Name: Parth Das

Sap: 60004220185

Roll No: C-111

Artificial Intelligence **Experiment 1**

1. AI for Personalized Marketing

Problem Statement: Design an AI system to create personalized marketing strategies and content tailored to individual customer preferences and behaviors.

Description: This problem involves creating an AI system that uses customer data to develop customized marketing strategies. The AI analyzes individual customer preferences, behaviors, and past interactions to tailor marketing messages, offers, and advertisements. The goal is to increase engagement, improve conversion rates, and maximize return on investment by delivering highly relevant content to each customer.

PEAS:

- **Performance Measure:** Engagement rates, conversion rates, and return on investment.
- **Environment:** Customer data, marketing channels, and product information.
- **Actuators:** Personalized ads, email campaigns, and product recommendations.
- **Sensors:** Customer interaction data, purchase history, and browsing patterns.

---

2. Predictive Text Input

Problem Statement: Develop an AI-powered predictive text input system to enhance typing efficiency and accuracy on mobile and desktop devices.

Description: This problem centers on developing an AI system that predicts and suggests text as users type on their devices. By analyzing typing patterns, user context, and linguistic data, the system aims to enhance typing efficiency and accuracy. The AI predicts the next word or phrase, offers autocomplete suggestions, and corrects errors in real-time, making typing faster and reducing user frustration.

PEAS:

- **Performance Measure:** Typing speed, accuracy, and user satisfaction.
- **Environment:** Text input fields, user typing patterns, and language models.

- **Actuators:** Predictive text suggestions, auto-corrections, and word completions.
- **Sensors:** User keystrokes, text input history, and contextual information.

---

3. AI for Drug Discovery

Problem Statement: Create an AI system to accelerate drug discovery by identifying potential drug candidates and predicting their effectiveness.

Description: This problem involves creating an AI system to accelerate the process of discovering new drugs. The AI analyzes vast amounts of chemical and biological data to identify potential drug candidates and predict their effectiveness against diseases. By leveraging machine learning algorithms, the system helps researchers focus on the most promising compounds, reducing the time and cost associated with drug development.

PEAS:

- **Performance Measure:** Discovery rate of viable drug candidates, accuracy of predictions, and reduction in research time.
- **Environment:** Chemical compound databases, biological data, and research literature.
- **Actuators:** Drug candidate recommendations, research prioritization, and experimental suggestions.
- **Sensors:** Molecular data, biological assays, and computational models.

---

4. Autonomous Vehicle Navigation

Problem Statement: Design an AI system to enable autonomous vehicles to navigate safely and efficiently through diverse and dynamic environments.

Description: This problem focuses on designing an AI system for autonomous vehicles that can navigate safely and efficiently through various environments. The system must handle diverse traffic conditions, interpret road signs and signals, and avoid obstacles such as pedestrians and other vehicles. The AI integrates data from sensors like cameras, LIDAR, and radar to make real-time driving decisions and ensure safe navigation.

PEAS:

- **Performance Measure:** Safety, navigation accuracy, efficiency, and compliance with traffic laws.
- **Environment:** Road networks, traffic conditions, and environmental factors.
- **Actuators:** Steering, acceleration, braking, and signaling systems.
- **Sensors:** Cameras, LIDAR, radar, GPS, and IMU.

5. AI for Financial Market Prediction

Problem Statement: Develop an AI model to predict stock market trends and assist in making informed investment decisions.

Description: This problem involves developing an AI model to predict financial market trends, aiding investors in making informed decisions. The AI analyzes historical market data, economic indicators, and financial news to forecast stock prices and market movements. The goal is to provide actionable insights that can help in making strategic investment decisions and managing financial risk.

PEAS:

- **Performance Measure:** Prediction accuracy, profitability of investment strategies, and risk management.
- **Environment:** Financial data, market trends, and economic indicators.
- **Actuators:** Investment recommendations, trading signals, and portfolio adjustments.
- **Sensors:** Stock prices, trading volume, and financial news.

---

6. Speech Recognition for Virtual Assistants

Problem Statement: Create an AI system for accurate speech recognition to enhance the functionality of virtual assistants in understanding and responding to user commands.

Description: This problem entails creating an AI system that accurately recognizes and processes human speech to enhance virtual assistants. The system must understand spoken commands, queries, and conversations in real-time, converting them into actionable responses or tasks. The AI improves user interaction by accurately interpreting speech and providing relevant responses or executing commands.

PEAS:

- **Performance Measure:** Recognition accuracy, response time, and user satisfaction.
- **Environment:** Audio input, user commands, and contextual information.
- **Actuators:** Voice responses, action execution, and command processing.
- **Sensors:** Microphones, speech processing algorithms, and language models.

---

7. AI-Driven Healthcare Diagnosis

Problem Statement: Design an AI system to assist healthcare professionals in diagnosing diseases based on patient symptoms and medical history.

Description: This problem focuses on developing an AI system to support healthcare professionals in diagnosing medical conditions. By analyzing patient symptoms, medical history, and diagnostic data, the AI provides diagnostic suggestions and treatment recommendations. The aim is to enhance diagnostic accuracy, support decision-making, and improve patient outcomes.

PEAS:

- **Performance Measure:** Diagnostic accuracy, response time, and integration with medical records.
- **Environment:** Patient symptoms, medical history, and diagnostic guidelines.
- **Actuators:** Diagnostic suggestions, treatment recommendations, and alerts.
- **Sensors:** Patient data, medical records, and symptom descriptions.

---

8. AI for Real-Time Language Translation

Problem Statement: Develop an AI system to provide real-time language translation for seamless communication in multilingual contexts.

Description: This problem involves designing an AI system that translates languages in real-time, facilitating communication in multilingual settings. The system must handle both spoken and written text, providing accurate and contextually relevant translations. It aims to break down language barriers, enabling seamless communication across different languages.

PEAS:

- **Performance Measure:** Translation accuracy, latency, and user satisfaction.
- **Environment:** Text or speech input, languages, and context.
- **Actuators:** Translated text or speech output.
- **Sensors:** Language models, translation algorithms, and speech recognition tools.

---

9. AI for E-commerce Product Recommendations

Problem Statement: Create an AI system to recommend products to users based on their browsing behavior and purchase history.

Description: This problem centers on creating an AI system that generates personalized product recommendations for e-commerce platforms. The system uses user browsing history, purchase patterns, and product attributes to suggest items that match individual preferences. The goal is to enhance the shopping experience and increase sales by providing relevant product suggestions.

PEAS:

- **Performance Measure:** Recommendation accuracy, user engagement, and sales increase.
- **Environment:** User browsing history, product catalog, and purchase data.
- **Actuators:** Product recommendations, personalized ads, and content suggestions.
- **Sensors:** User behavior data, purchase history, and product attributes.

---

10. AI for Dynamic Pricing in Retail

Problem Statement: Design an AI system to optimize retail pricing dynamically based on market conditions, demand, and competitor pricing.

Description: This problem involves developing an AI system to adjust retail prices dynamically based on market conditions, demand, and competitor pricing. The AI analyzes real-time data to optimize pricing strategies, aiming to maximize revenue, maintain competitiveness, and respond to market fluctuations effectively.

PEAS:

- **Performance Measure:** Revenue maximization, price accuracy, and market competitiveness.
- **Environment:** Market conditions, demand data, and competitor pricing.
- **Actuators:** Price adjustments, promotional offers, and discount strategies.
- **Sensors:** Sales data, competitor prices, and customer demand patterns.

Problem Statement	Observable	Deterministic	Episodic	Static	Discrete	Agents
1. AI for Personalized Marketing	Partially	Stochastic	Sequential	Dynamic	Discrete	Single (Marketing AI)
2. Predictive Text Input	Observable	Stochastic	Sequential	Static	Discrete	Single (Text Input AI)
3. AI for Drug Discovery	Observable	Stochastic	Sequential	Dynamic	Continuous	Single (Discovery AI)
4. Autonomous Vehicle Navigation	Fully Observable	Stochastic	Sequential	Dynamic	Continuous	Single (Autonomous Vehicle)

5. AI for Financial Market Prediction	Partially	Stochastic	Sequential	Dynamic	Continuous	Single (Prediction AI)
6. Speech Recognition for Virtual Assistants	Observable	Stochastic	Sequential	Static	Discrete	Single (Virtual Assistant)
7. AI-Driven Healthcare Diagnosis	Observable	Stochastic	Sequential	Static	Discrete	Single (Healthcare AI)
8. AI for Real-Time Language Translation	Observable	Stochastic	Sequential	Static	Discrete	Single (Translation AI)
9. AI for E-commerce Product Recommendations	Observable	Stochastic	Sequential	Static	Discrete	Single (Recommendation AI)
10. AI for Dynamic Pricing in Retail	Observable	Stochastic	Sequential	Dynamic	Discrete	Single (Pricing AI)

Conclusion:-

Understanding the distinctions between fully and partially observable, static and dynamic, and deterministic and stochastic environments is essential in various fields. Fully observable environments provide complete information, simplifying decision-making, while partially observable environments require inference and probabilistic reasoning. Static environments remain unchanged unless acted upon, whereas dynamic environments continuously evolve due to external factors. Deterministic systems yield predictable outcomes given initial conditions, whereas stochastic systems incorporate randomness, leading to variable outcomes. These concepts guide the design and implementation of algorithms and systems in AI, robotics, game development, and beyond, ensuring they effectively handle the nature and complexity of the environments they operate in.

