Vidyavardhini's
College of Engineering & Technology

Vasai Road (W)

Department of Computer Engineering

Laboratory Manual
(Student Copy)

Semester	VI	**Class**	**T.E**
Course Code	CSL604		
Course Name	Artificial Intelligence Lab		

Vidyavardhini's College of Engineering & Technology

Vision

To be a premier institution of technical education; always aiming at becoming a valuable resource for industry and society.

Mission

- To provide technologically inspiring environment for learning.
- To promote creativity, innovation and professional activities.
- To inculcate ethical and moral values.
- To cater personal, professional and societal needs through quality education.

Department Vision:

To evolve as a center of excellence in the field of Computer Engineering to cater to industrial and societal needs.

Department Mission:

- To provide quality technical education with the aid of modern resources.
- Inculcate creative thinking through innovative ideas and project development.
- To encourage life-long learning, leadership skills, entrepreneurship skills with ethical & moral values.

Program Education Objectives (PEOs):

PEO1: To facilitate learners with a sound foundation in the mathematical, scientific and engineering fundamentals to accomplish professional excellence and succeed in higher studies in Computer Engineering domain
PEO2: To enable learners to use modern tools effectively to solve real-life problems in the field of Computer Engineering.
PEO3: To equip learners with extensive education necessary to understand the impact of computer technology in a global and social context.
PEO4: To inculcate professional and ethical attitude, leadership qualities, commitment to societal responsibilities and prepare the learners for life-long learning to build up a successful career in Computer Engineering.

Program Specific Outcomes (PSOs):

PSO1: Analyze problems and design applications of database, networking, security, web technology, cloud computing, machine learning using mathematical skills, and computational tools.

PSO2: Develop computer-based systems to provide solutions for organizational, societal problems by working in multidisciplinary teams and pursue a career in the IT industry.

Program Outcomes (POs):

Engineering Graduates will be able to:

- **PO1. Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
- **PO2. Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
- **PO3. Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
- **PO4. Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
- **PO5. Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
- **PO6. The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
- **PO7. Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
- **PO8. Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
- **PO9. Individual and teamwork:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
- **PO10. Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
- **PO11. Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
- **PO12. Life-long learning:** Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Course Objectives

1	To conceptualize the basic ideas and techniques underlying the design of intelligent systems.
2	To make students understand and Explore the mechanism of mind that enables intelligent thought and action.
3	To make students understand advanced representation formalism and search techniques.
4	To make students understand how to deal with uncertain and incomplete information.

Course Outcomes

CO	Students will be able to	**Action verbs**	**Bloom's Level**
CSL604.1	Analyze PEAS descriptors of an Intelligent agent.	Analyze	Analyze (Level 4)
CSL604.2	Implement Uninformed searching algorithms for problem solving	Implement	Create (Level 6)
CSL604.3	Implement Informed searching algorithms for problem solving.	Implement	Create (Level 6)
CSL604.4	Create a knowledge base using any AI language.	Create	Create (Level 6)
CSL604.5	Create Inference system using reasoning technique for given AI problem	Create	Create (Level 6)
CSL604.6	Identify the components of AI applications in the field of NLP and Healthcare.	Identify	Analyze (Level 4)

Mapping of Experiments with Course Outcomes

Experiments	Course Outcomes					
	CSL604.1	CSL604.2	CSL604.3	CSL604.4	CSL604.5	CSL604.6
Identification of the problem and Determination of its PEAS Descriptor	3	-	-	-	-	-
Study and Implementation of Depth first search for problem solving.	-	3	-	-	-	-
Study and Implementation of Breadth first search for problem solving.	-	3	-	-	-	-
Study and Implementation of Informed search method: A* Search algorithm	-	-	3	-	-	-
Study and Implementation of Min-Max algorithm.	-	-	3	-	-	-
Study and create knowledge base in Prolog.	-	-	-	3	-	-
Study and Implementation of unification algorithm in Prolog.	-	-	-	-	3	-

Implementation for Bayes Belief Network	-	-	-	-	3	-
Case Study on an Expert System in healthcare domain.	-	-	-	-	-	3
Case Study on Natural Language Processing applications.	-	-	-	-	-	3

Enter correlation level 1, 2 or 3 as defined below
1: Slight (Low) 2: Moderate (Medium) 3: Substatial (High)
If there is no correlation put "—".

