

**BANSILAL RAMNATH AGARWAL CHARITABLE TRUST’S**

**VISHWAKARMA INSTITUTE OF INFORMATION.TECHNOLOGY.**

## Sr.No.2/3/4,Kondhwa(BK),Pune-48

LAB MANUAL

Artificial Intelligence Lab Manual

FOR

Subject Code: **CAUA31201**

Submitted by:

Name: Sai Sanjay Nangare

RollNo: 381049

PRN: 22210801

# ASSIGNMENT NO-1

**1. DFS and BFS for the 8-Puzzle Problem**

**Problem Statement:**

Implement DFS and BFS to solve the 8-puzzle problem, where the objective is to rearrange tiles to match the goal state using these search strategies.

**Objective:**

• Compare Depth-First Search (DFS) and Breadth-First Search (BFS) efficiency in a state-space search problem.

• Measure time complexity, space complexity, and performance.

**Theory:**

• DFS: Explores as deep as possible along one branch before backtracking.

• BFS: Explores all nodes at a given depth before proceeding to the next level.

• 8-puzzle: A sliding puzzle with 8 tiles and one empty space, where you rearrange tiles to match a goal state.

**Advantages:**

1. **BFS:**
   * Guarantees the shortest solution (if one exists).
   * Suitable for finding solutions in unweighted graphs.
2. **DFS:**
   * Consumes less memory than BFS.
   * May find solutions faster if the goal node is deep in the search tree.

**Disadvantages:**

1. **BFS:**
   * High memory consumption.
   * Can be slow for large search spaces.
2. **DFS:**
   * May get stuck in deep or infinite branches.
   * Does not guarantee the shortest path.

**Applications:**

* **BFS:** Social networks (shortest connection path), GPS navigation, web crawling.
* **DFS:** Maze solving, finding strongly connected components in a graph, scheduling problems.

**Library Used:**

* numpy for reshaping and printing the puzzle in a 3x3 grid format.

**Input:**

import numpy as np

def bfs(src, target):

    queue = [src]    # Queue to explore nodes

    visited = []     # Visited nodes

    while queue:

        state = queue.pop(0)

        visited.append(state)

        print\_puzzle(state)

        if state == target:

            print("\nSuccess! Goal state achieved!")

            return

        # Get possible moves and append to queue if not visited

        for move in possible\_moves(state, visited):

            if move not in visited and move not in queue:

                queue.append(move)

def possible\_moves(state, visited):

    b = state.index(0)  # Position of empty tile (0)

    directions = []

    if b > 2: directions.append('u')  # Move up

    if b < 6: directions.append('d')  # Move down

    if b % 3 > 0: directions.append('l')  # Move left

    if b % 3 < 2: directions.append('r')  # Move right

    return [gen(state, dir, b) for dir in directions if gen(state, dir, b) not in visited]

def gen(state, direction, b):

    new\_state = state.copy()

    swaps = {'u': -3, 'd': 3, 'l': -1, 'r': 1}

    new\_b = b + swaps[direction]

    new\_state[b], new\_state[new\_b] = new\_state[new\_b], new\_state[b]

    return new\_state

def print\_puzzle(state):

    print(np.array(state).reshape(3, 3), "\n")

# Initial setup

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

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

print("Initial state:")

print\_puzzle(src)

print("Goal state:")

print\_puzzle(target)

print("BFS solution:")

bfs(src, target)

**Output:**

Initial state:

[[1 2 3]

[4 5 6]

[0 7 8]]

Goal state:

[[1 2 3]

[4 5 6]

[7 8 0]]

BFS solution:

[[1 2 3]

[4 5 6]

[0 7 8]]

[[1 2 3]

[0 5 6]

[4 7 8]]

[[1 2 3]

[4 5 6]

[7 0 8]]

[[0 2 3]

[1 5 6]

[4 7 8]]

[[1 2 3]

[5 0 6]

[4 7 8]]

[[1 2 3]

[4 0 6]

[7 5 8]]

[[1 2 3]

[4 5 6]

[7 8 0]]

Success! Goal state achieved!

# ASSIGNMENT NO-2

**2. Constraint Satisfaction Problem (CSP)**

**Problem Statement:**

Implement a CSP solver (e.g., Sudoku) that finds solutions satisfying all given constraints.

