Sri Sukhmani Institute of Engineering & Technology

Derabassi (140507)

Department Of Computer Science Engineering

Artificial Intelligence Lab

BTCS 605 - 18

Submitted by - **Aditya Mishra**

Submitted to - **Mr. Pawan**

University Roll Number - **2206979**

INDEX

Practical No. 01

Aim: Introduction to Artificial Intelligence and its Applications

Artificial Intelligence (AI)

- **Definition:** AI is the science of creating intelligent machines or programs.
- **Goal:** To make machines think and act like humans.
- **Main Areas:** Reasoning, learning, problem-solving, perception, and language understanding.

Objectives of AI

- Enable machines to reason and make decisions.
- Represent knowledge to process information.
- Plan and execute tasks.
- Learn from data and experiences.
- Understand and use natural language.
- Move and interact with the environment.

Approaches in AI

- Uses statistics, logic, and algorithms to solve problems.
- Applies neural networks and optimization techniques.
- Combines rules and data-driven learning.

Goals of AI

- Build expert systems that assist and guide users.
- Develop machines that mimic human thinking and behavior.

Foundations of AI

- Based on computer science, math, psychology, linguistics, and philosophy.

Applications of AI

- **Gaming:** Helps machines make smart moves in games like chess.
- **NLP:** Allows computers to understand human language.
- **Expert Systems:** Offer expert-level decisions and advice.
- **Vision Systems:** Interpret images and visuals like a human eye.
- **Speech Recognition:** Understands spoken words and converts them into text.
- **Handwriting Recognition:** Converts handwritten text into digital format.
- **Intelligent Robots:** Robots follow and perform human commands.

Practical No. 02

AIM: Implementation of Depth-First Search (Uninformed Search Strategy)

Depth-First Search (DFS) - Theory (Short & Point-wise)

- DFS is an uninformed search that explores as far as possible along a branch before backtracking.

- It uses **Stack (LIFO)** data structure for traversal.

- DFS explores deep nodes first, which helps in reducing memory usage compared to BFS.

- **Advantages:**
  - Requires less memory.
  - May reach the goal faster if it's in a deep path.
  - Stop when the first solution is found.

- **Disadvantages:**
  - May enter infinite loops if cycles exist.
  - No guarantee to find the shortest or any solution.

DFS Algorithm Steps:

1. Push the starting node onto the stack.
2. While the stack is not empty:
   - Pop the top node, visit it.
   - Push all unvisited neighbors onto the stack.
3. Repeat until the goal is found or all nodes are visited.

// **Program to implement DFS**

```
#include <iostream>
using namespace std;

int cost[10][10], visited[10], n;

void DFS(int v) {
   cout << v << " ";
   visited[v] = 1;
   for (int i = 1; i <= n; i++) {
      if (cost[v][i] == 1 && !visited[i]) {
         DFS(i);
      }
   }
}

int main() {
   int m, i, j, v;
   cout << "Enter number of vertices: ";
   cin >> n;
   cout << "Enter number of edges: ";
   cin >> m;

   cout << "Enter edges (format: from to):\n";
   for (int k = 1; k <= m; k++) {
      cin >> i >> j;
      cost[i][j] = 1;
      cost[j][i] = 1; // For undirected graph
   }

   cout << "Enter starting vertex: ";
   cin >> v;

   cout << "Order of Visited vertices:\n";
   DFS(v);

   return 0;
}
```

```
Enter number of vertices: 5
Enter number of edges: 6
Enter edges (format: from to):
1 2
1 3
2 4
2 5
3 5
4 5
Enter starting vertex: 1
Order of Visited vertices:
1 2 4 5 3
```

Practical No. 03

AIM: Implementation of Breadth-First Search (Uninformed Search Strategy)

Breadth-First Search (BFS) – Theory (Short & Point-wise)

- **BFS** is a graph traversal algorithm that explores neighbors level by level.
- **Data Structure Used:** Queue (FIFO).
- **Starts** at the root or any given starting node.

Advantages of BFS

- Finds the shortest path between nodes.
- Always gives the optimal solution.
- No unnecessary paths, as it expands level by level.
- Find the closest goal quickly in many cases.

Disadvantages of BFS

- Requires more memory to store all nodes at the current level.
- Slower if the graph is very wide (many branches).

BFS Algorithm Steps

1. Mark all nodes as **ready (STATUS = 1)**.
2. Enqueue the starting node, mark it **waiting (STATUS = 2)**.
3. While queue is not empty:
    - Dequeue node, mark it **processed (STATUS = 3)** and print it.
    - Enqueue all unvisited neighbors and mark them **waiting (STATUS = 2)**.