Name: Parth Das
Sap: 60004220185
Roll No:- C-111
AI Experiment 2

Aim:- Implement BFS and DFS algorithm

Parth Das
60004220185

C-111

AI Experiment-2

Aim:- To implement BFS & DFS to reach goal state

Theory:- 1) Breadth First Search (BFS):-
BFS is a graph traversal algorithm that starts at a source node and explores all its neighbours at the present depth level before moving on to nodes at the next depth level. It uses a queue to keep track of the nodes to be explored next.

Algorithm:-
1. Initialize a queue and enqueue the starting node. mark the starting node as visited.
2. While the queue is not empty:
   - Dequeue a node from front of the queue.
   - For each unvisited neighbour of the dequeued node:
     - → mark the neighbour as visited
     - → Enqueue the neighbour in the queue.
3. Repeat until the queue is empty

Observation:-
1) Layered Exploration:-
BFS explores nodes level by level. Nodes are visited in order of their distance from the starting node.
2) Shortest Path:-
BFS finds the shortest path in terms of the number of edges from the starting node to any node in an unweighted graph
3) Use of Queue:-
The queue ensures nodes are explored in the order they are discovered, providing a systematic level-order traversal.

Time complexity:- O(V+E) V: vertices E: edges
Space complexity:- O(V)

2) Depth First Search (DFS):

DFS is a graph traversal algorithm that starts at source node and explores as far as possible along each branch before backtracking. It uses a stack (or recursion) to keep track of nodes to be explored.

Algorithm:-

1. Initialize stack and push starting node onto the stack. Mark the starting node as visited.
2. While the stack is not empty:
   - Pop a node from the top of the stack.
   - For each unvisited neighbour of the popped node:
     → Mark the neighbour as visited
     → Push the neighbour onto the stack.
3. Repeat until stack is empty.

Observation:-

1) Deep Exploration:- DFS explores as deep as possible along a path before backtracking. It does not explore nodes level by level but rather follows a single path to its end.

2) Path Discovery:- DFS can be used to find a path between nodes and is ~~well~~ suited for problems requiring exploration of all possible paths.

3) Use of Stack:- The stack ensures that nodes are revisited in the reverse order of their discovery, facilitating deep exploration and backtracking. T.C = O(V+E) V: Vertices

S.C :- O(V) E: Edges

Conclusion:-

Both BFS & DFS are essential graph algorithms, each with its own strength & appropriate use cases. BFS is favored for level-order exploration and shortest path in unweighted graphs while DFS excels in deep exploration and pathfinding. Understanding their mechanisms helps in selecting right algorithm based on problem requirements.

Code:-

BFS.cpp

```
#include <iostream>
#include <vector>
#include <limits>
#include <queue>
using namespace std;
int path_cost;
// BFS implementation with path cost calculation
bool bfs(const vector<vector<int>>& graph, const vector<vector<int>>&
cost, int start, int goal, int numNodes, vector<bool>& visited) {
    vector<int> costToNode(numNodes, numeric_limits<int>::max()); //
Initialize costs to a large value
    queue<int> q;
```

```
    q.push(start);
    visited[start] = true;
    costToNode[start] = 0;
    while (!q.empty()) {
        int current = q.front();
        q.pop();
        cout << "Visited node: " << current << endl;
        // Check if we have reached the goal
        if (current == goal) {
            path_cost = costToNode[goal];
            return true;
        }
        // Process all adjacent nodes
        for (int i = 0; i < numNodes; i++) {
            if (graph[current][i] && !visited[i]) {
                // Update cost to reach the adjacent node
                if (costToNode[current] + cost[current][i] <
costToNode[i]) {
                    costToNode[i] = costToNode[current] +
cost[current][i];
                }
                q.push(i);
                visited[i] = true;
            }
        }
    }
    return false;
}
int main() {
    int numNodes = 6;
    int start = 0;
    int goal = 4;
    // Initialize matrices
    vector<vector<int>> graph(numNodes, vector<int>(numNodes, 0)); //
Adjacency matrix
    vector<vector<int>> cost(numNodes, vector<int>(numNodes,
numeric_limits<int>::max())); // Cost matrix
    vector<bool> visited(numNodes, false); // Visited array
    graph[0][1] = 1;
    graph[0][2] = 1;
    graph[1][3] = 1;
    graph[2][3] = 1;
    graph[3][4] = 1;
    graph[4][5] = 1;
    cost[0][1] = 22;
    cost[0][2] = 24;
    cost[1][3] = 11;
    cost[2][3] = 3;
```

```
    cost[3][4] = 15;
    cost[4][5] = 19;
    if (bfs(graph, cost, start, goal, numNodes, visited)) {
        cout << "Path to goal node " << goal << " found." << endl;
        cout << "Path cost is " << path_cost << endl;
    } else {
        cout << "No path to goal node " << goal << "." << endl;
    }

    cout << "Non-visited nodes:" << endl;
    for (int i = 0; i < numNodes; i++) {
        if (!visited[i]) {
            cout << "Node " << i << endl;

        }
    }
    return 0;
}
```

Output:-

```
Visited node: 0
Visited node: 1
Visited node: 2
Visited node: 3
Visited node: 4
Path to goal node 4 found.
Path cost is 48
Non-visited nodes:
Node 5
```

Code:-

DFS.cpp

```
#include <iostream>
#include <vector>
#include <limits>

using namespace std;
#define MAX_NODES 100
int path_cost;
bool dfs(const vector<vector<int>>& graph, const vector<vector<int>>&
cost, int current, int goal, vector<bool>& visited, int numNodes,
vector<int>& costToNode) {
    // Mark the current node as visited
    visited[current] = true;
    cout << "Visited node: " << current << endl;
    if (current == goal) {
        path_cost = costToNode[goal];
```

```
        return true;
    }
    for (int i = 0; i < numNodes; i++) {
        if (graph[current][i] && !visited[i]) {
            if (costToNode[current] + cost[current][i] < costToNode[i])
{
                costToNode[i] = costToNode[current] + cost[current][i];
            }
            if (dfs(graph, cost, i, goal, visited, numNodes,
costToNode)) {
                return true;
            }
        }
    }
    return false;
}
int main() {
    vector<vector<int>> graph(MAX_NODES, vector<int>(MAX_NODES, 0)); //
Adjacency matrix
    vector<vector<int>> cost(MAX_NODES, vector<int>(MAX_NODES,
numeric_limits<int>::max())); // Cost matrix
    vector<bool> visited(MAX_NODES, false);
    vector<int> costToNode(MAX_NODES, numeric_limits<int>::max()); //
Cost to node array
    int numNodes, start, goal;
    numNodes = 6;
    path_cost = 0;
    graph[0][1] = 1;
    graph[0][2] = 1;
    graph[1][3] = 1;
    graph[2][3] = 1;
    graph[3][4] = 1;
    graph[4][5] = 1;
    cost[0][1] = 22;
    cost[0][2] = 24;
    cost[1][3] = 11;
    cost[2][3] = 3;
    cost[3][4] = 15;
    cost[4][5] = 19;

    start = 0;
    goal = 4;
    costToNode[start] = 0;
    if (dfs(graph, cost, start, goal, visited, numNodes, costToNode)) {
        cout << "Path to goal node " << goal << " found." << endl;
        cout << "Path cost is " << path_cost << endl;
    } else {
        cout << "No path to goal node " << goal << "." << endl;
```

```
    }
    cout << "Non-visited nodes:" << endl;
    for (int i = 0; i < numNodes; i++) {
        if (!visited[i]) {
            cout << "Node " << i << endl;
        }
    }

    return 0;
}
```

Output:-

```
Visited node: 0
Visited node: 1
Visited node: 3
Visited node: 4
Path to goal node 4 found.
Path cost is 48
Non-visited nodes:
Node 2
Node 5
```

Name: Parth Das
Sap: 60004220185
Roll No:- C-111
AI Experiment 3

DFID Implementation

Parth Das
60004220185

C-111

AI Experiment - 3

Aim:- To implement DFID search algorithm to reach goal.

