = set()

visited.add(self.state\_to\_tuple(self.start))

while not frontier.empty():

current = frontier.get()

if current.state == self.goal:

path = []

while current.parent:

path.append(current.move)

current = current.parent

path.reverse()

self.moves = path

return path

for child in self.expand\_node(current, visited):

child.cost = self.heuristic(child)

frontier.put(child)

visited.add(self.state\_to\_tuple(child.state))

return None

def main():

print("N-PUZZLE SOLVER USING A\* SEARCH\n")

size = int(input("Enter puzzle size (e.g., 3 for 3x3): "))

print("Enter the start state row by row, use 0 for blank:")

start = [list(map(int, input().split())) for \_ in range(size)]

print("Enter the goal state row by row:")

goal = [list(map(int, input().split())) for \_ in range(size)]

puzzle = NPuzzle(start, goal)

solution = puzzle.solve()

if solution:

print("\nPuzzle solved in", len(solution), "moves:")

for i, move in enumerate(solution):

print(f"{i+1}. {move}")

else:

print("No solution found.")

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

main()

input:

Enter the start state row by row, use 0 for blank:

1 2 3

4 0 6

7 5 8

Enter the goal state row by row:

1 2 3

4 5 6

7 8 0

Puzzle solved in 2 moves:

1. DOWN

2. RIGHT

or

import heapq

import copy

# Moves: Right, Down, Left, Up

moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]

def manhattan(state, goal\_state, n):

dist = 0

for i in range(n):

for j in range(n):

val = state[i][j]

if val != 0:

for x in range(n):

for y in range(n):

if goal\_state[x][y] == val:

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

return dist

def get\_neighbors(state, n):

neighbors = []

for i in range(n):

for j in range(n):

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

x, y = i, j

break

for dx, dy in moves:

nx, ny = x + dx, y + dy

if 0 <= nx < n and 0 <= ny < n:

new\_state = copy.deepcopy(state)

new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]

neighbors.append(new\_state)

return neighbors

def print\_state(state):

for row in state:

print(" ".join(map(str, row)))

print()

def a\_star\_search(start, goal\_state, n):

visited = set()

pq = []

heapq.heappush(pq, (manhattan(start, goal\_state, n), 0, start)) # (f, g, state)

steps = 0

while pq:

f, g, state = heapq.heappop(pq)

state\_tuple = tuple(tuple(row) for row in state)

if state\_tuple in visited:

continue

visited.add(state\_tuple)

print(f"Step {steps}, Cost (f = g+h) = {f}")

print\_state(state)

steps += 1

if state == goal\_state:

print("Goal Reached!")

return

for neighbor in get\_neighbors(state, n):

neighbor\_tuple = tuple(tuple(row) for row in neighbor)

if neighbor\_tuple not in visited:

g\_new = g + 1

h\_new = manhattan(neighbor, goal\_state, n)

heapq.heappush(pq, (g\_new + h\_new, g\_new, neighbor))

print("No solution found.")

def input\_state(name, n):

print(f"Enter the {name} state row by row (space-separated, use 0 for blank):")

state = []

used = set()

for i in range(n):

while True:

try:

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

if len(row) != n or any(x < 0 or x >= n\*n or x in used for x in row):

raise ValueError

used.update(row)

state.append(row)

break

except:

print(f"Invalid input. Enter {n} distinct numbers from 0 to {n\*n - 1}.")

return state

# ----------- MAIN -----------

def main():

n = int(input("Enter the value of N for N-Puzzle (e.g., 3 for 8-puzzle, 4 for 15-puzzle): "))

goal\_state = [[(i \* n + j + 1) % (n \* n) for j in range(n)] for i in range(n)]

initial\_state = input\_state("initial", n)

print("\nGoal State:")

print\_state(goal\_state)

a\_star\_search(initial\_state, goal\_state, n)

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

main()

5b)robot navigation:

import heapq

import math

class Node:

def \_init\_(self, x, y, walkable=True):

self.x = x

self.y = y

self.walkable = walkable

self.g\_score = float('inf')

self.h\_score = 0

self.f\_score = float('inf')

self.parent = None

def \_lt\_(self, other):

return self.f\_score < other.f\_score

def \_eq\_(self, other):

return isinstance(other, Node) and self.x == other.x and self.y == other.y

def \_hash\_(self):

return hash((self.x, self.y))