INDEX

Sr. No.	Name of Experiment	D.O.P.	D.O.C.	Page No.	Remark
1	Identification of the problem and Determination of its PEAS Descriptor.				
2	Study and Implementation of Depth first search for problem solving.				
3	Study and Implementation of Breadth first search for problem solving.				
4	Study and Implementation of Informed search method: A* Search algorithm				
5	Study and Implementation of Min-Max algorithm.				
6	Study and create knowledge base in Prolog.				
7	Study and Implementation of unification algorithm in Prolog.				
8	Implementation for Bayes Belief Network				
9	Case Study on an Expert System in healthcare domain.				
10	Case Study on Natural Language Processing applications.				

D.O.P: Date of performance

D.O.C : Date of correction

Experiment No.1
Identification of the problem and Determination of its PEAS Descriptor.
Date of Performance:
Date of Submission:

Aim: Identification of the problem and Determination of its PEAS Descriptor.

Objective: To analyze the Performance Measure, Environment, Actuators, Sensors (PEAS) for given problem before building an intelligent agent.

Theory:

The goal of AI is to build intelligent system which can think and act rationally. For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has. Rationality is relative to a performance measure.

Designer of rational agent can judge rationality based on:

- The performance measure that defines the criterion of success.
- The agent prior knowledge of the environment.
- The possible actions that the agent can perform.
- The agent's percept sequence to date.

When we define a rational agent, we group these properties under PEAS, the problem specification for the task environment.

Performance Measure:

If the objective function to judge the performance of the agent, things we can evaluate an agent against to know how well it performs.

Environment:

It the real environment where the agent need to deliberate actions. What the agent can perceive.

Actuators:

These are the tools, equipment or organs using which agent performs actions in the environment. This works as output of the agent. What an agent can use to act in its environment.

Sensors:

These are tools, organs using which agent captures the state of the environment. This works as input to the agent. What an agent can use to perceive its environment

PEAS Descriptors Examples/Problems

1. PEAS descriptor for Automated Car Driver:

Performance Measure:

- **Safety**: Automated system should be able to drive the car safely without dashing anywhere.
- **Optimum speed:** Automated system should be able to maintain the optimal speed depending upon the surroundings.
- **Comfortable journey:** Automated system should be able to give a comfortable journey to the end user.

Environment:

- **Roads:** Automated car driver should be able to drive on any kind of a road ranging from city roads to highway.
- **Traffic conditions:** You will find different sort of traffic conditions for different type of roads.

Actuators:

- **Steering wheel:** used to direct car in desired directions.
- **Accelerator, gear:** To increase or decrease speed of the car.

Sensors:

- To take i/p from environment in car driving example cameras, sonar system etc.

2. PEAS descriptor for playing soccer.

Performance Measure: scoring goals, defending, speed

Environment: playground, teammates, opponents, ball

Actuators: body, dribbling, tackling, passing ball, shooting

Sensors: camera, ball sensor, location sensor, other players locator

3. PEAS descriptor for Exploring the subsurface oceans of Titan.

Performance Measure: safety, images quality, video quality

Environment: ocean, water

Actuators: mobile diver, steering, break, accelerator

Sensors: video, accelerometers, depth sensor, GPS

4. PEAS descriptor for Shopping for used AI books on the Internet.

Performance Measure: price, quality, authors, book review

Environment: web, vendors, shippers

Actuators: fill in form, follow URL, display to user

Sensors: HTML

5. PEAS descriptor for playing a tennis match.

Performance Measure: winning

Environment: playground, acquet, ball, opponent

Actuators: ball, raquet, joint arm

Sensors: ball locator, camera, racquet sensor, opponent locator

6. PEAS descriptor for practicing tennis against a wall.

Performance Measure: hit speed, hit accuracy

Environment: playground, racquet, ball, wall

Actuators: ball, racquet, joint arm

Sensors: ball locator, camera, racquet sensor

Conclusion:
Comment on importance of PEAS properties in terms of design of intelligent agent.