Theory:- Depth First Iterative Deepening (DFID):-
DFID is a graph traversal algorithm that combines the space efficiency of Depth-First Search (DFS) with the completeness of Breadth-First Search (BFS). It performs a series of depth limited DFS searches with increasing depth limits until the goal is found. This approach ensures that every level of the search tree is explored in a depth-first manner, but with the advantage of eventually exploring all possible depths.

Algorithm:-
1. Initialize:- Set initial depth limit to 0.
2. Iterative Search:-
   - Perform a depth limited DFS with current depth limit.
   - If goal is found during DFS, terminate & return the solution.
   - If not found, increase the depth limit by 1 and repeat the search.
3. Depth-limited DFS:-
   - Initialize a stack & push the starting node onto the stack. Mark the starting node as visited.
   - While stack is not empty:-
     → Pop node from top of the stack
     → If node is the goal, return the solution
     → For each unvisited neighbour of the node:-
       →→ If the depth limit of neighbour is within current depth limit, mark it as visited and push it onto the stack.
   - If the search reaches the depth limit without finding the goal, return to iterative phase with an increased depth limit.

FOR EDUCATIONAL USE

Observation:-

1) Completeness:-
DFID is complete, meaning it will find the goal if it exists, because it incrementally increases the depth limit until all possible nodes are explored.

2) Optimality:-
In an unweighted graph, DFID will find the shortest path to the goal, as it explores nodes level by level in increasing depth.

3) Space Efficiency:-
DFID uses less ~~less~~ memory compared to BFS. Although it performs multiple depth limited searches, each individual search uses the space complexity of DFS $O(d)$ where 'd' is current depth limit.

4) Redundant Work:-
DFID performs redundant work at each depth-limited search revisits nodes at shallower depths. However, this is generally outweighted by its space efficiency and completeness.

Conclusion:-
DFID is a hybrid search strategy that combines the strengths of DFS and BFS. It is ideal for problems where depth of the solution is unknown and can be bounded by depth limit. DFID guarantees finding the solution if one exists while maintaining space efficiency similar to DFS. It is especially useful in scenarios with large or infinite search spaces where BFS would be impractical due to high memory consumption, but a complete search is still required.

Time Complexity:-
The time complexity of DFID is $O(b^d)$ where
b: branching factor and d: depth of shallowest goal.
Despite repeated nodes it remains comparable.

Space Complexity:-
The space complexity of DFID is $O(d)$ where d: depth of shallowest goal. DFID only stores a single path from root node to current node

DFID is complete for finite search tree ensuring that it will find a solution if it exists.

DFID is optimal when all edges (or costs) are uniform, i.e. will find solution with min depth or cost.

8/9/24

Code:

```
#include <bits/stdc++.h>
using namespace std;

class Solution
{
public:
    vector<int> dfsOfGraph(int V, vector<pair<int, int>> adj[], int
goal, int maxDepth)
    {
        vector<int> bfs;
        vector<int> cost(V, INT_MAX);
        vector<bool> vis(V, false);
        vector<int> depth(V, -1);
        queue<tuple<int, int, int>> q;
        q.push({0, 0, 0});
        bool goalFound = false;
        int foundDepth = -1;
        unordered_map<int, vector<int>> nodesAtDepth;
        while (!q.empty())
        {
            int node, currentCost, currentDepth;
            tie(node, currentCost, currentDepth) = q.front();
            q.pop();
            if (currentDepth > maxDepth)
                continue; // Skip nodes beyond maxDepth
            if (vis[node])
                continue;
            vis[node] = true;
            bfs.push_back(node);
            cost[node] = currentCost;
            depth[node] = currentDepth;
            nodesAtDepth[currentDepth].push_back(node);
            if (node == goal)
            {
                goalFound = true;
                foundDepth = currentDepth;
            }
            for (auto it : adj[node])
            {
                int neighbor = it.first;
                int edgeCost = it.second;
                if (!vis[neighbor])
                {
                    q.push({neighbor, currentCost + edgeCost,
currentDepth + 1});
                }
            }
```

```
    }
    for (int d = 0; d <= maxDepth; ++d)
    {
        cout << "At depth " << d << ":" << endl;
        set<int> visitedNodes;
        for (int i = 0; i <= d; ++i)
        {
            for (int node : nodesAtDepth[i])
            {
                visitedNodes.insert(node);
            }
        }
        cout << "Visited nodes: ";
        for (int node : visitedNodes)
        {
            cout << node << " ";
        }
        cout << endl;
        cout << "Unvisited nodes: ";
        vector<int> unvisitedNodes;
        for (int i = 0; i < V; ++i)
        {
            if (visitedNodes.find(i) == visitedNodes.end())
            {
                unvisitedNodes.push_back(i);
            }
        }
        sort(unvisitedNodes.rbegin(), unvisitedNodes.rend());
        for (int node : unvisitedNodes)
        {
            cout << node << " ";
        }
        cout << endl;
        cout << "Path cost to goal: ";
        if (cost[goal] != INT_MAX)
        {
            cout << cost[goal] << endl;
        }
        else
        {
            cout << "Goal not reachable" << endl;
        }
        cout << endl;
    }
    if (goalFound)
    {
        cout << "Goal found at depth: " << foundDepth << endl;
    }
```

```cpp
        else
        {
            cout << "Goal not found within depth limit" << endl;
        }
        return bfs;
    }
    void printGraph(int V, vector<pair<int, int>> adj[])
    {
        cout << "Graph representation:" << endl;
        for (int i = 0; i < V; i++)
        {
            cout << "Node " << i << ": ";
            for (auto it : adj[i])
            {
                cout << "(" << it.first << ", " << it.second << ") ";
            }
            cout << endl;
        }
    }
};
void addEdge(vector<pair<int, int>> adj[], int u, int v, int cost)
{
    adj[u].push_back({v, cost});
    adj[v].push_back({u, cost});
}
int main()
{
    int V = 14;
    vector<pair<int, int>> adj[V];
    addEdge(adj, 0, 1, 2);
    addEdge(adj, 0, 2, 3);
    addEdge(adj, 1, 3, 4);
    addEdge(adj, 1, 4, 7);
    addEdge(adj, 2, 5, 8);
    addEdge(adj, 3, 7, 1);
    addEdge(adj, 2, 6, 2);
    addEdge(adj, 4, 9, 2);
    addEdge(adj, 4, 10, 3);
    addEdge(adj, 5, 11, 5);
    addEdge(adj, 11, 8, 5);
    addEdge(adj, 6, 12, 10);
    addEdge(adj, 9, 13, 1);
    Solution obj;
    obj.printGraph(V, adj);
    int goal = 13;
    int maxDepth = 4;
    vector<int> ans = obj.dfsOfGraph(V, adj, goal, maxDepth);
    return 0;}
```

Output:-

```
Graph representation:
Node 0: (1, 2) (2, 3)
Node 1: (0, 2) (3, 4) (4, 7)
Node 2: (0, 3) (5, 8) (6, 2)
Node 3: (1, 4) (7, 1)
Node 4: (1, 7) (9, 2) (10, 3)
Node 5: (2, 8) (11, 5)
Node 6: (2, 2) (12, 10)
Node 7: (3, 1)
Node 8: (11, 5)
Node 9: (4, 2) (13, 1)
Node 10: (4, 3)
Node 11: (5, 5) (8, 5)
Node 12: (6, 10)
Node 13: (9, 1)
At depth 0:
Visited nodes: 0
Unvisited nodes: 13 12 11 10 9 8 7 6 5 4 3 2 1
Path cost to goal: 12
```

```
At depth 0:
Visited nodes: 0
Unvisited nodes: 13 12 11 10 9 8 7 6 5 4 3 2 1
Path cost to goal: 12

At depth 1:
Visited nodes: 0 1 2
Unvisited nodes: 13 12 11 10 9 8 7 6 5 4 3
Path cost to goal: 12

At depth 2:
Visited nodes: 0 1 2 3 4 5 6
Unvisited nodes: 13 12 11 10 9 8 7
Path cost to goal: 12

At depth 3:
Visited nodes: 0 1 2 3 4 5 6 7 9 10 11 12
Unvisited nodes: 13 8
Path cost to goal: 12

At depth 4:
Visited nodes: 0 1 2 3 4 5 6 7 8 9 10 11 12 13
Unvisited nodes:
Path cost to goal: 12

Goal found at depth: 4
```

Name:Parth Das
Sap: 60004220185
Roll No: C-111
AI Experiment 4
A* Algorithm

Name:- Parth Das
SAP:- 60004220185

C-111

AI Experiment 4

Aim:- To implement Informed Search A* Algorithm

Theory:-
Informed search algorithms use additional information, typically in the form of a heuristic, to guide the search process towards the goal more efficiently than uninformed search algorithms. Informed search tries to improve the search process by estimating the cost from the current state to the goal, often speeding up the search by exploring fewer nodes. These algorithms make use of heuristics to evaluate which path seems most promising.