**Objective:**

• Use CSP techniques like backtracking and arc consistency to solve puzzles.

• Minimize search by pruning using constraints.

**Theory:**

A CSP is defined by variables, domains, and constraints. The goal is to assign values to variables such that all constraints are met.

**Advantages:**

* Guarantees an optimal solution.
* Can solve complex constraint-based problems effectively.

**Disadvantages:**

* Computationally expensive for large CSPs.
* Backtracking may be time-consuming without proper pruning.

**Applications:**

* Solving scheduling problems.
* Cryptarithm puzzles.
* Sudoku and crossword puzzle solvers.

**Libraries Used:**

* **Java** standard libraries (java.util.\*).

**Input:**

import java.util.\*;

public class Assignment2 {

    static int im = 3, ic = 3, fm = 0, fc = 0, flag = 0, select = 0;

    static void display(char bpass1, char bpass2) {

        System.out.println("\n\n\n");

        for (int i = 0; i < fm; i++) {

            System.out.print(" M ");

        }

        for (int i = 0; i < fc; i++) {

            System.out.print(" C ");

        }

        if (flag == 0)

            System.out.print("\_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_B0(" + bpass1 + "," + bpass2 + ")AT  ");

        else

            System.out.print("     BO(" + bpass1 + "," + bpass2 + ")AT\_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_  ");

        for (int i = 0; i < im; i++) {

            System.out.print(" M ");

        }

        for (int i = 0; i < ic; i++) {

            System.out.print(" C ");

        }

    }

    static boolean win() {

        return !(fc == 3 && fm == 3);

    }

    static void solution() {

        while (win()) {

            if (flag == 0) {

                switch (select) {

                    case 1:

                        display('C', ' ');

                        ic++;

                        break;

                    case 2:

                        display('C', 'M');

                        ic++;

                        im++;

                        break;

                }

                if (((im - 2) >= ic && (fm + 2) >= fc) || (im - 2) == 0) {

                    im = im - 2;

                    select = 1;

                    display('M', 'M');

                    flag = 1;

                } else if ((ic - 2) < im && (fm == 0 || (fc + 2) <= fm) || im == 0) {

                    ic = ic - 2;

                    select = 2;

                    display('C', 'C');

                    flag = 1;

                } else if ((ic--) <= (im--) && (fm++) >= (fc++)) {

                    ic = ic - 1;

                    im = im - 1;

                    select = 3;

                    display('M', 'C');

                    flag = 1;

                }

            } else {

                switch (select) {

                    case 1:

                        display('M', 'M');

                        fm = fm + 2;

                        break;

                    case 2:

                        display('C', 'C');

                        fc = fc + 2;

                        break;

                    case 3:

                        display('M', 'C');

                        fc = fc + 1;

                        fm = fm + 1;

                        break;

                }

                if (win()) {

                    if (((fc > 1 && fm == 0) || im == 0)) {

                        fc--;

                        select = 1;

                        display('C', ' ');

                        flag = 0;

                    } else if ((ic + 2) > im) {

                        fc--;

                        fm--;

                        select = 2;

                        display('C', 'M');

                        flag = 0;

                    }

                }

            }

        }

    }

    public static void main(String[] args) {

        System.out.println("MISSIONARIES AND CANNIBALS");

        display(' ', ' ');

        solution();

        display(' ', ' ');

        System.out.println("\n\n");

    }

}

**Output:**

\_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_B0( , )AT M M M C C C

\_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_B0(C,C)AT M M M C

BO(C,C)AT\_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_ M M M C

C BO(C, )AT\_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_ M M M C

C \_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_B0(C, )AT M M M C

C \_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_B0(C,C)AT M M M

C BO(C,C)AT\_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_ M M M

C C BO(C, )AT\_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_ M M M

C C \_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_B0(C, )AT M M M

C C \_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_B0(M,M)AT M C

C C BO(M,M)AT\_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_ M C

M C BO(C,M)AT\_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_ M C

M C \_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_B0(C,M)AT M C

M C \_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_B0(M,M)AT C C

M C BO(M,M)AT\_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_ C C

M M M BO(C, )AT\_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_ C C

M M M \_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_B0(C, )AT C C

M M M \_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_B0(C,C)AT C

M M M BO(C,C)AT\_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_ C

M M M C BO(C, )AT\_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_ C

M M M C \_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_B0(C, )AT C

M M M C \_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_B0(C,C)AT

M M M C BO(C,C)AT\_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_

M M M C C C BO( , )AT\_\_\_\_\_\_\_\_\_\_WATER\_\_\_\_\_\_\_\_\_\_\_

PS C:\Users\saina\OneDrive\Documents\Desktop\Data\AI Lab Assignment>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# ASSIGNMENT NO-3

**3. Parsing Family Tree Using Knowledge-Base**

**Problem Statement:**

Create a family tree parser that uses a knowledge base to infer relationships (e.g., "Who is X's grandmother?").