// Program to implement BFS

```
#include <iostream>
using namespace std;

int cost[10][10], visited[10], n;

void BFS(int start) {
   int queue[10], front = 0, rear = 0;
   visited[start] = 1;
   queue[rear++] = start;

   while (front < rear) {
      int v = queue[front++];
      cout << v << " ";

      for (int i = 1; i <= n; i++) {
         if (cost[v][i] == 1 && !visited[i]) {
            queue[rear++] = i;
            visited[i] = 1;
         }
      }
   }
}

int main() {
   int m, i, j, start;
   cout << "Enter number of vertices: ";
   cin >> n;
   cout << "Enter number of edges: ";
   cin >> m;

   cout << "Enter edges (format: from to):\n";
   for (int k = 1; k <= m; k++) {
      cin >> i >> j;
      cost[i][j] = 1;
      cost[j][i] = 1; // For undirected graph
   }

   cout << "Enter starting vertex: ";
   cin >> start;

   cout << "Visited vertices in BFS order:\n";
   BFS(start);
   return 0;
}
```

```
Enter number of vertices: 5
Enter number of edges: 6
Enter edges (format: from to):
1 2
1 3
2 4
2 5
3 5
4 5
Enter starting vertex: 1
Visited vertices in BFS order:
1 2 3 4 5
```

Practical No. 04

AIM: Implementation of A* Algorithm (Informed Search Strategy)

Introduction:

The A* (A-star) Search algorithm is a popular and efficient pathfinding and graph traversal algorithm. It is widely used in applications like AI, robotics, and game development for finding the optimal path from a starting point to a destination while avoiding obstacles.

The algorithm works by evaluating nodes based on two values:

- **g(n):** The actual cost from the start node to the current node.
- **h(n):** The estimated cost from the current node to the goal (using a heuristic).
- **f(n):** The total cost of a node, calculated as:
  f(n) = g(n) + h(n)

The A* algorithm guarantees finding the optimal path when the heuristic function is admissible.

Objectives:

- To understand the working of the A* algorithm.
- To implement the algorithm for pathfinding in a grid-based system.
- To demonstrate the functionality of A* using a sample grid.

Theory:

1. *A Search Algorithm:**
   - The algorithm maintains two lists:
     - **Open List**: Contains nodes that are being evaluated.
     - **Closed List**: Contains nodes that have already been evaluated.
   - The A* algorithm uses a combination of actual and estimated costs to choose which node to expand next, ensuring an optimal and efficient pathfinding solution.
2. **Formula:**

- **f(n) = g(n) + h(n)**
  - **g(n)**: The cost to reach node n from the start node.
  - **h(n)**: The estimated cost from node n to the goal.
  - **f(n)**: The total cost to reach node n.

3. **Heuristic Function (h(n)):**

   - The heuristic function helps estimate the cost to the goal, commonly calculated using the Euclidean or Manhattan distance.

Algorithm:

1. **Initialize**: Create two lists: Open list (with the starting node) and Closed list (empty). Initialize the cost values of all nodes.
2. **Fail**: If Open list is empty, terminate the search as the goal is unreachable.
3. **Select**: Choose the node with the minimum f(n) from the Open list, and move it to the Closed list.
4. **Expand**: For each valid neighbor of the selected node, calculate f(n), g(n), and h(n) and add the node to the Open list if not already in the Closed list.
5. **Terminate**: If the destination node is reached, terminate and return the path.
6. **Repeat**: Continue expanding nodes until the destination is found or no solution exists.

Procedure:

1. **Set up the grid**: Create a grid (2D array) representing the environment. Some cells represent obstacles (blocked) and others represent open paths (unblocked).
2. **Define Source and Destination**: Select the starting point and destination on the grid.
3. **Implementation**: Implement the A* algorithm using the following code (as provided in the problem statement).
4. **Run the Program**: Execute the program with the selected grid and verify if the algorithm successfully finds the shortest path from the source to the destination.

// Program to implement A* Algorithm

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

#define ROW 9
#define COL 10
typedef pair<int, int> Pair;
typedef pair<double, pair<int, int>> pPair;

struct cell {
   int parent_i, parent_j;
   double f, g, h;
};

bool isValid(int row, int col) {
   return (row >= 0) && (row < ROW) && (col >= 0) && (col < COL);
}

bool isUnBlocked(int grid[][COL], int row, int col) {
   return (grid[row][col] == 1);
}

bool isDestination(int row, int col, Pair dest) {
   return (row == dest.first && col == dest.second);
}

double calculateHValue(int row, int col, Pair dest) {
   return (sqrt((row - dest.first) * (row - dest.first) +
            (col - dest.second) * (col - dest.second)));
}