A* Algorithm
This is a graph traversal and pathfinding algorithm that uses heuristics to find the least cost path from a given starting node to a goal node. It is widely used due to its ability to find an optimal solution efficiently. It combines the benefits of both Dijkstra's Algorithm and Greedy Best-First Search.

Steps:-
1. Initialize:
   - Open priority queue that stores nodes to be evaluated
   - closed list (or set) that stores nodes already evaluated.
   - Start node with cost $g(start) = 0$ and an estimated total cost $f(start) = g(start) + h(start)$.
2. Select the Node:-
   - Pick node with lowest $f(n) = g(n) + h(n)$ from open list
     - $g(n)$ is cost from start node to current node n
     - $h(n)$ is heuristic estimate from node n to goal node.

3. Goal Test.
- If current node is the goal node, return solution (path & cost)

4. Expand the Node:
- Generate the neighbors of the current node
- For each neighbor:
  - Calculate g(neighbor) = g(current) + cost(current, neighbor)
  - Update f(neighbor) = g(neighbor) + h(neighbor)
  - If this neighbor has not been evaluated before (or if it offers a cheaper path), add it to the open list.

5. Repeat:-
- Move the current node to the closed list and continue selecting the node with the smallest f(n) from the open list until the goal is reached or open list is empty.
- Once the goal node is reached, backtrack through the parents of the nodes to construct the final path.

Features:-
- Completeness:- A^* is complete, meaning it will find a solution if one exists, as long as the heuristic h(n) is admissible (does not overestimate the actual cost).

- Optimality:- A^* is optimal if the heuristic is admissible and consistent. It guarantees the shortest path (least cost) to the goal.

- Time Complexity:- The time complexity of A^* depends on the branching factor b, depth of solution d, and the heuristic's accuracy. In worst case, it may explore a large portion of the graph, leading to exponential time complexity $O(b^d)$

• Space complexity: A* stores all explored nodes in memory, so its space complexity is also $O(b^d)$. Memory requirements are typically a limiting factor of A*

Heuristic Function:-

• Admissible heuristic:- The heuristic $h(n)$ is admissible if it never overestimates the cost to reach the goal. For eg. Straight line distance is an admissible heuristic for pathfinding problems in grinds or graphs.

• Consistent heuristic:- A heuristic is consistent if, for every node n and its neighbor n', the estimated cost of reaching the goal frome n is no greater than the cost of reaching the neighbor n' plus the estimated cost of reaching the goal from n'.

$$h(n) \leq cost(n, n') + h(n')$$

Limitations:-

~~Also~~

Memory usage:- A* requires storing all explored nodes, which can result in high memory usage for large search spaces.

Conclusion:-

While A* is complete and guarantees optimal solution, its time and space complexity can become a limiting factor, especially in large search spaces, due to its requirement to store all explored nodes. Despite this, A* remains one of the most efficient search algorithms for a variety of pathfinding and graph traversal problems when the heuristic function is well-chosen.

20/9/24

FOR EDUCATIONAL USE

Code:-

```
#include<bits/stdc++.h>

using namespace std;
// Node structure to store the node, gCost, hCost, and fCost
struct Node {
    string name;
    int gCost, hCost, fCost;
    Node(string name_, int gCost_, int hCost_)
        : name(name_), gCost(gCost_), hCost(hCost_) {
        fCost = gCost + hCost;
    }
    // Comparator for priority queue (min-heap based on fCost)
    bool operator<(const Node& other) const {
```

```cpp
        return fCost > other.fCost;
    }
};
// A* algorithm implementation
void AStar(unordered_map<string, vector<pair<string, int>>> graph,
unordered_map<string, int> heuristics, string start, string goal) {
    // Priority queue for open nodes (min-heap)
    priority_queue<Node> openList;
    // Cost map to store gCost of nodes (cost from start)
    unordered_map<string, int> gCostMap;

    // Parent map to track the path (backtracking from goal to start)
    unordered_map<string, string> parentMap;
    // Set gCost of all nodes to infinity initially
    for (const auto& node : graph) {
        gCostMap[node.first] = numeric_limits<int>::max();
    }
    // Closed set to track visited nodes
    unordered_map<string, bool> closedSet;
    openList.push(Node(start, 0, heuristics[start]));
    gCostMap[start] = 0;
    parentMap[start] = "";  // Start node has no parent
    int iteration = 0;
    while (!openList.empty()) {
        iteration++;
        Node current = openList.top();
        openList.pop();
        cout << "Iteration " << iteration << ": Processing node " <<
current.name << " with fCost: " << current.fCost << " (gCost: " <<
current.gCost << ", hCost: " << current.hCost << ")\n";
        if (current.name == goal) {
            cout << "Goal '" << goal << "' reached with total cost: "
<< current.gCost << endl;
            stack<string> path;
            string node = goal;
            while (node != "") {
                path.push(node);
                node = parentMap[node];
            }
            cout << "Path: ";
            while (!path.empty()) {
                cout << path.top();
                path.pop();
                if (!path.empty()) cout << " -> ";
            }
            cout << endl;
            return;
        }
```

```cpp
        closedSet[current.name] = true;
        for (const auto& neighbor : graph[current.name]) {
            string neighborName = neighbor.first;
            int pathCost = neighbor.second;
            if (closedSet[neighborName]) continue;
            int newGCost = current.gCost + pathCost;
            if (newGCost < gCostMap[neighborName]) {
                gCostMap[neighborName] = newGCost;
                int hCost = heuristics[neighborName];
                // Push the neighbor to the priority queue
                openList.push(Node(neighborName, newGCost, hCost));
                parentMap[neighborName] = current.name;
                cout << "  Adding neighbor " << neighborName << " with
gCost: " << newGCost << ", hCost: " << hCost << ", fCost: " << newGCost
+ hCost << endl;
            }
        }
        cout << "  Priority Queue: ";
        priority_queue<Node> tempQueue = openList;
        while (!tempQueue.empty()) {
            Node node = tempQueue.top();
            tempQueue.pop();
            cout << node.name << "(fCost: " << node.fCost << ") ";
        }
        cout << endl << endl;
    }
    cout << "Goal '" << goal << "' cannot be reached.\n";
}
int main() {
    unordered_map<string, int> heuristics = {
        {"A", 10}, {"B", 8}, {"C", 5}, {"D", 7}, {"E", 3},
        {"F", 6}, {"G", 5}, {"H", 3}, {"I", 1}, {"J", 0}
    };
    unordered_map<string, vector<pair<string, int>>> graph = {
        {"A", {{"B", 6}, {"F", 3}}},
        {"B", {{"A", 6}, {"C", 3}, {"D", 2}}},
        {"C", {{"B", 3}, {"D", 1}, {"E", 5}}},
        {"D", {{"B", 2}, {"C", 1}, {"E", 8}}},
        {"E", {{"C", 5}, {"D", 8}, {"I", 5}, {"J", 5}}},
        {"F", {{"A", 3}, {"H", 7}, {"G", 1}}},
        {"G", {{"F", 1}, {"I", 3}}},
        {"H", {{"F", 7}, {"I", 2}}},
        {"I", {{"E", 5}, {"G", 3}, {"H", 2}, {"J", 3}}},
        {"J", {{"E", 5}, {"I", 3}}}
    };
    string start = "A";
    string goal = "J";
    AStar(graph, heuristics, start, goal);
```

```
    return 0;
}
```