**Objective:**

• Parse relationships efficiently using first-order logic and a knowledge-based approach.

• Demonstrate the use of predicates and rules for inference.

**Theory:**

A knowledge base consists of facts and rules. Using parsing algorithms and inference engines, the system can derive conclusions from given inputs.

**Advantages:**

* **Scalability:** New facts and rules can be added without modifying the whole system.
* **Logical Inference:** The system can infer new knowledge from existing facts.
* **Automation:** Useful in automating complex queries with predefined rules.

**Disadvantages:**

* **Computational Complexity:** Can be slow if the dataset and rules grow large.
* **Ambiguity Issues:** Requires well-defined predicates to avoid misinterpretation.
* **Data Inconsistency:** Conflicting facts can lead to incorrect results without proper handling.

**Applications:**

* **Genealogical Research:** Automatically generate family trees or relationships.
* **Knowledge Graphs:** Applications like **Google Knowledge Graph** use similar logic to infer relationships between entities.
* **AI-based Chatbots:** Handling queries related to relationships (e.g., family assistants).
* **Healthcare Systems:** Storing family medical history for better diagnoses.

**Libraries Used:**

* **Python Libraries:**
  + **prologpy** or **pyDatalog** for first-order logic and knowledge-based reasoning.
  + **networkx** for creating graph-based representations of family trees.
  + **NLP libraries** (like spaCy) for parsing user queries.

**Input:**

# Sample knowledge base: family members and their relationships

family\_tree = {

    "Raj": {"parent": ["Anil", "Sunita"], "spouse": "Priya", "children": ["Aarav", "Ananya"]},

    "Anil": {"spouse": "Sunita", "children": ["Raj"]},

    "Sunita": {"spouse": "Anil", "children": ["Raj"]},

    "Priya": {"spouse": "Raj", "children": ["Aarav", "Ananya"]},

    "Aarav": {"parent": ["Raj", "Priya"]},

    "Ananya": {"parent": ["Raj", "Priya"]},

}

# Function to find parents of a person

def find\_parents(person):

    if person in family\_tree and "parent" in family\_tree[person]:

        return family\_tree[person]["parent"]

    return "No parents found"

# Function to find children of a person

def find\_children(person):

    if person in family\_tree and "children" in family\_tree[person]:

        return family\_tree[person]["children"]

    return "No children found"

# Function to find spouse of a person

def find\_spouse(person):

    if person in family\_tree and "spouse" in family\_tree[person]:

        return family\_tree[person]["spouse"]

    return "No spouse found"

# Function to find siblings of a person

def find\_siblings(person):

    parents = find\_parents(person)

    if isinstance(parents, list):

        siblings = {sibling for parent in parents for sibling in find\_children(parent)}

        siblings.discard(person)

        return list(siblings)

    return "No siblings found"

# Function to find grandparents of a person

def find\_grandparents(person):

    parents = find\_parents(person)

    if isinstance(parents, list):

        grandparents = {grandparent for parent in parents for grandparent in find\_parents(parent)}

        return list(grandparents)

    return "No grandparents found"

# Example queries using the parser functions

def run\_queries(person):

    print(f"Parents of {person}: {find\_parents(person)}")

    print(f"Children of {person}: {find\_children(person)}")

    print(f"Spouse of {person}: {find\_spouse(person)}")

    print(f"Siblings of {person}: {find\_siblings(person)}")

    print(f"Grandparents of {person}: {find\_grandparents(person)}")

# Test the code with a person

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

    person = "Aarav"

    run\_queries(person)

**Output:**

Parents of Aarav: ['Raj', 'Priya']

Children of Aarav: No children found

Spouse of Aarav: No spouse found

Siblings of Aarav: ['Ananya']

Grandparents of Aarav: ['n', 'N', 's', 'f', 't', 'Sunita', 'd', 'o', 'a', ' ', 'p', 'e', 'u', 'r', 'Anil']

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# ASSIGNMENT NO-4

**4. A Algorithm for an Application\***

**Problem Statement:**

Implement A\* to solve a real-world problem (e.g., finding the shortest path in a maze or route planning).