#stores nodes in sets

class Grid:

def \_init\_(self, width, height, grid\_data):

self.width = width

self.height = height

self.nodes = {}

for y in range(height):

for x in range(width):

walkable = grid\_data[y][x] == '0'

self.nodes[(x, y)] = Node(x, y, walkable)

#creates width\*height grid of nodes 0=walkable,1= obstacle

def get\_node(self, x, y):

return self.nodes.get((x, y))

#returns node at position(x,y)

def get\_neighbors(self, node):

neighbors = []

directions = [(0, 1), (1, 0), (0, -1), (-1, 0),

(1, 1), (1, -1), (-1, -1), (-1, 1)]

for dx, dy in directions:

x, y = node.x + dx, node.y + dy

if 0 <= x < self.width and 0 <= y < self.height:

neighbor = self.get\_node(x, y)

if neighbor and neighbor.walkable:

# Prevent corner cutting

if abs(dx) == 1 and abs(dy) == 1:

if not (self.get\_node(node.x + dx, node.y).walkable and

self.get\_node(node.x, node.y + dy).walkable):

continue

neighbors.append(neighbor)

return neighbors

def euclidean\_distance(a, b):

return math.sqrt((b.x - a.x) \*\* 2 + (b.y - a.y) \*\* 2)

def reconstruct\_path(node):

path = []

current = node

while current:

path.append((current.x, current.y))

current = current.parent

return path[::-1]

def a\_star(grid, start\_pos, goal\_pos):

start\_node = grid.get\_node(\*start\_pos)

goal\_node = grid.get\_node(\*goal\_pos)

if not start\_node or not goal\_node or not start\_node.walkable or not goal\_node.walkable:

return []

start\_node.g\_score = 0

start\_node.h\_score = euclidean\_distance(start\_node, goal\_node)

start\_node.f\_score = start\_node.h\_score

open\_set = []

heapq.heappush(open\_set, start\_node)

open\_set\_hash = {start\_node}

closed\_set = set()

while open\_set:

current = heapq.heappop(open\_set)

open\_set\_hash.remove(current)

if current == goal\_node:

return reconstruct\_path(current)

closed\_set.add(current)

for neighbor in grid.get\_neighbors(current):

if neighbor in closed\_set:

continue

is\_diagonal = abs(neighbor.x - current.x) == 1 and abs(neighbor.y - current.y) == 1

movement\_cost = 1.4 if is\_diagonal else 1.0

tentative\_g\_score = current.g\_score + movement\_cost

if neighbor not in open\_set\_hash or tentative\_g\_score < neighbor.g\_score:

neighbor.parent = current

neighbor.g\_score = tentative\_g\_score

neighbor.h\_score = euclidean\_distance(neighbor, goal\_node)

neighbor.f\_score = neighbor.g\_score + neighbor.h\_score

if neighbor not in open\_set\_hash:

heapq.heappush(open\_set, neighbor)

open\_set\_hash.add(neighbor)

return []

def main():

print("A\* Robot Navigation\n")

width = int(input("Enter grid width: "))

height = int(input("Enter grid height: "))

print("\nEnter your grid data row by row")

print("Use '0' for empty spaces and '1' for obstacles")

grid\_data = []

for i in range(height):

while True:

row = input(f"Row {i}: ")

if len(row) == width and all(c in "01" for c in row):

grid\_data.append(row)

break

else:

print(f"Error: Row must be exactly {width} characters long and contain only 0s and 1s.")

grid = Grid(width, height, grid\_data)

while True:

try:

start\_x, start\_y = map(int, input("\nStart position (x,y): ").split(','))

if 0 <= start\_x < width and 0 <= start\_y < height:

break

else:

print(f"Error: Coordinates must be within grid (0-{width-1},0-{height-1}).")

except ValueError:

print("Error: Enter coordinates as 'x,y'.")

while True:

try:

goal\_x, goal\_y = map(int, input("Goal position (x,y): ").split(','))

if 0 <= goal\_x < width and 0 <= goal\_y < height:

break

else:

print(f"Error: Coordinates must be within grid (0-{width-1},0-{height-1}).")

except ValueError:

print("Error: Enter coordinates as 'x,y'.")

path = a\_star(grid, (start\_x, start\_y), (goal\_x, goal\_y))

if path:

print(f"\nPath found with {len(path)-1} steps:")

for i, (x, y) in enumerate(path):

print(f"Step {i}: ({x},{y})")

else:

print("No path found.")