Output:-

```
Iteration 1: Processing node A with fCost: 10 (gCost: 0, hCost: 10)
  Adding neighbor B with gCost: 6, hCost: 8, fCost: 14
  Adding neighbor F with gCost: 3, hCost: 6, fCost: 9
  Priority Queue: F(fCost: 9) B(fCost: 14)

Iteration 2: Processing node F with fCost: 9 (gCost: 3, hCost: 6)
  Adding neighbor H with gCost: 10, hCost: 3, fCost: 13
  Adding neighbor G with gCost: 4, hCost: 5, fCost: 9
  Priority Queue: G(fCost: 9) H(fCost: 13) B(fCost: 14)

Iteration 3: Processing node G with fCost: 9 (gCost: 4, hCost: 5)
  Adding neighbor I with gCost: 7, hCost: 1, fCost: 8
  Priority Queue: I(fCost: 8) H(fCost: 13) B(fCost: 14)

Iteration 4: Processing node I with fCost: 8 (gCost: 7, hCost: 1)
  Adding neighbor E with gCost: 12, hCost: 3, fCost: 15
  Adding neighbor H with gCost: 9, hCost: 3, fCost: 12
  Adding neighbor J with gCost: 10, hCost: 0, fCost: 10
  Priority Queue: J(fCost: 10) H(fCost: 12) H(fCost: 13) B(fCost: 14) E(fCost: 15)

Iteration 5: Processing node J with fCost: 10 (gCost: 10, hCost: 0)
Goal 'J' reached with total cost: 10
Path: A -> F -> G -> I -> J
```

Name: Parth Das
SAP:- 60004220185
Roll No: C-111
AI EXperiment 5
Hill Climb Alogorithm

Name:- Parth Das
SAP:- 60004220185

C-111

AI Experiment 5

Aim:- To implement Hill Climb Algorithm

Theory:-
Hill climbing is an informed search algorithm that uses heuristic functions to search for optimal solutions in a search space. The goal is to find the best possible solution by making iterative moves towards higher (or lower) heuristic values. It's is a local search algorithm, meaning it explores neighbouring states of the current solution in the search space, trying to find a better solution at each step.

Block World Problem
In this problem, we aim to rearrange blocks into a desired configuration using a set of allowed moves. Hill climbing can be used to find optimal sequence of moves. The heuristic function evaluates how far the current configuration is from the goal, and the algorithm continuously improves the configuration by making incremental changes.

Algorithm:-
1. Initialize:
   - Start with initial configuration of blocks
2. Evaluate the current State:
   - Compute the heuristic value based on how far the current state is from the goal state.

3. Find Neighbouring States:-
- Generate all possible valid moves (neighbour states) from the current state.
- Each neighbour is created by moving a block either onto the table or onto another block.

4. Choose the Best Neighbour:
- Evaluate the heuristic value of each neighbour and choose the one that reduces the heuristic (brings the state closer to the goal)

5. Move to the Best Neighbour:
- If the best neighbour is an improvement, move to it. Otherwise, stop (local optimum reached).

6. Repeat
- Continue this process until the current state matches the goal state or a local optimum is reached.

Features:-

- Completeness:-

Hill climb is not complete, it may get stuck in a local optimum where no neighbouring states improve the ~~sol~~ solution, even if better solution exists elsewhere.

- Optimality:-

Hill climbing is not guaranteed to be optimal because it may find local optima rather than global optimum.

- Space Complexity:-

O(1) as the algorithm only needs to store the current state and the immediate neighbours.

3. Find Neighbouring States:-
- Generate all possible valid moves (neighbour states) from the current state.
- Each neighbour is created by moving a block either onto the table or onto another block.

4. Choose the Best Neighbour:
- Evaluate the heuristic value of each neighbour and choose the one that reduces the heuristic (brings the state closer to the goal)

5. Move to the Best Neighbour:
- If the best neighbour is an improvement, move to it. Otherwise, stop (local optimum reached).

6. Repeat
- Continue this process until the current state matches the goal state or a local optimum is reached.

Features:-
- Completeness:-
Hill climb is not complete, it may get stuck in a local optimum where no neighbouring states improve the ~~sol~~ solution, even if better solution exists elsewhere.

- Optimality:-
Hill climbing is not guaranteed to be optimal because it may find local optima rather than global optimum.

- Space Complexity:-
O(1) as the algorithm only needs to store the current state and the immediate neighbours.

Code:-

```
from copy import deepcopy
def get_position_index(final):
    fs = final[0]
    p = dict()
    for i in range(len(fs)):
        if i == 0:
            p[fs[i]] = None
        else:
            p[fs[i]] = fs[i - 1]
    return p
def get_next_states(s, visited):
    states = []
    for i in range(len(s)):
        if len(s[i]) == 0:
```

```
            continue
        for j in range(len(s)):
            if i == j:
                continue
            ns = deepcopy(s)
            ns[j].append(ns[i].pop())
            if ns not in visited:
                states.append(ns)
    return states
def get_h(s, f):
    c = 0
    for x in s:
        for i in range(len(x)):
            if i == 0:
                c += 1 if f[x[i]] is None else -1
            else:
                c += 1 if f[x[i]] == x[i - 1] else -1
    return c
# Count of maximum platforms to be used
pfc = 3
# Taking user input in a way that works for Python 3
initial_input = input("Enter the initial state (e.g., 'A B C'):
").split()
final_input = input("Enter the final state (e.g., 'A B C'): ").split()
initial = (initial_input, [], [])
final = (final_input, [], [])
fi = get_position_index(final)
visited = []
not_visited = []
current = deepcopy(initial)
moves = 0
while get_h(current, fi) != get_h(final, fi):
    moves += 1
    visited.append(current)
    # Generate possible next states
    ns = get_next_states(current, visited)
    ns_h = []
    # Calculate heuristic value for each state
    for s in ns:
        ns_h.append((get_h(s, fi), s))
    # Sort by the heuristic value
    ns_h = sorted(ns_h, key=lambda x: x[0])
    try:
        # Get the best state to explore
        c, current = ns_h.pop()
        # Add the remaining to not visited for future use
        not_visited = not_visited + ns_h
    except IndexError:
```

```python
        # If there are no new next states, backtrack
        print('No new unique states, backtracking...')
        not_visited = sorted(not_visited, key=lambda x: x[0])
        c, current = not_visited.pop()
    print("Current heuristic value: " + str(c), current)
print('Found solution in ' + str(moves) + ' moves')
```

Output:-

```
Enter the initial state (e.g., 'A B C'): A B C D
Enter the final state (e.g., 'A B C'): B D C A
Current heuristic value: -4 (['A', 'B', 'C'], [], ['D'])
Current heuristic value: -2 (['A', 'B'], [], ['D', 'C'])
Current heuristic value: 0 (['A'], ['B'], ['D', 'C'])
Current heuristic value: 2 ([], ['B'], ['D', 'C', 'A'])
Current heuristic value: 2 (['B'], [], ['D', 'C', 'A'])
Current heuristic value: 0 (['B'], ['A'], ['D', 'C'])
Current heuristic value: 0 (['B', 'A'], [], ['D', 'C'])
Current heuristic value: -2 (['B', 'A'], ['C'], ['D'])
Current heuristic value: 0 (['B'], ['C', 'A'], ['D'])
Current heuristic value: 2 (['B', 'D'], ['C', 'A'], [])
Current heuristic value: 0 (['B', 'D'], ['C'], ['A'])
Current heuristic value: 2 (['B', 'D', 'C'], [], ['A'])
Current heuristic value: 4 (['B', 'D', 'C', 'A'], [], [])
Found solution in 13 moves
```

Name: Parth Das
Sap: 60004220185
Batch: C-22
Roll no: C-111

Subject : Artificial Intelligence

EXPERIMENT NO. 6

Topic: Implementation of Genetic Algorithm

THEORY:

Parth Das
60004220185 C2-2

C-111

AI Experiment-6

Aim:- To implement Genetic Algorithm

Theory:-
A Genetic Algorithm (GA) is a type of ~~comb~~ Optimization algorithm inspired by the process of natural selection in biological evolution. It is widely used to solve optimization and search problems, particularly when the solution space is vast and complex. GAs belong to the family of evolutionary algorithms, which generate solutions to optimization problems using techniques inspired by natural selection.

Key Concepts:-
1. Population:-
Each individual in the population represents a potential solution to be the problem and is often encoded as chromosome (string or array of variables)
2. Fitness Function:-
In AI, this function quantifies how well a solution performs on a given task, whether it's optimization or classification.
3. Selection:-
-> GAs use selection mechanisms to choose individuals from the current population to reproduce.
-> The individual with higher fitness scores are more likely to be selected for the next generation (just like "survival of the fittest")
-> common selection strategies include:-
- Roulette Wheel Selection: Probability of selection ~~is~~ proportional to fitness
- Tournament Selection:- A set of individuals is chosen randomly, and the best among them is selected.