void tracePath(cell cellDetails[][COL], Pair dest) {
   stack<Pair> Path;
   int row = dest.first, col = dest.second;
   while (!(cellDetails[row][col].parent_i == row &&
         cellDetails[row][col].parent_j == col)) {
      Path.push(make_pair(row, col));
      int temp_row = cellDetails[row][col].parent_i;
      int temp_col = cellDetails[row][col].parent_j;
      row = temp_row;
      col = temp_col;
   }
   Path.push(make_pair(row, col));
   while (!Path.empty()) {
      pair<int, int> p = Path.top();
      Path.pop();
      cout << "-> (" << p.first << "," << p.second << ") ";
   }
}

void aStarSearch(int grid[][COL], Pair src, Pair dest) {
   if (!isValid(src.first, src.second) || !isValid(dest.first, dest.second)) {
```

```
    cout << "Invalid Source or Destination" << endl;
    return;
}
if (!isUnBlocked(grid, src.first, src.second) || !isUnBlocked(grid, dest.first, dest.second)) {
    cout << "Source or Destination is blocked" << endl;
    return;
}
bool closedList[ROW][COL];
memset(closedList, false, sizeof(closedList));

cell cellDetails[ROW][COL];
for (int i = 0; i < ROW; i++) {
    for (int j = 0; j < COL; j++) {
        cellDetails[i][j].f = FLT_MAX;
        cellDetails[i][j].g = FLT_MAX;
        cellDetails[i][j].h = FLT_MAX;
        cellDetails[i][j].parent_i = -1;
        cellDetails[i][j].parent_j = -1;
    }
}

int i = src.first, j = src.second;
cellDetails[i][j].f = 0.0;
cellDetails[i][j].g = 0.0;
cellDetails[i][j].h = 0.0;
cellDetails[i][j].parent_i = i;
cellDetails[i][j].parent_j = j;

set<pPair> openList;
openList.insert(make_pair(0.0, make_pair(i, j)));

bool foundDest = false;

while (!openList.empty()) {
    pPair p = *openList.begin();
    openList.erase(openList.begin());

    i = p.second.first;
    j = p.second.second;
    closedList[i][j] = true;

    double gNew, hNew, fNew;
    // Check all 8 possible directions (up, down, left, right, diagonals)
    for (int di = -1; di <= 1; di++) {
        for (int dj = -1; dj <= 1; dj++) {
            int newRow = i + di;
            int newCol = j + dj;
            if (isValid(newRow, newCol)) {
                if (isDestination(newRow, newCol, dest)) {
                    cellDetails[newRow][newCol].parent_i = i;
                    cellDetails[newRow][newCol].parent_j = j;
                    cout << "Destination found!" << endl;
```

```
                tracePath(cellDetails, dest);
                foundDest = true;
                return;
            }
            if (!closedList[newRow][newCol] && isUnBlocked(grid, newRow, newCol)) {
                gNew = cellDetails[i][j].g + 1.0;
                hNew = calculateHValue(newRow, newCol, dest);
                fNew = gNew + hNew;
                if (cellDetails[newRow][newCol].f == FLT_MAX || cellDetails[newRow][newCol].f > fNew) {
                    openList.insert(make_pair(fNew, make_pair(newRow, newCol)));
                    cellDetails[newRow][newCol].f = fNew;
                    cellDetails[newRow][newCol].g = gNew;
                    cellDetails[newRow][newCol].h = hNew;
                    cellDetails[newRow][newCol].parent_i = i;
                    cellDetails[newRow][newCol].parent_j = j;
                }
            }
        }
    }
  }
  if (!foundDest) {
    cout << "Failed to find the destination cell." << endl;
  }
}
int main() {
  int grid[ROW][COL] = { {1, 0, 1, 1, 1, 1, 0, 1, 1, 1},
                         {1, 1, 1, 0, 1, 1, 0, 1, 1, 1},
                         {1, 0, 1, 0, 1, 1, 1, 0, 0, 1},
                         {1, 0, 1, 0, 1, 1, 1, 1, 1, 1},
                         {1, 1, 1, 1, 1, 0, 0, 0, 0, 1},
                         {1, 0, 1, 0, 1, 1, 1, 1, 0, 1},
                         {1, 1, 0, 0, 0, 1, 0, 1, 1, 1},
                         {1, 0, 1, 1, 0, 0, 1, 0, 1, 1},
                         {1, 1, 1, 1, 0, 1, 1, 1, 0, 1} };

  Pair src = make_pair(0, 0); // Starting point
  Pair dest = make_pair(8, 9); // Destination
  aStarSearch(grid, src, dest);
  return 0;
}
```

OUTPUT :

```
Destination found!
-> (0,0) -> (1,0) -> (2,0) -> (3,0) -> (4,0) -> (5,0) -> (6,0) -> (7,0) -> (8,0) -> (8,1) ->