Experiment No.2
Study and Implementation of Depth first search for problem solving.
Date of Performance:
Date of Submission:

Aim: Study and Implementation of Depth first search for problem solving.

Objective: To study the uninformed searching techniques and its implementation for problem solving.

Theory:

Artificial Intelligence is the study of building agents that act rationally. Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.

- A search problem consists of:
    - **A State Space.** Set of all possible states where you can be.
    - **A Start State.** The state from where the search begins.
    - **A Goal Test.** A function that looks at the current state returns whether or not it is the goal state.
- The **Solution** to a search problem is a sequence of actions, called the **plan** that transforms the start state to the goal state.
- This plan is achieved through search algorithms.

Depth First Search: DFS is an uninformed search method. It is also called blind search. Uninformed search strategies use only the information available in the problem definition. A search strategy is defined by picking the order of node expansion. Depth First Search (DFS) searches deeper into the problem space. It is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

The basic idea is as follows:

Pick a starting node and push all its adjacent nodes into a stack.

Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

Repeat this process until the stack is empty.

However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

Algorithm:

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited

2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1.Start by putting any one of the graph's vertices on top of a stack.

2.Take the top item of the stack and add it to the visited list.

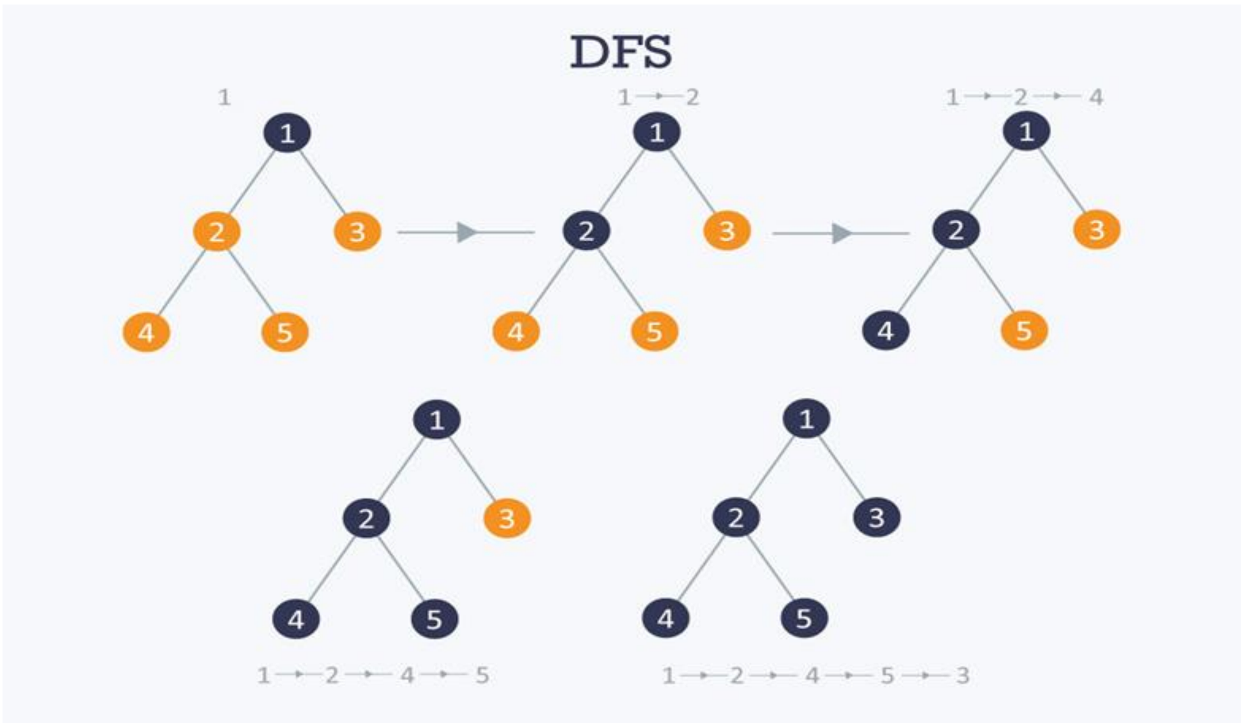
3.Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.