4. Crossover (Recombination):-

-> The crossover operation takes 2 parent individuals and combines parts of their chromosomes to produce offspring (new solutions).

-> The idea is that by combining successful traits from two parents, you can create offspring with even better traits.

5. Mutation:

-> This operation introduces small random changes in the chromosome of an individual.

-> Mutation ensures diversity in the population and helps avoid local optima by exploring new areas of the solution space.

6. Termination:-

-> This algo. repeats the process of selection, crossover and mutation over multiple generations until termination condition is met.

-> This could be a fixed number of generations, reaching a time limit or achieving a certain fitness threshold.

Optimality:- Not guaranteed, approximates near-optimal solutions.

Completeness:- Not complete, may not find a solution in finite time.

Time Complexity:- O(G * P * f(n)) where G = generations, P: Population size, f(n): fitness function cost.

Space Complexity: O(P * n) where P = population size, n = chromosome length.

Conclusion:-

In the context of A I, Genetic Algorithms are useful for problems where search space is vast and poorly understood, and they offer a flexible way to explore solutions through an evolutionary approach. However, they require careful tuning and are typically used in conjunction with other optimization techniques in real world A I applications.

21/10/24

FOR EDUCATIONAL USE

CODE:

```
import random

# Parameters
POPULATION_SIZE = 100
GENOME_LENGTH = 10  # Number of bits (for each dimension)
MUTATION_RATE = 0.1
GENERATIONS = 100

# Function to convert binary to decimal within a specific range
def binary_to_decimal(binary):
```

```
    return int(binary, 2) / (2 ** GENOME_LENGTH - 1) * 5.12 - 2.56  # Scale to [-2.56, 2.56]


# Sphere function
def sphere(genome):
    x = [binary_to_decimal(genome[i:i + GENOME_LENGTH]) for i in range(0, len(genome), GENOME_LENGTH)]
    return sum(xi**2 for xi in x)


# Generate initial population
def generate_population(size):
    return [".join(random.choice('01') for _ in range(GENOME_LENGTH * 2)) for _ in range(size)]  # Double genome length for two dimensions


# Selection: Tournament selection
def selection(population):
    tournament = random.sample(population, 5)
    return min(tournament, key=sphere)  # Minimize the Sphere function


# Crossover: Two-point crossover
def two_point_crossover(parent1, parent2):
    point1 = random.randint(1, len(parent1) - 2)
    point2 = random.randint(point1 + 1, len(parent1) - 1)
    child1 = parent1[:point1] + parent2[point1:point2] + parent1[point2:]
    child2 = parent2[:point1] + parent1[point1:point2] + parent2[point2:]
    return child1, child2


# Mutation: Flip a bit
def mutate(genome):
    return ".join(bit if random.random() > MUTATION_RATE else random.choice('01') for bit in genome)


# Main genetic algorithm
def genetic_algorithm():
    population = generate_population(POPULATION_SIZE)
```

```
    for generation in range(GENERATIONS):
        best_fitness = min(sphere(genome) for genome in population)
        print(f"Generation {generation}: Best fitness = {best_fitness}")

        new_population = []
        while len(new_population) < POPULATION_SIZE:
            parent1 = selection(population)
            parent2 = selection(population)
            child1, child2 = two_point_crossover(parent1, parent2)
            new_population.extend([mutate(child1), mutate(child2)])

        population = new_population

    # Best solution
    best_genome = min(population, key=sphere)
    print(f"Best genome: {best_genome}, Best fitness: {sphere(best_genome)}")

if __name__ == "__main__":
    genetic_algorithm()
```

OUTPUT:

```
Generation 0: Best fitness = 0.18342056837412085
Generation 1: Best fitness = 0.10326408919389689
Generation 2: Best fitness = 0.005974162588901006
Generation 3: Best fitness = 0.0024172188252786
Generation 4: Best fitness = 0.00031311124679774926
Generation 5: Best fitness = 1.2524449871909348e-05
Generation 6: Best fitness = 0.00021291564782247003
Generation 7: Best fitness = 6.262224935954675e-05
Generation 8: Best fitness = 6.262224935954675e-05
Generation 9: Best fitness = 1.2524449871909348e-05
Generation 10: Best fitness = 1.2524449871909348e-05
Generation 11: Best fitness = 1.2524449871909348e-05
Generation 12: Best fitness = 1.2524449871909348e-05
Generation 13: Best fitness = 1.2524449871909348e-05
```

Name: Parth Das
Sap: 60004220185
Roll No:- C-111
AI Experiment 7:- Perceptron Learning

Parth Das
60004220185 C-22

C-111

AI Assignment-7

Aim :- To implement Perceptron Learning

Theory:-
The Perceptron is one of the simplest types of artificial neural networks used for binary classification problems. It was first introduced by Frank Rosenblatt in 1958 and forms the foundation for more complex neural networks. The perceptron learning algorithm is an iterative process to update the weights of a linear classifier.

Key concepts:-

- Inputs: Feature values (e.g. $x_1, x_2, \ldots x_n$)
- Weights: coefficients associated with the inputs, initialized randomly (e.g. $w_1, w_2, \ldots w_n$)
- Bias:- An additional parameter (b) that allows the decision boundary to be shifted.
- Activation Function:- A step function that outputs either 1 or -1 (binary output) based on whether the weighted sum of the inputs exceeds a threshold.

The decision rule is,

$$y = \text{sign}(w_1x_1 + w_2x_2 + \ldots + w_nx_n + b)$$

where output y is either 1 or -1 depending on classification boundary

Algorithm:-

1. Initialization:- Start with small random weights $w_1, w_2 \ldots w_n$ and bias b.
2. Prediction:- For each input sample, calculate weighted sum and apply the activation function to classify the sample.

FOR EDUCATIONAL USE

3. Update Weights:
- If the prediction matches the actual label, do nothing
- If the prediction is incorrect, adjust the weights and bias

$w_i = w_i + \eta * (y - \hat{y}) * x_i$

$b = b + \eta * (y - \hat{y})$

where η is the learning rate, y is actual label, $\hat{y}$ is predicted label.

4. Repeat the process for a specified number of iterations or until convergence.

Features:-

-> completeness:- Only complete if dataset is linearly separable. In other words, it can find a solution if there exists a linear boundary that separates the data into two classes

-> optimality:-
This algorithm doesn't guarantee an optimal solution in terms of minimizing classification errors.

-> Time complexity:-
$O(T * n * d)$ where T: no. of iterations
n: number of training samples
d: number of features per sample.

-> Space complexity:-
$O(n * d)$ if weights $O(d)$ and Bias $(O(1))$ with input data $O(n * d)$

Conclusion:-
This algorithm is foundation in ML, particularly for binary linear classification problems. It efficiently finds a decision boundary when the data is linearly separable, though it does not guarantee optimality in terms of maximizing the margin.

21/10/24

Code:-

```python
import numpy as np
import pandas as pd
class PerceptronNetwork:
    def __init__(self, input_size):
        self.weights = np.random.randn(input_size)
        self.bias = np.random.rand(1)
    def predict(self, x):
        weighted_sum = np.dot(self.weights, x) + self.bias
        return 1 if weighted_sum >= 0 else 0
    def train(self, X, y, learning_rate, epochs):
        results = []
        for epoch in range(epochs):
            for i in range(len(X)):
                prediction = self.predict(X[i])
                error = y[i] - prediction
                self.weights += (learning_rate * error * X[i])
                self.bias += (learning_rate * error)
                # Collecting results for this epoch
                results.append({
                    "Sample": i,
                    "Input": X[i],
                    "Weights": self.weights.copy(),
                    "Epoch": epoch + 1,
                    "Bias": self.bias[0],
                    "Prediction": prediction,
                    "Error": error,
                    "Learning Rate": learning_rate
                })
        # Convert results to a DataFrame for better visualization
        df_results = pd.DataFrame(results)
        print(df_results)
# Create a perceptron network and train it
perceptron_network = PerceptronNetwork(input_size=2)
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 1, 1, 1])
# Training with 1 epoch for initial results
perceptron_network.train(X, y, 0.1, 1)
# Predictions after initial training
for x in X:
    prediction = perceptron_network.predict(x)
    print(f"Input: {x}, Perceptron Prediction: {prediction}")
# Further training for 10 epochs
perceptron_network.train(X, y, 0.1, 10)
# Final predictions after training
for x in X:
    prediction = perceptron_network.predict(x)
    print(f"Input: {x}, Perceptron Prediction: {prediction}")
```