**Objective:**

• Use heuristics for optimal pathfinding.

• Compare performance with other search strategies.

**Theory:**

A\* (A-star) is a **pathfinding and graph traversal algorithm** widely used to find the shortest path between two points. It uses both **cost of the path so far (g(n))** and **a heuristic estimate (h(n))** to predict the remaining cost to reach the goal. The total cost for any node is calculated as:

f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n)

Where:

* **g(n):** The actual cost from the start node to the current node.
* **h(n):** The heuristic estimate of the cost from the current node to the goal (e.g., Euclidean or Manhattan distance).

The A\* algorithm is **greedy and optimal**. It prioritizes nodes based on the lowest total estimated cost, ensuring it finds the shortest path efficiently in most cases.

**Advantages:**

* **Optimal and Complete:** If the heuristic is admissible, A\* guarantees finding the shortest path.
* **Flexible:** Works with many types of heuristics (e.g., Manhattan or Euclidean distance).
* **Widely Applicable:** Can be applied to a variety of domains like games, robotics, and navigation.

**Disadvantages:**

* **Memory-Intensive:** A\* can consume significant memory for large graphs or grids.
* **Performance Degrades with Poor Heuristics:** If the heuristic is not well-tuned, it may explore unnecessary paths.
* **Slower for Large Graphs:** Compared to simpler algorithms like BFS in certain situations.

**Applications:**

* **Route Planning:** GPS systems use A\* to find the shortest path between locations.
* **Game AI:** Helps non-player characters (NPCs) navigate complex maps efficiently.
* **Robotics:** Used in robot path planning for obstacle avoidance.
* **Logistics and Delivery Services:** Determines optimal delivery routes in real-time.

**Libraries Used:**

* **Python Libraries:**
  + **networkx**: For graph-based problems.
  + **numpy**: For handling matrices or grids (e.g., representing a maze).
  + **heapq**: For implementing priority queues efficiently.
  + **pygame**: For visualizing the algorithm on maps or grids.

**Input:**

*# Python program for A\* Search Algorithm*

import math

import heapq

*# Define the Cell class*

class Cell:

def \_\_init\_\_(self):

*# Parent cell's row index*

self.parent\_i = 0

*# Parent cell's column index*

self.parent\_j = 0

*# Total cost of the cell (g + h)*

self.f = float('inf')

*# Cost from start to this cell*

self.g = float('inf')

*# Heuristic cost from this cell to destination*

self.h = 0

*# Define the size of the grid*

ROW = 9

COL = 10

*# Check if a cell is valid (within the grid)*

def is\_valid(row, col):

return (row >= 0) and (row < ROW) and (col >= 0) and (col < COL)

*# Check if a cell is unblocked*

def is\_unblocked(grid, row, col):

return grid[row][col] == 1

*# Check if a cell is the destination*

def is\_destination(row, col, dest):

return row == dest[0] and col == dest[1]

*# Calculate the heuristic value of a cell (Euclidean distance to destination)*

def calculate\_h\_value(row, col, dest):

return ((row - dest[0]) \*\* 2 + (col - dest[1]) \*\* 2) \*\* 0.5

*# Trace the path from source to destination*

def trace\_path(cell\_details, dest):

print("The Path is ")

path = []

row = dest[0]

col = dest[1]

*# Trace the path from destination to source using parent cells*

while not (cell\_details[row][col].parent\_i == row and cell\_details[row][col].parent\_j == col):

path.append((row, col))

temp\_row = cell\_details[row][col].parent\_i

temp\_col = cell\_details[row][col].parent\_j

row = temp\_row

col = temp\_col

*# Add the source cell to the path*

path.append((row, col))

*# Reverse the path to get the path from source to destination*

path.reverse()

*# Print the path*

for i in path:

print("->", i, end=" ")

print()

*# Implement the A\* search algorithm*

def a\_star\_search(grid, src, dest):