(8,2) -> (8,3) -> (8,4) -> (8,5) -> (8,6) -> (8,7) -> (8,8) -> (8,9)
```

Practical No. 05

AIM: Write a Program to implement the Tower of Hanoi.

Introduction:

The Tower of Hanoi is a classic puzzle that involves moving disks from one pole to another, subject to certain constraints. The puzzle involves three poles and a number of disks of different sizes. The goal is to move all the disks from the source pole to the destination pole while following these rules:

- Only one disk can be moved at a time.
- A disk can only be placed on top of a larger disk or an empty pole.
- You can use an auxiliary pole to help with the movement.

Algorithm:

1. If there is only one disk, move it directly from the source pole to the destination pole.
2. For **n disks**:
    - Move the top **n-1 disks** from the source pole to the auxiliary pole using the destination pole as a temporary storage.
    - Move the **nth disk** (the largest disk) directly to the destination pole.
    - Move the **n-1 disks** from the auxiliary pole to the destination pole using the source pole as temporary storage.

Explanation:

The recursive approach helps break down the problem of moving **n disks** into smaller subproblems until the base case (moving one disk) is reached.

// Code to implement the Tower of Hanoi.

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

void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
   // Base condition: if no disks are left to move, return
   if (n == 0)
   {
      return;
   }

   // Step 1: Move n-1 disks from source to auxiliary rod
   towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);

   // Step 2: Move the nth disk from source to destination rod
   cout << "Move disk " << n << " from rod " << from_rod << " to rod " << to_rod <<
endl;

   // Step 3: Move the n-1 disks from auxiliary rod to destination rod
   towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}
int main()
{
   int n = 4; // Number of disks

   // Calling towerOfHanoi function with
source, destination, and auxiliary rods

   towerOfHanoi(n, 'A', 'C', 'B');
   return 0;
}
```

```
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
Move disk 4 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 2 from rod C to rod A
Move disk 1 from rod B to rod A
Move disk 3 from rod C to rod B
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
```

Practical No. 06

Aim: Implementation of the Tic-Tac-Toe game.

Introduction:

Tic-Tac-Toe is a two-player game, where one player uses 'X' and the other uses 'O'. Players take turns marking a cell in a 3x3 grid, and the goal is to get three of their marks in a row, either horizontally, vertically, or diagonally. If neither player achieves this and the grid is full, the game ends in a draw.

In this program, the game is played between the HUMAN and the COMPUTER. The players make moves alternately, with the computer randomly choosing its moves. The program checks for a winner after each move.

Game Rules:

- The game is played on a 3x3 grid.
- Players alternate turns, one choosing 'X' and the other 'O'.
- A player wins if they fill a row, column, or diagonal with their symbol.
- The game ends in a draw if the grid is filled without a winner.

Program Logic:

1. Marking the grid: The grid consists of 9 cells (0-8). Each cell is marked by either 'X', 'O', or remains empty.
2. Check for a winner: After each move, the program checks if either player has won the game by filling a row, column, or diagonal.
3. Random move for the computer: The computer's move is chosen randomly using the `rand()` function.