Output:-

```
   Sample   Input                                      Weights  Epoch      Bias  Prediction  Error  Learning Rate
0       0  [0, 0]  [0.2104908952127543, -0.32273565879466526]      1  0.586074           1     -1            0.1
1       1  [0, 1]  [0.2104908952127543, -0.32273565879466526]      1  0.586074           1      0            0.1
2       2  [1, 0]  [0.2104908952127543, -0.32273565879466526]      1  0.586074           1      0            0.1
3       3  [1, 1]  [0.2104908952127543, -0.32273565879466526]      1  0.586074           1      0            0.1
Input: [0 0], Perceptron Prediction: 1
Input: [0 1], Perceptron Prediction: 1
Input: [1 0], Perceptron Prediction: 1
Input: [1 1], Perceptron Prediction: 1
    Sample   Input                                      Weights  Epoch      Bias  Prediction  Error  Learning Rate
0        0  [0, 0]   [0.2104908952127543, -0.32273565879466526]      1  0.486074           1     -1            0.1
1        1  [0, 1]   [0.2104908952127543, -0.32273565879466526]      1  0.486074           1      0            0.1
2        2  [1, 0]   [0.2104908952127543, -0.32273565879466526]      1  0.486074           1      0            0.1
3        3  [1, 1]   [0.2104908952127543, -0.32273565879466526]      1  0.486074           1      0            0.1
4        0  [0, 0]   [0.2104908952127543, -0.32273565879466526]      2  0.386074           1     -1            0.1
5        1  [0, 1]   [0.2104908952127543, -0.32273565879466526]      2  0.386074           1      0            0.1
6        2  [1, 0]   [0.2104908952127543, -0.32273565879466526]      2  0.386074           1      0            0.1
7        3  [1, 1]   [0.2104908952127543, -0.32273565879466526]      2  0.386074           1      0            0.1
8        0  [0, 0]   [0.2104908952127543, -0.32273565879466526]      3  0.286074           1     -1            0.1
9        1  [0, 1]   [0.2104908952127543, -0.22273565879466525]      3  0.386074           0      1            0.1
10       2  [1, 0]   [0.2104908952127543, -0.22273565879466525]      3  0.386074           1      0            0.1
11       3  [1, 1]   [0.2104908952127543, -0.22273565879466525]      3  0.386074           1      0            0.1
12       0  [0, 0]   [0.2104908952127543, -0.22273565879466525]      4  0.286074           1     -1            0.1
13       1  [0, 1]   [0.2104908952127543, -0.22273565879466525]      4  0.286074           1      0            0.1
14       2  [1, 0]   [0.2104908952127543, -0.22273565879466525]      4  0.286074           1      0            0.1
15       3  [1, 1]   [0.2104908952127543, -0.22273565879466525]      4  0.286074           1      0            0.1
16       0  [0, 0]   [0.2104908952127543, -0.22273565879466525]      5  0.186074           1     -1            0.1
17       1  [0, 1]   [0.2104908952127543, -0.12273565879466525]      5  0.286074           0      1            0.1
18       2  [1, 0]   [0.2104908952127543, -0.12273565879466525]      5  0.286074           1      0            0.1
19       3  [1, 1]   [0.2104908952127543, -0.12273565879466525]      5  0.286074           1      0            0.1
20       0  [0, 0]   [0.2104908952127543, -0.12273565879466525]      6  0.186074           1     -1            0.1
21       1  [0, 1]   [0.2104908952127543, -0.12273565879466525]      6  0.186074           1      0            0.1
22       2  [1, 0]   [0.2104908952127543, -0.12273565879466525]      6  0.186074           1      0            0.1
23       3  [1, 1]   [0.2104908952127543, -0.12273565879466525]      6  0.186074           1      0            0.1
24       0  [0, 0]   [0.2104908952127543, -0.12273565879466525]      7  0.086074           1     -1            0.1
25       1  [0, 1]  [0.2104908952127543, -0.022735658794665242]      7  0.186074           0      1            0.1
26       2  [1, 0]  [0.2104908952127543, -0.022735658794665242]      7  0.186074           1      0            0.1
27       3  [1, 1]  [0.2104908952127543, -0.022735658794665242]      7  0.186074           1      0            0.1
28       0  [0, 0]  [0.2104908952127543, -0.022735658794665242]      8  0.086074           1     -1            0.1
29       1  [0, 1]  [0.2104908952127543, -0.022735658794665242]      8  0.086074           1      0            0.1
30       2  [1, 0]  [0.2104908952127543, -0.022735658794665242]      8  0.086074           1      0            0.1
31       3  [1, 1]  [0.2104908952127543, -0.022735658794665242]      8  0.086074           1      0            0.1
32       0  [0, 0]  [0.2104908952127543, -0.022735658794665242]      9 -0.013926           1     -1            0.1
33       1  [0, 1]    [0.2104908952127543, 0.07726434120533476]      9  0.086074           0      1            0.1
34       2  [1, 0]    [0.2104908952127543, 0.07726434120533476]      9  0.086074           1      0            0.1
35       3  [1, 1]    [0.2104908952127543, 0.07726434120533476]      9  0.086074           1      0            0.1
36       0  [0, 0]    [0.2104908952127543, 0.07726434120533476]     10 -0.013926           1     -1            0.1
37       1  [0, 1]    [0.2104908952127543, 0.07726434120533476]     10 -0.013926           1      0            0.1
38       2  [1, 0]    [0.2104908952127543, 0.07726434120533476]     10 -0.013926           1      0            0.1
39       3  [1, 1]    [0.2104908952127543, 0.07726434120533476]     10 -0.013926           1      0            0.1
Input: [0 0], Perceptron Prediction: 0
Input: [0 1], Perceptron Prediction: 1
Input: [1 0], Perceptron Prediction: 1
Input: [1 1], Perceptron Prediction: 1
```

Name: Parth Das
SAP:- 60004220185
Roll No: C-111
AI EXperiment 8 Prolog Tree

Parth Das
60004220185 C-2/2

C-111

AI Experiment 8

Aim:- Prolog Tree Implementation

Theory:-

Knowledge-Based Learning:
It refers to the process where a system makes decisions or solves problems by using explicitly encoded knowledge and logical reasoning. It involves the use of rules, facts and relationships to derive conclusions or make inferences.
Knowledge-based systems rely on artificial intelligence techniques to represent and manipulate this knowledge. Knowledge-Based Learning typically includes:
- Facts - Basic statements about the world, representing truths
- Rules:- Logical statements that define relationships between facts
- Inference Engine:- Uses the rules and facts to deduce new information or make decisions.

Prolog and its Role in Knowledge-Based Learning
Prolog is a logic programming language that is widely used for creating knowledge-based systems. It stands out due to its declarative nature, where you specify what needs to be done rather than how to do it. In Prolog, the focus is on defining facts & rules, and the Prolog interpreter uses these to infer answers through logical reasoning.
- Key concepts

-> Facts:- Represents known truths eg parent(john, mary).
means John is parent of Mary.
-> Rules:- Define relationships and how facts are connected.
eg grandparent(X,Y) :- parent(X,Z), parent(Z,Y), states

that X is a grandparent of Y if X is parent of Z and Z is parent of Y.

-> **Queries:-** Allow users to ask questions based on the facts and rules and Prolog will attempt to provide answers.

e.g. ?- grandparent (john, mary).

· **Prolog Tree (search Tree)**

A Prolog Tree is a visualization of the logic that Prolog uses to solve queries. It represents the step-by-step process by which Prolog tries to satisfy a query by traversing possible paths of facts and rules. This search process is typically conducted using backtracking.

How Prolog Tree works:-

1. **Query Initiation:-** When a query is posed, Prolog begins to search for a solution starting from the root of the tree.
2. **Goal matching:-** It matches the goal (query) against the facts and rules defined in the knowledge base.
3. **Branching:-** The tree branches out at each step as Prolog tries different rules or facts that might satisfy the current goal.
4. **Backtracking:-** If a branch leads to a dead end (failure to satisfy the goal), Prolog backtracks to the previous level and tries another branch.
5. **Solution Path:-** When Prolog finds a path that satisfies the goal, it returns the solution.