*# Check if the source and destination are valid*

if not is\_valid(src[0], src[1]) or not is\_valid(dest[0], dest[1]):

print("Source or destination is invalid")

return

*# Check if the source and destination are unblocked*

if not is\_unblocked(grid, src[0], src[1]) or not is\_unblocked(grid, dest[0], dest[1]):

print("Source or the destination is blocked")

return

*# Check if we are already at the destination*

if is\_destination(src[0], src[1], dest):

print("We are already at the destination")

return

*# Initialize the closed list (visited cells)*

closed\_list = [[False for \_ in range(COL)] for \_ in range(ROW)]

*# Initialize the details of each cell*

cell\_details = [[Cell() for \_ in range(COL)] for \_ in range(ROW)]

*# Initialize the start cell details*

i = src[0]

j = src[1]

cell\_details[i][j].f = 0

cell\_details[i][j].g = 0

cell\_details[i][j].h = 0

cell\_details[i][j].parent\_i = i

cell\_details[i][j].parent\_j = j

*# Initialize the open list (cells to be visited) with the start cell*

open\_list = []

heapq.heappush(open\_list, (0.0, i, j))

*# Initialize the flag for whether destination is found*

found\_dest = False

*# Main loop of A\* search algorithm*

while len(open\_list) > 0:

*# Pop the cell with the smallest f value from the open list*

p = heapq.heappop(open\_list)

*# Mark the cell as visited*

i = p[1]

j = p[2]

closed\_list[i][j] = True

*# For each direction, check the successors*

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

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

for dir in directions:

new\_i = i + dir[0]

new\_j = j + dir[1]

*# If the successor is valid, unblocked, and not visited*

if is\_valid(new\_i, new\_j) and is\_unblocked(grid, new\_i, new\_j) and not closed\_list[new\_i][new\_j]:

*# If the successor is the destination*

if is\_destination(new\_i, new\_j, dest):

*# Set the parent of the destination cell*

cell\_details[new\_i][new\_j].parent\_i = i

cell\_details[new\_i][new\_j].parent\_j = j

print("The destination cell is found")

*# Trace and print the path from source to destination*

trace\_path(cell\_details, dest)

found\_dest = True

return

else:

*# Calculate the new f, g, and h values*

g\_new = cell\_details[i][j].g + 1.0

h\_new = calculate\_h\_value(new\_i, new\_j, dest)

f\_new = g\_new + h\_new

*# If the cell is not in the open list or the new f value is smaller*

if cell\_details[new\_i][new\_j].f == float('inf') or cell\_details[new\_i][new\_j].f > f\_new:

*# Add the cell to the open list*

heapq.heappush(open\_list, (f\_new, new\_i, new\_j))

*# Update the cell details*

cell\_details[new\_i][new\_j].f = f\_new

cell\_details[new\_i][new\_j].g = g\_new

cell\_details[new\_i][new\_j].h = h\_new

cell\_details[new\_i][new\_j].parent\_i = i

cell\_details[new\_i][new\_j].parent\_j = j

*# If the destination is not found after visiting all cells*

if not found\_dest:

print("Failed to find the destination cell")

*# Driver Code*

def main():

*# Define the grid (1 for unblocked, 0 for blocked)*

grid = [

[1, 0, 1, 1, 1, 1, 0, 1, 1, 1],

[1, 1, 1, 0, 1, 1, 1, 0, 1, 1],

[1, 1, 1, 0, 1, 1, 0, 1, 0, 1],

[0, 0, 1, 0, 1, 0, 0, 0, 0, 1],

[1, 1, 1, 0, 1, 1, 1, 0, 1, 0],

[1, 0, 1, 1, 1, 1, 0, 1, 0, 0],

[1, 0, 0, 0, 0, 1, 0, 0, 0, 1],

[1, 0, 1, 1, 1, 1, 0, 1, 1, 1],

[1, 1, 1, 0, 0, 0, 1, 0, 0, 1]

]

*# Define the source and destination*

src = [8, 0]

dest = [0, 0]

*# Run the A\* search algorithm*

a\_star\_search(grid, src, dest)

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

main()

**Output:**

The destination cell is found

The Path is

-> (8, 0) -> (7, 0) -> (6, 0) -> (5, 0) -> (4, 1) -> (3, 2) -> (2, 1) -> (1, 0) -> (0, 0)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# ASSIGNMENT NO-5

**5. Minimax Algorithm for Game Playing**

**Problem Statement:**

Implement the Minimax algorithm to create a game-playing AI (e.g., Tic-Tac-Toe).