Program:

```
#include <iostream>
using namespace std;

char square[9] = {'0','1','2','3','4','5','6','7','8'};

int checkwin()
{
   // Check rows, columns, and diagonals
   if (square[0] == square[1] && square[1] == square[2]) {
      if (square[0] == 'X') return 1;
      else return 2;
   }
   else if (square[3] == square[4] && square[4] == square[5]) {
      if (square[3] == 'X') return 1;
```

```
        else return 2;
    }
    else if (square[6] == square[7] && square[7] == square[8]) {
        if (square[6] == 'X') return 1;
        else return 2;
    }
    else if (square[0] == square[3] && square[3] == square[6]) {
        if (square[0] == 'X') return 1;
        else return 2;
    }
    else if (square[1] == square[4] && square[4] == square[7]) {
        if (square[1] == 'X') return 1;
        else return 2;
    }
    else if (square[2] == square[5] && square[5] == square[8]) {
        if (square[2] == 'X') return 1;
        else return 2;
    }
    else if (square[0] == square[4] && square[4] == square[8]) {
        if (square[0] == 'X') return 1;
        else return 2;
    }
    else if (square[2] == square[4] && square[4] == square[6]) {
        if (square[2] == 'X') return 1;
        else return 2;
    }
    else return 0;
}

void mark(int player, int box) {
    if (player == 1) square[box] = 'X';
    else square[box] = 'O';
}

void display() {
    for (int i = 0; i < 9; i++) {
        cout << square[i] << "\t";
        if (i == 2 || i == 5 || i == 8) cout << "\n";
    }
}

int main() {
    int player1 = 1, player2 = 2; // player 1 is 'X', player 2 is 'O'
    int box, result = 0, flag = 0;
```

```
    for (int i = 1; i <= 5; i++) {
        // Player 1's turn
        cout << "\nPlayer 1 (X), Enter the Box (0-8): ";
        cin >> box;
        mark(player1, box);
        display();
        result = checkwin();
        if (result == 1) {
            cout << "\nCongratulations! Player 1 (X) has won!";
            flag = 1;
            break;
        }
        else if (result == 2) {
            cout << "\nCongratulations! Player 2 (O) has won!";
            flag = 1;
            break;
        }

        // Player 2's turn
        cout << "\nPlayer 2 (O), Enter the Box (0-8): ";
        cin >> box;
        mark(player2, box);
        display();
        result = checkwin();
        if (result == 1) {
            cout << "\nCongratulations! Player 1 (X) has won!";
            flag = 1;
            break;
        }
        else if (result == 2) {
            cout << "\nCongratulations! Player 2 (O) has won!";
            flag = 1;
            break;
        }
    }

    if (flag == 0)
        cout << "\nSorry, the game is a draw.";

    return 0;
}
```

```
Player 1 (X), Enter the Box (0-8): 0
1   2   3
4   5   6
7   8   9

Player 2 (O), Enter the Box (0-8): 4
X   2   3
4   O   6
7   8   9

Player 1 (X), Enter the Box (0-8): 1
X   X   3
4   O   6
7   8   9

Player 2 (O), Enter the Box (0-8): 7
X   X   3
4   O   6
O   8   9

Player 1 (X), Enter the Box (0-8): 2
Congratulations! Player 1 (X) has won!
```

Practical No. 07

Aim: Write a program to implement the Water Jug Problem using state-space search.

Statement:

We are given two jugs:

- One 4-liter jug (A)
- One 3-liter jug (B)

There are no measurement markings on the jugs. A pump is available to fill the jugs. The goal is to obtain exactly 2 liters of water in the 4-liter jug (Jug A).

Assumptions:

1. We can fill any jug completely from the pump.
2. We can empty any jug onto the ground.
3. We can pour water from one jug to another until one is either full or empty.
4. No partial measuring devices are available.
5. State space:
   `{ (i, j) | i = 0..4, j = 0..3 }`,
   where `i` is the amount in Jug A and `j` is the amount in Jug B.

Algorithm:

1. Start with the initial state (0, 0).
2. Repeat the following steps until Jug A has 2 liters:
   - If Jug B is empty, fill Jug B.
   - If Jug A is full, empty Jug A.
   - Pour water from Jug B to Jug A.
3. Print the intermediate states.
4. Stop when Jug A contains exactly 2 liters

.

Program:

```
#include <iostream>
using namespace std;

class Jug {
    int cap, val;
public:
    Jug(int c) : cap(c), val(0) {}
    void fill() { val = cap; }
    void empty() { val = 0; }
    bool isFull() { return val == cap; }
    bool isEmpty() { return val == 0; }
    void pourInto(Jug &o) {
        int d = min(val, o.cap - o.val);
        val -= d, o.val += d;
    }
    int get() { return val; }
};

int gcd(int a, int b) { return b ? gcd(b, a % b) : a; }

bool valid(int a, int b, int c) {
    return c <= a && c % gcd(a, b) == 0;
}
void solve(Jug &A, Jug &B, int goal) {
    while (A.get() != goal) {
        if (B.isEmpty()) B.fill(), cout << "Fill B\n";
        else if (A.isFull()) A.empty(), cout << "Empty A\n";
        else B.pourInto(A), cout << "Pour B into A\n";
        cout << "(A, B) = (" << A.get() << ", " << B.get() << ")\n";
    }
}
int main() {
    int a, b, c;
    cout << "Aditya Mishra\nEnter A, B: ";
    cin >> a >> b;
    do {
        cout << "Required in A: ";
        cin >> c;
    } while (!valid(a, b, c));
    Jug A(a), B(b);
    solve(A, B, c);
    return 0;
}
```

```
Aditya Mishra
Enter A, B: 4 3
Required in A: 2
Fill B
(A, B) = (0, 3)
Pour B into A
(A, B) = (3, 0)
Fill B
(A, B) = (3, 3)
Pour B into A
(A, B) = (4, 2)
Empty A
(A, B) = (0, 2)
Pour B into A
(A, B) = (2, 0)
```

Practical No: 08

AIM: Write a program to construct a Bayesian Network from given data.