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

main()

input:

A\* Robot Navigation

Enter grid width: 5

Enter grid height: 5

Enter your grid data row by row

Use '0' for empty spaces and '1' for obstacles

Row 0: 00000

Row 1: 00100

Row 2: 00100

Row 3: 00000

Row 4: 00000

Start position (x,y): 0,0

Goal position (x,y): 4,4

Path found with 6 steps:

Step 0: (0,0)

Step 1: (1,1)

Step 2: (1,2)

Step 3: (1,3)

Step 4: (2,4)

Step 5: (3,4)

Step 6: (4,4)

5c)cities:

import heapq

def a\_star\_search(graph, start, goal, heuristic):

open\_list = []

heapq.heappush(open\_list, (0 + heuristic[start], 0, start, [start]))

visited = set()

print("\nStep-by-step A\* search:")

print(f"{'Current':<10}{'g(n)':<10}{'h(n)':<10}{'f(n)':<10}{'Path'}")

while open\_list:

est\_total\_cost, cost\_so\_far, current, path = heapq.heappop(open\_list)

if current in visited:

continue

visited.add(current)

g = cost\_so\_far

h = heuristic[current]

f = g + h

print(f"{current:<10}{g:<10}{h:<10}{f:<10}{' -> '.join(path)}")

if current == goal:

print("\nReached goal.")

return path, cost\_so\_far

for neighbor, distance in graph[current].items():

if neighbor not in visited:

total\_cost = cost\_so\_far + distance

est\_cost = total\_cost + heuristic[neighbor]

heapq.heappush(open\_list, (est\_cost, total\_cost, neighbor, path + [neighbor]))

return None, float('inf')

def take\_input():

graph = {}

heuristic = {}

n = int(input("Enter number of cities: "))

print("Enter city names:")

cities = [input(f"City {i + 1}: ") for i in range(n)]

print("\nEnter distances between cities (format: City1 City2 Distance), type 'done' to stop:")

while True:

entry = input()

if entry.lower() == 'done':

break

city1, city2, dist = entry.split()

dist = int(dist)

if city1 not in graph:

graph[city1] = {}

if city2 not in graph:

graph[city2] = {}

graph[city1][city2] = dist

graph[city2][city1] = dist # assuming undirected graph

print("\nEnter heuristic values (Estimated distance to goal):")

for city in cities:

heuristic[city] = int(input(f"Heuristic for {city}: "))

start = input("\nEnter source city: ")

goal = input("Enter destination city: ")

return graph, heuristic, start, goal

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

graph, heuristic, start, goal = take\_input()

path, cost = a\_star\_search(graph, start, goal, heuristic)

if path:

print("\nShortest Path:", ' -> '.join(path))

print("Total Cost:", cost)

else:

print("No path found between the given cities.")

input:

PS C:\Users\Anagh\OneDrive\Desktop\ss> python code5c.py

Enter number of cities: 4

Enter city names:

City 1: A

City 2: B

City 3: C

City 4: D

Enter distances between cities (format: City1 City2 Distance), type 'done' to stop:

A B 1

B C 3

A D 4

C D 2

done

Enter heuristic values (Estimated distance to goal):

Heuristic for A: 4

Heuristic for B: 2

Heuristic for C: 1

Heuristic for D: 0

Enter source city: A

Enter destination city: D

Step-by-step A\* search:

Current g(n) h(n) f(n) Path

A 0 4 4 A

B 1 2 3 A -> B

D 4 0 4 A -> D

Reached goal.

Shortest Path: A -> D

Total Cost: 4

6)constraint satisfaction

6a) cryptarithmetic:

from itertools import permutations

import re

def extract\_unique\_letters(expression):

return sorted(set(re.sub(r'[^A-Z]', '', expression)))

def expression\_to\_number(expr, mapping):

return int(''.join(str(mapping[c]) for c in expr))

def is\_valid\_mapping(mapping, terms, result):

# No term can start with a zero

for word in terms + [result]:

if mapping[word[0]] == 0:

return False

total = sum(expression\_to\_number(term, mapping) for term in terms)

return total == expression\_to\_number(result, mapping)

def solve\_cryptarithmetic():

expr = input("Enter expression (e.g., SEND + MORE = MONEY): ").replace(" ", "").upper()

# Parse expression

try:

left, right = expr.split('=')

terms = left.split('+')

result = right

except ValueError:

print("Invalid format. Use format: SEND + MORE = MONEY")

return

letters = extract\_unique\_letters(expr)

if len(letters) > 10:

print("Too many unique letters (max 10 allowed).")