**Objective:**

• Develop an AI that plays optimally against a human or another AI.

• Use depth-based pruning to enhance performance (Alpha-Beta Pruning).

**Theory**:

The Minimax algorithm evaluates the utility of terminal game states and assumes the opponent will always make the optimal move. Alpha-beta pruning improves efficiency by avoiding unnecessary evaluations.

**Advantages:**

* **Optimal Strategy:** Minimax ensures the best possible move is chosen if both players play perfectly.
* **Simple to Implement:** Works well with smaller games like Tic-Tac-Toe or Checkers.
* **Alpha-Beta Pruning**: Reduces time complexity, making the algorithm scalable for deeper searches.

**Disadvantages:**

* **Exponential Time Complexity:** Minimax becomes computationally expensive for large games like Chess or Go.
* **Memory-Intensive:** It requires keeping track of many game states in deeper trees.
* **Suboptimal with Incomplete Information:** Minimax assumes perfect information and optimal play from both players, which might not align with real-world scenarios.

**Applications:**

* **Tic-Tac-Toe:** Building an unbeatable AI for small board games.
* **Chess Engines:** AI like Stockfish uses variations of Minimax with deep pruning techniques.
* **Connect-4 and Checkers:** Game AIs that challenge human players effectively.
* **Game Development:** Developing competitive AI opponents for board games or interactive puzzles.

**Libraries Used:**

* **Python Libraries:**
  + **pygame**: For creating and visualizing the game interface (e.g., Tic-Tac-Toe).
  + **numpy**: For managing the game board as matrices or grids.
  + **random**: To add randomness to moves when the AI faces multiple equally good options.
  + **time**: To measure the performance improvements with and without Alpha-Beta Pruning.

**Input:**

import numpy as np

*# Constants for players*

PLAYER\_X = 1 *# AI*

PLAYER\_O = -1 *# Human*

EMPTY = 0

*# Function to initialize the board*

def initialize\_board():

return np.zeros((3, 3), dtype=int)

*# Function to check for a winner*

def check\_winner(board):

for row in board:

if abs(sum(row)) == 3:

return np.sign(sum(row))

for col in board.T:

if abs(sum(col)) == 3:

return np.sign(sum(col))

if abs(sum(board.diagonal())) == 3:

return np.sign(sum(board.diagonal()))

if abs(sum(np.fliplr(board).diagonal())) == 3:

return np.sign(sum(np.fliplr(board).diagonal()))

return 0 *# No winner*

*# Function to check if the board is full*

def is\_board\_full(board):

return not np.any(board == EMPTY)

*# Minimax function*

def minimax(board, depth, is\_maximizing):

winner = check\_winner(board)

if winner == PLAYER\_X:

return 10 - depth *# AI wins*

elif winner == PLAYER\_O:

return depth - 10 *# Human wins*

elif is\_board\_full(board):

return 0 *# Draw*

if is\_maximizing:

best\_score = -np.inf

for row in range(3):

for col in range(3):

if board[row, col] == EMPTY:

board[row, col] = PLAYER\_X *# AI move*

score = minimax(board, depth + 1, False)

board[row, col] = EMPTY *# Undo move*

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

return best\_score

else:

best\_score = np.inf

for row in range(3):

for col in range(3):

if board[row, col] == EMPTY:

board[row, col] = PLAYER\_O *# Human move*

score = minimax(board, depth + 1, True)

board[row, col] = EMPTY *# Undo move*

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

return best\_score

def find\_best\_move(board):

best\_score = -np.inf

best\_move = (-1, -1)

for row in range(3):

for col in range(3):

if board[row, col] == EMPTY:

board[row, col] = PLAYER\_X *# AI move*

score = minimax(board, 0, False)

board[row, col] = EMPTY *# Undo move*

if score > best\_score:

best\_score = score

best\_move = (row, col)

return best\_move

def play\_game():

board = initialize\_board()

while True:

print(board)

*# Player's turn (Human)*

row, col = map(int, input("Enter your move (row col): ").split())

if board[row, col] != EMPTY:

print("Invalid move. Try again.")