Theory:

A Bayesian Network is a **probabilistic graphical model** that represents a set of variables and their conditional dependencies via a directed acyclic graph (DAG). It consists of two main components:

1. **Directed Acyclic Graph (DAG):**
   Each node represents a random variable, and each directed edge shows the probabilistic dependency between the variables.
2. **Conditional Probability Distribution (CPD):**
   Each node is associated with a CPD that quantifies the effects of the parent nodes.

Bayesian networks are useful in reasoning under uncertainty using probabilistic inference.

Python Program:

```
import numpy as np
import pandas as pd
from pgmpy.models import BayesianModel
from pgmpy.estimators import MaximumLikelihoodEstimator
from pgmpy.inference import VariableElimination

# Read Cleveland Heart Disease data
heartDisease = pd.read_csv('heart.csv')
heartDisease = heartDisease.replace('?', np.nan)

# Display data samples
print('Few examples from the dataset:')
print(heartDisease.head())

# Model the Bayesian Network
model = BayesianModel([
```

```
    ('age', 'trestbps'), ('age', 'fbs'),
    ('sex', 'trestbps'), ('exang', 'trestbps'),
    ('trestbps', 'heartdisease'), ('fbs', 'heartdisease'),
    ('heartdisease', 'restecg'), ('heartdisease', 'thalach'),
    ('heartdisease', 'chol')
])

print('\nLearning CPD using Maximum Likelihood Estimators')
model.fit(heartDisease, estimator=MaximumLikelihoodEstimator)

# Inference
print('\nInferencing with Bayesian Network:')
HeartDisease_infer = VariableElimination(model)

print('\n1. Probability of HeartDisease given Age = 30')
q1 = HeartDisease_infer.query(variables=['heartdisease'], evidence={'age': 30})
print(q1['heartdisease'])

print('\n2. Probability of HeartDisease given Cholesterol = 100')
q2 = HeartDisease_infer.query(variables=['heartdisease'], evidence={'chol': 100})
print(q2['heartdisease'])
```

```
Few examples from the dataset:
   age  sex  cp  trestbps  chol  fbs  restecg  thalach  exang  heartdisease
0   63    1   1       145   233    1        2      150      0             1
1   37    1   4       130   250    0        2      187      0             1
...

Learning CPD using Maximum Likelihood Estimators

Inferencing with Bayesian Network:

1. Probability of HeartDisease given Age = 30
+----------------+------------------+
| heartdisease   |   phi(heartdisease) |
+----------------+------------------+
| 0              |           0.671  |
| 1              |           0.329  |
+----------------+------------------+

2. Probability of HeartDisease given Cholesterol = 100
+----------------+------------------+
| heartdisease   |   phi(heartdisease) |
+----------------+------------------+
| 0              |           0.582  |
| 1              |           0.418  |
+----------------+------------------+
```

Practical No. 09

AIM: Write a program to perform inference on a Bayesian Network.

Theory:
Inference in a Bayesian Network refers to the process of deriving the posterior probability distribution of certain variables given some known evidence. Inference can be of two main types:

1. **Joint Probability Evaluation**: Computing the probability of a particular assignment of values for each variable or a subset.

2. **Conditional Probability Query (P(x|e))**: Finding the probability of certain variables (x) given the known values of other variables (evidence, e).

Bayesian Networks are probabilistic graphical models that represent a set of variables and their conditional dependencies via a directed acyclic graph (DAG).
This demonstrates inference using the `pgmpy` library in Python.

Program:

```
from pgmpy.models import BayesianNetwork
from pgmpy.factors.discrete import TabularCPD
import networkx as nx
import pylab as plt

# Defining the structure of the Bayesian Network
model = BayesianNetwork([('Guest', 'Host'), ('Price', 'Host')])

# Defining Conditional Probability Distributions (CPDs)
cpd_guest = TabularCPD(variable='Guest', variable_card=3, values=[[0.33], [0.33],
[0.33]])
cpd_price = TabularCPD(variable='Price', variable_card=3, values=[[0.33], [0.33],
[0.33]])
cpd_host = TabularCPD(
    variable='Host', variable_card=3,
    values=[[0, 0, 0, 0, 0.5, 1, 0, 1, 0.5],
            [0.5, 0, 1, 0, 0, 0, 1, 0, 0.5],
            [0.5, 1, 0, 1, 0.5, 0, 0, 0, 0]],
    evidence=['Guest', 'Price'],
    evidence_card=[3, 3]
)

# Adding CPDs to the model
model.add_cpds(cpd_guest, cpd_price, cpd_host)
```

```
# Checking if the model is valid
model.check_model()

# Performing inference
from pgmpy.inference import VariableElimination
infer = VariableElimination(model)
posterior_p = infer.query(variables=['Host'], evidence={'Guest': 2, 'Price': 2})

# Displaying the result
print(posterior_p)

# Drawing and saving the model structure
nx.draw(model, with_labels=True)
plt.savefig('model.png')
plt.close()
```

```
+---------+-------------+
| Host    |   phi(Host) |
+=========+=============+
| Host_0  |      0.5000 |
+---------+-------------+
| Host_1  |      0.0000 |
+---------+-------------+
| Host_2  |      0.5000 |
+---------+-------------+
```

Practical No. 10

AIM: Write a program to run value and policy iteration in a grid world.