return

for perm in permutations(range(10), len(letters)):

mapping = dict(zip(letters, perm))

if is\_valid\_mapping(mapping, terms, result):

print("\nSolution found!")

for k in sorted(mapping.keys()):

print(f"{k} = {mapping[k]}")

term\_values = [expression\_to\_number(term, mapping) for term in terms]

result\_value = expression\_to\_number(result, mapping)

print(f"\n{' + '.join(map(str, term\_values))} = {result\_value}")

return

print("No solution found.")

solve\_cryptarithmetic()

input: to + go = out

6b)crossword

b. Crossword puzzle

def solve\_crossword(grid, words):

def is\_valid(word, row, col, direction):

if direction == 'across':

if col + len(word) > len(grid[0]):

return False

for i, letter in enumerate(word):

if grid[row][col + i] not in ('\_', letter):

return False

elif direction == 'down':

if row + len(word) > len(grid):

return False

for i, letter in enumerate(word):

if grid[row + i][col] not in ('\_', letter):

return False

return True

def place\_word(word, row, col, direction):

if direction == 'across':

for i, letter in enumerate(word):

grid[row][col + i] = letter

elif direction == 'down':

for i, letter in enumerate(word):

grid[row + i][col] = letter

def remove\_word(word, row, col, direction):

if direction == 'across':

for i, \_ in enumerate(word):

grid[row][col + i] = '\_'

elif direction == 'down':

for i, \_ in enumerate(word):

grid[row + i][col] = '\_'

def solve\_util(words\_to\_place):

if not words\_to\_place:

return True

word = words\_to\_place.pop()

for row in range(len(grid)):

for col in range(len(grid[0])):

for direction in ('across', 'down'):

if is\_valid(word, row, col, direction):

place\_word(word, row, col, direction)

if solve\_util(words\_to\_place):

return True

remove\_word(word, row, col, direction)

words\_to\_place.append(word)

return False

words\_to\_place = list(words)

return solve\_util(words\_to\_place)

def print\_grid(grid):

for row in grid:

print(' '.join(row))

# Input

rows = int(input("Enter the number of rows in the crossword grid: "))

cols = int(input("Enter the number of columns in the crossword grid: "))

grid = [['\_' for \_ in range(cols)] for \_ in range(rows)]

num\_words = int(input("Enter the number of words: "))

words = []

for i in range(num\_words):

word = input(f"Enter word {i + 1}: ").upper()

words.append(word)

# Solve the crossword puzzle

solution\_found = solve\_crossword(grid, words)

if solution\_found:

print("Solution found:")

print\_grid(grid)

else:

print("No solution found.")

Enter the number of rows in the crossword grid: 5

Enter the number of columns in the crossword grid: 5

Enter the number of words: 3

Enter word 1: HELLO

Enter word 2: WORLD

Enter word 3: HOW

6c)mapcoloring:

def get\_user\_input():

n = int(input("Enter number of nodes: "))

nodes = []

adjacency = {}

print("Enter node names (e.g. A, B, C):")

for \_ in range(n):

node = input("Node: ").strip().upper()

nodes.append(node)

print("Enter adjacency list (space-separated neighbors):")

for node in nodes:

neighbors = input(f"Neighbors of {node}: ").strip().upper().split()

adjacency[node] = neighbors

print("Enter colors separated by commas (e.g. Red,Green,Blue):")

colors = [color.strip().capitalize() for color in input().split(',')]

return nodes, adjacency, colors

def is\_safe(state, node, color, adjacency):

for neighbor in adjacency[node]:

if neighbor in state and state[neighbor] == color:

return False

return True

def map\_coloring(state, nodes, adjacency, colors):

if not nodes:

print("\nColoring Solution:")

for node in sorted(state):

print(f"{node} => {state[node]}")

return True

node = nodes[0]

for color in colors:

if is\_safe(state, node, color, adjacency):

state[node] = color

if map\_coloring(state, nodes[1:], adjacency, colors):

return True

del state[node]

return False

def main():

nodes, adjacency, colors = get\_user\_input()

if not map\_coloring({}, nodes, adjacency, colors):

print("\nNo valid coloring found.")

main()