continue

board[row, col] = PLAYER\_O

if check\_winner(board):

print("You win!")

break

if is\_board\_full(board):

print("It's a draw!")

break

*# AI's turn (Minimax)*

ai\_move = find\_best\_move(board)

board[ai\_move] = PLAYER\_X

if check\_winner(board):

print("AI wins!")

break

if is\_board\_full(board):

print("It's a draw!")

break

play\_game()

**Output:**

[[0 0 0]

[0 0 0]

[0 0 0]]

Enter your move (row col): 0 0

[[-1 0 0]

[ 0 1 0]

[ 0 0 0]]

Enter your move (row col): 1 1

Invalid move. Try again.

[[-1 0 0]

[ 0 1 0]

[ 0 0 0]]

Enter your move (row col): 2 2

[[-1 1 0]

[ 0 1 0]

[ 0 0 -1]]

Enter your move (row col): 2 1

[[-1 1 0]

[ 0 1 0]

[ 1 -1 -1]]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# ASSIGNMENT NO-6

**6. Basic Search Strategies – 8-Queens Problem**

**Problem Statement:**

Implement search algorithms to solve the 8-Queens problem.

**Objective:**

• Demonstrate how basic search strategies (e.g., backtracking, brute-force search) solve the problem.

• Minimize computation using constraint checks.

**Theory:**

The **8-Queens problem** is a **Constraint Satisfaction Problem (CSP)** where the goal is to place 8 queens on an 8x8 chessboard such that no two queens attack each other. A queen can attack any piece in the same **row, column, or diagonal**. This problem is often solved using search strategies like:

1. **Brute-Force Search:**
   * Try every possible combination of queen placements until a valid solution is found. This approach is computationally expensive but guarantees correctness.
2. **Backtracking:**
   * A more efficient strategy where a queen is placed column by column. If a conflict is detected (a queen attacks another), the algorithm **backtracks** and tries the next possibility.

Constraint checks (like checking the row, column, and diagonal conflicts) help **prune invalid paths**, minimizing the number of explored states.

**Advantages:**

* **Guaranteed Solution:** Both brute-force and backtracking will eventually find a solution.
* **Backtracking is Efficient:** It dramatically reduces search space compared to brute force.
* **Easy to Implement:** Basic search strategies are straightforward and do not require advanced heuristics.

**Disadvantages:**

* **Brute Force is Inefficient:** Time complexity grows exponentially with the board size.
* **Backtracking Limitations:** As the board size increases (e.g., n-Queens for larger n), even backtracking can become slow.
* **No Scalability:** Basic search strategies are not suitable for real-time applications or very large search spaces.

**Applications:**

* **Puzzle Solving:** Variants of the 8-Queens problem are used in other puzzles.
* **AI and Constraint Satisfaction Problems:** The principles of backtracking and constraint checks are used in AI applications such as **scheduling, resource allocation**, and **Sudoku solvers**.
* **Algorithm Research:** It serves as a classic problem for teaching **search algorithms** and **optimization techniques**.
* **Game Development:** Elements of the algorithm are useful for **pathfinding** and managing game states.

**Libraries Used:**

* **Python Libraries:**
  + **numpy**: To represent the chessboard as an 8x8 matrix.
  + **itertools**: Useful for generating combinations in brute-force methods.
  + **time**: To measure and compare the performance of different strategies.
  + **matplotlib**: For visualizing the board and solution with queens placed on it.

**Input:**

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

*# Check the column*

for i in range(row):

if board[i] == col:

return False

*# Check the left diagonal*

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

if board[i] == j:

return False

*# Check the right diagonal*

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

if board[i] == j:

return False

return True

def solve\_8\_queens(board, row=0):

if row == len(board):

print\_board(board)

return True *# Stop after the first solution*

for col in range(len(board)):

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

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

if solve\_8\_queens(board, row + 1): *# Recur to the next row*

return True

board[row] = -1 *# Backtrack*

return False

def print\_board(board):

for row in board:

print(' '.join('Q' if col == row else '.' for col in range(len(board))))

print()

*# Initialize the board*

n = 8

board = [-1] \* n

solve\_8\_queens(board)

**Output:**

Q . . . . . . .

. . . . Q . . .

. . . . . . . Q

. . . . . Q . .

. . Q . . . . .

. . . . . . Q .