Theory:
A **grid world** is a common environment used in reinforcement learning to understand the concepts of value functions and policies.

There are two key techniques for solving such environments:

1. **Policy Iteration**
    - A policy maps states to actions.
    - Policy Iteration consists of two main steps:
        - **Policy Evaluation**: Compute the value function for a given policy.
        - **Policy Improvement**: Improve the policy based on the current value function.
    - This process is repeated until the policy converges to the optimal policy $\pi*$\pi^*$\pi*$.
2. **Value Iteration**
    - Value Iteration combines policy evaluation and improvement into a single step.
    - It iteratively updates the value function until it converges, and then derives the optimal policy.

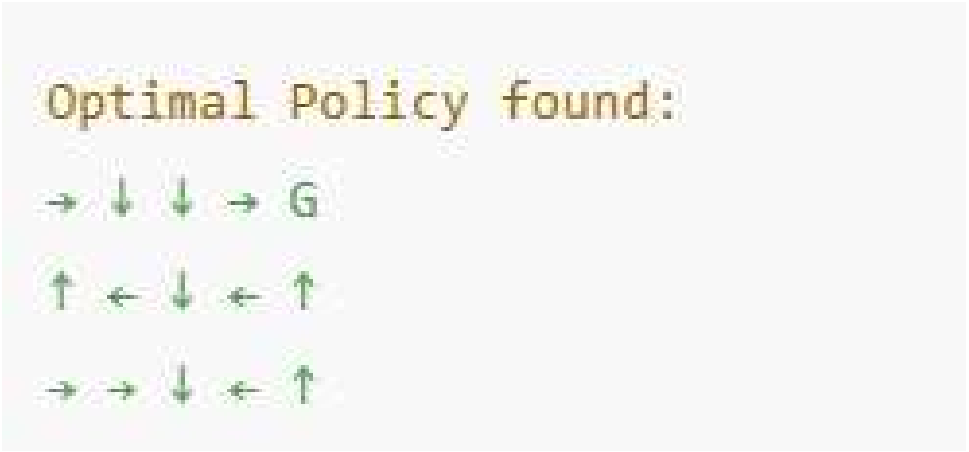
These algorithms are foundational for dynamic programming in reinforcement learning.

Program:

```
# Simplified representation of the code logic for policy iteration in a grid world

import numpy as np
import copy

def pi(self, cell, action):
    if len(self.policy) == 0:
```

```
        return 1
    return 1 if self.policyActionForCell(cell) == action else 0

def transitionProbabilityForIllegalMoves(oldState, newState):
    return 1 if newState == oldState else 0

def getTransitionProbability(self, oldState, newState, action, gridWorld):
    proposedCell = gridWorld.proposeMove(action)
    if proposedCell is None:
        return transitionProbabilityForIllegalMoves(oldState, newState)
    if oldState.isGoal():
        return 0
    if proposedCell != newState:
        return 0
    return 1

def R(self, oldState, newState, action):
    return 0 if newState and newState.isGoal() else -1

def improvePolicy(policy, gridWorld, gamma=1):
    policy = copy.deepcopy(policy)
    if len(policy.values) == 0:
        policy.evaluatePolicy(gridWorld)
    greedyPolicy = findGreedyPolicy(policy.getValues(), gridWorld, policy.gameLogic,
gamma)
    policy.setPolicy(greedyPolicy)
    return policy

def findGreedyPolicy(values, gridWorld, gameLogic, gamma=1):
    stateGen = StateGenerator()
    greedyPolicy = [Action(Actions.NONE)] * len(values)
    for (i, cell) in enumerate(gridWorld.getCells()):
        gridWorld.setActor(cell)
        if not cell.canBeEntered():
            continue
        maxPair = (Actions.NONE, -np.inf)
        for actionType in Actions:
            if actionType == Actions.NONE:
                continue
            proposedCell = gridWorld.proposeMove(actionType)
            if proposedCell is None:
                continue
            Q = 0.0
            proposedStates = stateGen.generateState(gridWorld, actionType, cell)
            for proposedState in proposedStates:
```

```
            actorPos = proposedState.getIndex()
            transitionProb = gameLogic.getTransitionProbability(cell, proposedState,
actionType, gridWorld)
            reward = gameLogic.R(cell, proposedState, actionType)
            expectedValue = transitionProb * (reward + gamma * values[actorPos])
            Q += expectedValue
         if Q > maxPair[1]:
            maxPair = (actionType, Q)
      gridWorld.unsetActor(cell)
      greedyPolicy[i] = Action(maxPair[0])
   return greedyPolicy

def policyIteration(policy, gridWorld):
   lastPolicy = copy.deepcopy(policy)
   lastPolicy.resetValues()
   while True:
      improvedPolicy = improvePolicy(lastPolicy, gridWorld)
      if improvedPolicy == lastPolicy:
         break
      improvedPolicy.resetValues()
      lastPolicy = improvedPolicy
   return improvedPolicy
```