7)nlp

7a) pos tagging:

import spacy

# Load the spaCy English model

nlp = spacy.load("en\_core\_web\_sm")

def pos\_tagging(text):

doc = nlp(text)

results = []

for token in doc:

results.append((token.text, token.pos\_, token.tag\_, spacy.explain(token.tag\_)))

return results

# Example usage

text = "The quick brown fox jumps over the lazy dog"

tagged = pos\_tagging(text)

print("Token".ljust(15), "POS".ljust(10), "Tag".ljust(10), "Description")

print("-" \* 50)

for token in tagged:

print(f"{token[0].ljust(15)}{token[1].ljust(10)}{token[2].ljust(10)}{token[3]}")

or userinput m:

import spacy

import warnings

warnings.filterwarnings("ignore", category=UserWarning)

# Load the spaCy English model

nlp = spacy.load("en\_core\_web\_sm")

def pos\_tagging(text):

doc = nlp(text)

results = []

for token in doc:

results.append((token.text, token.pos\_, token.tag\_, spacy.explain(token.tag\_)))

return results

# Example usage

text = input("enter the sentence: ")

tagged = pos\_tagging(text)

print("Token".ljust(15), "POS".ljust(10), "Tag".ljust(10), "Description")

print("-" \* 50)

for token in tagged:

print(f"{token[0].ljust(15)}{token[1].ljust(10)}{token[2].ljust(10)}{token[3]}")

pip install spacy

python -m spacy download en\_core\_web\_md

7b) SIMILARITY CHECKER:

import spacy

# Load pre-trained spaCy model

nlp = spacy.load("en\_core\_web\_md")

def similarity\_score():

s1 = input("Enter first sentence: ")

s2 = input("Enter second sentence: ")

# Process the sentences using spaCy

doc1 = nlp(s1)

doc2 = nlp(s2)

# Calculate the similarity

score = doc1.similarity(doc2)

print(f"\nSemantic Similarity Score: {score:.2f}")

# Call similarity score function

similarity\_score()

pip install spacy

python -m spacy download en\_core\_web\_md

7c) spell checker:

import language\_tool\_python

tool = language\_tool\_python.LanguageTool('en-US')

sentence = input("Enter a sentence with spelling errors: ")

matches = tool.check(sentence)

corrected = language\_tool\_python.utils.correct(sentence, matches)

print("\nCorrected Sentence:")

print(corrected)

8)minimax algorithm tic tac toe:

import math

def print\_board(board):

for row in board:

print(" | ".join(row))

print("-" \* 9)

def is\_winner(board, player):

for row in board:

if all(cell == player for cell in row):

return True

for col in range(3):

if all(board[row][col] == player for row in range(3)):

return True

if all(board[i][i] == player for i in range(3)) or \

all(board[i][2 - i] == player for i in range(3)):

return True

return False

def is\_draw(board):

return all(cell != ' ' for row in board for cell in row)

def get\_empty\_cells(board):

return [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']

def minimax(board, is\_maximizing):

if is\_winner(board, 'O'):

return 1

elif is\_winner(board, 'X'):

return -1

elif is\_draw(board):

return 0

if is\_maximizing:

best\_score = -math.inf

for i, j in get\_empty\_cells(board):

board[i][j] = 'O'

score = minimax(board, False)

board[i][j] = ' '

best\_score = max(score, best\_score)

return best\_score

else:

best\_score = math.inf

for i, j in get\_empty\_cells(board):

board[i][j] = 'X'

score = minimax(board, True)

board[i][j] = ' '

best\_score = min(score, best\_score)

return best\_score

def best\_move(board):

best\_score = -math.inf

move = None

for i, j in get\_empty\_cells(board):

board[i][j] = 'O'

score = minimax(board, False)

board[i][j] = ' '

if score > best\_score:

best\_score = score

move = (i, j)

return move

# Main game loop

board = [[' ' for \_ in range(3)] for \_ in range(3)]

print("Welcome to Tic Tac Toe! You are X, computer is O.")

print\_board(board)

while True:

# User move

row = int(input("Enter row (0-2): "))

col = int(input("Enter column (0-2): "))

if board[row][col] != ' ':

print("Invalid move! Try again.")

continue

board[row][col] = 'X'

print\_board(board)

if is\_winner(board, 'X'):

print("You win!")

break

if is\_draw(board):

print("It's a draw!")

break

# Computer move

i, j = best\_move(board)

board[i][j] = 'O'

print("\nComputer move:")

print\_board(board)

if is\_winner(board, 'O'):

print("Computer wins!")

break

if is\_draw(board):

print("It's a draw!")

break