Properties:-

Optimality:- Not optimal by default uses depth first search with backtracking finds the first solution without necessarily being the best one.

Completeness:-

Complete if the search space is finite and properly structured. Can get stuck in infinite loops

Time Complexity:- Exponential in worst case $O(b^d)$ Highly sensitive to size of the search b: branching factor
d: depth of search tree.

Space complexity:- Linear in depth of search tree $O(d)$.
more memory efficient than BFS.

Optimization:- To improve optimality in Prolog, you can add constraints or use strategies like best first search or heuristics, which are not part of standard Prolog but can be implemented.

Conclusion:-

While Prolog is a powerful for knowledge-based systems due to its logical reasoning capabilities, its default search method (depth first with backtracking) may not always be optimal or efficient. Understanding the limitations in terms of optimality, completeness, and the computational cost helps in designing more efficient logic programs or choosing alternatives search strategies for specific problem domains.

21/10/24

s

Relationships:-

```
% Male members
male(john).
male(mike).
male(alex).
male(tom).
male(jake).
male(eric).

% Female members
female(susan).
female(linda).
female(kate).
female(anna).
female(sophia).

% Defining parent relationships
parent(john, mike).     % John is the parent of Mike
parent(john, linda).    % John is the parent of Linda
parent(susan, mike).    % Susan is the parent of Mike
parent(susan, linda).   % Susan is the parent of Linda

parent(mike, alex).     % Mike is the parent of Alex
parent(mike, kate).     % Mike is the parent of Kate
parent(linda, tom).     % Linda is the parent of Tom
parent(linda, sophia).  % Linda is the parent of Sophia

parent(alex, jake).     % Alex is the parent of Jake
parent(kate, anna).     % Kate is the parent of Anna
```

Rules:-

```prolog
% Grandfather and Grandmother
grandfather(X, Y) :- male(X), parent(X, Z), parent(Z, Y).
grandmother(X, Y) :- female(X), parent(X, Z), parent(Z, Y).

% Father and Mother
father(X, Y) :- male(X), parent(X, Y).
mother(X, Y) :- female(X), parent(X, Y).

% Son and Daughter
son(X, Y) :- male(X), parent(Y, X).
daughter(X, Y) :- female(X), parent(Y, X).

% Uncle and Aunt
uncle(X, Y) :- male(X), parent(Z, Y), sibling(X, Z).
aunt(X, Y) :- female(X), parent(Z, Y), sibling(X, Z).

% Sibling (brother or sister)
sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.

% Cousin (cousin brother or cousin sister)
cousin(X, Y) :- parent(Z, X), parent(W, Y), sibling(Z, W).
```

**Queries:-**

1. Who is John's grandson?

```
grandfather(john, X), male(X)
X = alex
X = tom
```

2. Who are the siblings of Kate?

```
sibling(kate, X).
X = alex
```

3. Who is the cousin brother of Sophia?

```
cousin(sophia, X),male(X).
X = alex
```

4. Who is the cousin sister of Sophia?

```
cousin(sophia, X),female(X).
X = kate
```

5. Who is the uncle of Tom?

```
uncle(X, tom).
X = mike
```

6. Who is Anna's grandfather?

```
grandfather(X, anna).
X = mike
```

7. Who are the children of Mike?

```
parent(mike, X).
X = alex
X = kate
```

8. Is Jake a son of Alex?

```
son(jake, alex).
true
```

9. Who is the aunt of Tom?

```
aunt(X, tom).
false
```

10. Is Linda the grandmother of Anna?

```
grandmother(linda, anna).
false
```

Names: **SapId:**

PARTH DAS **60004220185**

Subject: AI
Batch: C2-2

EXPERIMENT NO. 9

Topic: Game Design

GAME TITLE:

PacMan Lite

IDEA/DESCRIPTION:

The PacMan Lite game is a simplified version of the classic PacMan game. The player controls a PacMan character and navigates through a maze, collecting pellets while avoiding ghosts. The objective is to collect all the pellets in the maze before the player's lives are depleted.

The core elements of the game include:

Maze Generation: The game features a pre-defined maze layout, with walls, pellets, and spawn locations for the PacMan character and ghosts.

Player Control: The player can control the PacMan character using the arrow keys, moving it in four directions (up, down, left, right) to navigate the maze.

Pellet Control:- The player must collect all the pellets scattered throughout the maze to complete the level.

Ghost Avoidance:- The player must avoid colliding with the ghosts, which will cause the player to lose a life. The ghosts move autonomously using a simple pathfinding algorithm.

RULES:

1. The player starts with a set number of lives (e.g., 3).

2. The player can move the PacMan character in four directions (up, down, left, right) using the arrow keys.

3. The player must collect all the pellets in the maze to complete the level.

4. If the PacMan character collides with a ghost, the player loses a life.

5. If the player runs out of lives, the game is over.

6. The player can restart the game by clicking the "Play Again" button.

TASK ENVIRONMENT:

Property	Value
Observable	Partially
Deterministic	Deterministic
Episodic	Sequential
Static	Dynamic
Discrete	Discrete
Agents	Single

Observable:The environment is **partially observable** because the player can only see the immediate area around the PacMan character, and the positions of the ghosts are only revealed when they are in the player's field of view.

Deterministic:: The environment is **deterministic** because the maze layout, pellet positions, and ghost movements are predetermined and do not change during the game.

Episodic: The task is **sequential** because the player's actions (moving the PacMan character) directly affect the game state and the outcome of the current episode (level).

Static: The environment is **dynamic** during the game, as the PacMan character and ghosts move and the pellets are collected, changing the state of the game.

Discrete:: The environment is **discrete**, as the maze is divided into a grid of cells, and the PacMan character and ghosts move between these cells.

Agents: : There is a **single agent** (the player controlling the PacMan character) in the game

Algorithms

1. **<u>Dijkstra's Algorithm:-</u>**: This finds the shortest path from each enemy to the closest resource when the enemy's health is low. By minimizing the distance, enemies can reach resources quickly to "heal."

2.**<u>Heuristic Function:</u>** The heuristic function (Manhattan distance) provides a baseline for calculating distances.

CONCLUSION

The PacMan Lite game presents a partially observable, deterministic, and sequential environment where a single agent (the player) navigates a discrete grid-based maze, collecting pellets while avoiding autonomous ghosts. The game showcases the implementation of pathfinding algorithms for the ghosts and collision detection between game entities. The simplified nature of the game allows for an interactive and accessible experience in understanding classic arcade-style game mechanics.

Name: Parth Das
Sap: 60004220185
Batch: C-22
Roll no: C-111
Subject : Artificial Intelligence

Experiment 10:- Case Study of an AI Application

Sr. No	Paper Title	Publication Name (Year)	Algorithm/Technique Used	Dataset (If Any)	Results Mentioned in Paper	Observations
1	DetectGPT: Zero-Shot Machine-Generated Text Detection using Probability Curvature	arXiv (2023)	Probability Curvature	None specified	DetectGPT distinguishes human from AI text with significant accuracy	This paper emphasizes transparency in AI content generation detection and has potential applications in education and journalism
2	HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face	arXiv (2023)	Integration of ChatGPT with Hugging Face models	None specified	Shows improved efficiency in performing AI tasks by leveraging existing models	Demonstrates the versatility of combining language models with other AI tools for diverse tasks

3	**Tree of Thoughts: Deliberate Problem Solving with Large Language Models**	**arXiv (2023)**	**Large Language Models with reasoning**	**None specified**	**LLMs, when combined with deliberate problem-solving strategies, enhance performance in reasoning tasks**	**Highlights the importance of structured thought processes in AI-driven problem solving**
4	**Toolformer: Language Models Can Teach Themselves to Use Tools**	**arXiv (2023)**	**Toolformer framework**	**None specified**	**Language models demonstrated self-supervised learning to use tools independently**	**This research pushes the boundaries of self-learning models, enabling them to autonomously acquire and use external tools**
5	**A Watermark for Large Language Models**	**arXiv (2023)**	**Watermarking for LLM-generated content**	**None specified**	**Proposed a technique to trace and verify AI-generated content**	**Vital for AI content authenticity, particularly in legal, educational, and journalistic sectors**

6	**Is ChatGPT a General-Purpose Natural Language Processing Task Solver?**	**arXiv (2023)**	**ChatGPT performance on diverse NLP tasks**	**None specified**	**ChatGPT performs well across many NLP tasks but struggles with domain-specific tasks**	**Shows that while ChatGPT is versatile, domain-specific tuning is necessary for high precision tasks**