Practical No. 11

AIM: Write a program to do reinforcement learning in the grid world.

Theory:

Reinforcement Learning (RL) is a machine learning approach where an agent learns to take actions in an environment in order to maximize a cumulative reward. Unlike supervised learning, the agent learns from interaction with the environment rather than from labeled datasets.

Key Elements of Reinforcement Learning:

1. **Agent**: The learner or decision-maker.
2. **Environment**: What the agent interacts with.
3. **State (s)**: The current situation of the agent.
4. **Action (a)**: Choices the agent can make.
5. **Reward (r)**: Feedback from the environment.
6. **Policy (π)**: Strategy that the agent employs to determine actions.
7. **Value Function**: Measures how good a state or action is in terms of expected rewards.

The goal is to find the **optimal policy** that maximizes the **cumulative reward** over time.

Program:

```
import numpy as np

BOARD_ROWS = 3
BOARD_COLS = 4
WIN_STATE = (0, 3)
```

```
LOSE_STATE = (1, 3)
START = (2, 0)

class State:
    def __init__(self):
        self.state = START
        self.isEnd = False
        self.determine = True

    def giveReward(self):
        if self.state == WIN_STATE:
            return 1
        elif self.state == LOSE_STATE:
            return -1
        else:
            return 0

    def isEndFunc(self):
        if self.state == WIN_STATE or self.state == LOSE_STATE:
            self.isEnd = True

    def nxtPosition(self, action):
        if self.determine:
            if action == "up":
                nxtState = (self.state[0] - 1, self.state[1])
            elif action == "down":
                nxtState = (self.state[0] + 1, self.state[1])
            elif action == "left":
                nxtState = (self.state[0], self.state[1] - 1)
            else:
                nxtState = (self.state[0], self.state[1] + 1)
        else:
            nxtState = self.state

        # check if new position is legal
        if (0 <= nxtState[0] <= 2) and (0 <= nxtState[1] <= 3) and nxtState != (1, 1):
            return nxtState
        return self.state

class Agent:
    def __init__(self):
        self.State = State()
        self.state_values = {}
        self.states = []
        self.actions = ["up", "down", "left", "right"]
```

```
    self.lr = 0.2
    self.exp_rate = 0.3

    # initialize all state values to 0
    for i in range(3):
        for j in range(4):
            if (i, j) != (1, 1):
                self.state_values[(i, j)] = 0

def reset(self):
    self.State = State()
    self.states = []

def chooseAction(self):
    mx_nxt_reward = float("-inf")
    action = ""
    if np.random.uniform(0, 1) <= self.exp_rate:
        action = np.random.choice(self.actions)
    else:
        for a in self.actions:
            nxt_reward = self.state_values[self.State.nxtPosition(a)]
            if nxt_reward >= mx_nxt_reward:
                action = a
                mx_nxt_reward = nxt_reward
    return action

def takeAction(self, action):
    position = self.State.nxtPosition(action)
    newState = State()
    newState.state = position
    return newState

def play(self, rounds=10):
    i = 0
    while i < rounds:
        if self.State.isEnd:
            reward = self.State.giveReward()
            self.state_values[self.State.state] = reward
            print("Game End Reward", reward)
            for s in reversed(self.states):
                reward = self.state_values[s] + self.lr * (reward - self.state_values[s])
                self.state_values[s] = round(reward, 3)
            self.reset()
        else:
            i += 1
```

```
            action = self.chooseAction()
            self.states.append(self.State.nxtPosition(action))
            print("Current position {} action {}".format(self.State.state, action))
            self.State = self.takeAction(action)
            self.State.isEndFunc()
            print("Next state", self.State.state)
            print(" ")

# Play the game
agent = Agent()
agent.play(10)
```

```
Current position (2, 0) action right
Next state (2, 1)

Current position (2, 1) action up
Next state (1, 1)

Current position (1, 1) action right
Next state (1, 1)

...

Game End Reward 1
```