. Q . . . . . .

. . . Q . . . .

Out[8]:

True

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# ASSIGNMENT NO-7

**7. Forward Chaining Algorithm**

**Problem Statement:**

Implement a forward chaining algorithm to infer facts (e.g., diagnosing diseases from symptoms).

**Objective:**

• Use forward chaining to derive conclusions from known facts.

• Automate reasoning based on IF-THEN rules.

**Theory:**

Forward chaining starts from known facts and applies rules to infer new facts until a goal is reached.

Input:

def forward\_chaining(facts, rules):

inferred = set(facts) *# Set of known facts*

applied\_rules = set() *# Keep track of used rules*

while True:

applied = False

for rule, conclusion in rules:

if rule.issubset(inferred) and conclusion not in inferred:

inferred.add(conclusion)

*# Store the rule as a tuple for tracking*

applied\_rules.add((tuple(rule), conclusion))

print(f"Inferred: {conclusion}")

applied = True

if not applied:

break *# No more rules can be applied*

return inferred

*# Define the facts and rules*

initial\_facts = {"Rain", "Umbrella"}

rules = [

({"Rain"}, "Wet"), *# Rule 1: If Rain, then Wet*

({"Wet", "Umbrella"}, "Dry") *# Rule 2: If Wet and Umbrella, then Dry*

]

*# Run the Forward Chaining algorithm*

final\_facts = forward\_chaining(initial\_facts, rules)

print("\nFinal Inferred Facts:", final\_facts)

**Output:**

Inferred: Wet

Inferred: Dry

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# ASSIGNMENT NO-8

**8. Backward Chaining Algorithm**

**Problem Statement:**

Implement a backward chaining algorithm to reason backward from a goal (e.g., identifying which facts support a conclusion).

**Objective:**

• Use backward chaining to determine the prerequisites for a given conclusion.

• Implement a recursive search to find solutions.

**Theory:**

In backward chaining, the system starts with a goal and works backward, checking if known facts support the conclusion.

**Input:**

def backward\_chaining(goal, facts, rules, visited=None):

if visited is None:

visited = set() *# Track visited goals to avoid infinite loops*

*# Check if the goal is already a known fact*

if goal in facts:

print(f"Goal '{goal}' is a known fact.")

return True

*# Avoid cycles in recursive search*

if goal in visited:

print(f"Already visited '{goal}', avoiding infinite recursion.")

return False

visited.add(goal) *# Mark the goal as visited*

*# Search for rules that conclude the goal*

for premises, conclusion in rules:

if conclusion == goal:

print(f"Trying to satisfy the goal '{goal}' with rule: {premises} -> {goal}")

*# Recursively check if all premises are satisfied*

if all(backward\_chaining(premise, facts, rules, visited) for premise in premises):

return True

*# If no rule or fact can satisfy the goal, it fails*

print(f"Goal '{goal}' cannot be satisfied.")

return False

*# Define the facts and rules*

facts = {"Rain", "Umbrella"}

rules = [

({"Rain"}, "Wet"), *# Rule 1: If Rain, then Wet*

({"Wet", "Umbrella"}, "Dry") *# Rule 2: If Wet and Umbrella, then Dry*

]

*# Define the goal*

goal = "Dry"

*# Run the Backward Chaining algorithm*

if backward\_chaining(goal, facts, rules):

print(f"\nGoal '{goal}' is achieved!")

else:

print(f"\nGoal '{goal}' is not achievable.")

**Output:**

Trying to satisfy the goal 'Dry' with rule: {'Wet', 'Umbrella'} -> Dry

Trying to satisfy the goal 'Wet' with rule: {'Rain'} -> Wet

Goal 'Rain' is a known fact.

Goal 'Umbrella' is a known fact.

Goal 'Dry' is achieved!

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# ASSIGNMENT NO-9

**9. Chatbot Application for a Real-World Scenario**

**Problem Statement:**

Create a chatbot for a customer support application or educational assistant using NLP techniques.

**Objective:**

• Develop a chatbot capable of answering questions, providing guidance, and interacting with users.

• Use AI/ML models or simple rule-based systems to handle conversations.

**Theory:**

A chatbot is an AI-based system designed to simulate human conversation. NLP techniques help it understand user input, and predefined intents guide its responses.

**Input:**

**Output:**