4.Keep repeating steps 2 and 3 until the stack is empty.

Pseudocode:

```
DFS-iterative (G, s):                                   //Where G is
graph and s is source vertex
      let S be stack
      S.push( s )            //Inserting s in stack
      mark s as visited.
      while ( S is not empty):
          //Pop a vertex from stack to visit next
          v  =  S.top( )
         S.pop( )
         //Push all the neighbours of v in stack that are not
visited
        for all neighbours w of v in Graph G:
            if w is not visited :
                     S.push( w )
                    mark w as visited


    DFS-recursive(G, s):
        mark s as visited
        for all neighbours w of s in Graph G:
            if w is not visited:
                DFS-recursive(G, w)
```

DFS Working: Example



Path: 1 → 2→ 4→ 5→ 3

Searching Strategies are evaluated along the following dimensions:

1. **Completeness:** does it always find a solution if one exists?
2. **Time complexity:** number of nodes generated
3. **Space complexity:** maximum number of nodes in memory
4. **Optimality:** does it always find a least-cost solution?

Properties of depth-first search:

1. Complete:- No: fails in infinite-depth spaces, spaces with loops.
2. Time Complexity: O(bm)
3. Space Complexity: O(bm), i.e., linear space!
4. Optimal: No

Advantages of Depth-First Search:

1. Memory requirement is only linear with respect to the search graph.
2. The time complexity of a depth-first Search to depth d is O(b^d)

3. If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.

Disadvantages of Depth-First Search:

1. There is a possibility that it may go down the left-most path forever. Even a finite graph can generate an infinite tree.
2. Depth-First Search is not guaranteed to find the solution.
3. No guarantee to find a optimum solution, if more than one solution exists.

Applications

How to find connected components using DFS?

A graph is said to be disconnected if it is not connected, i.e. if two nodes exist in the graph such that there is no edge in between those nodes. In an undirected graph, a connected component is a set of vertices in a graph that are linked to each other by paths.

Consider the example given in the diagram. Graph G is a disconnected graph and has the following 3 connected components.

- First connected component is 1 → 2 → 3 as they are linked to each other
- Second connected component 4 → 5
- Third connected component is vertex 6

Conclusion:

Comment on your implemented program and results you got.

Experiment No. 3
Study and Implementation of Breadth first search for problem solving.
Date of Performance:
Date of Submission:

Aim: Study and Implementation of Breadth first search for problem solving.

Objective: To study the uninformed searching techniques and its implementation for problem solving.

Theory:

Artificial Intelligence is the study of building agents that act rationally. Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.

- A search problem consists of:
    - **A State Space.** Set of all possible states where you can be.
    - **A Start State.** The state from where the search begins.
    - **A Goal Test.** A function that looks at the current state returns whether or not it is the goal state.
- The **Solution** to a search problem is a sequence of actions, called the **plan** that transforms the start state to the goal state.
- This plan is achieved through search algorithms.

Breadth First Search: BFS is a uninformed search method. It is also called blind search. Uninformed search strategies use only the information available in the problem definition. A search strategy is defined by picking the order of node expansion. It expands nodes from the root of the tree and then generates one level of the tree at a time until a solution is found. It is very easily implemented by maintaining a queue of nodes. Initially the queue contains just the root. In each iteration, node at the head of the queue is removed and then expanded. The generated child nodes are then added to the tail of the queue.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

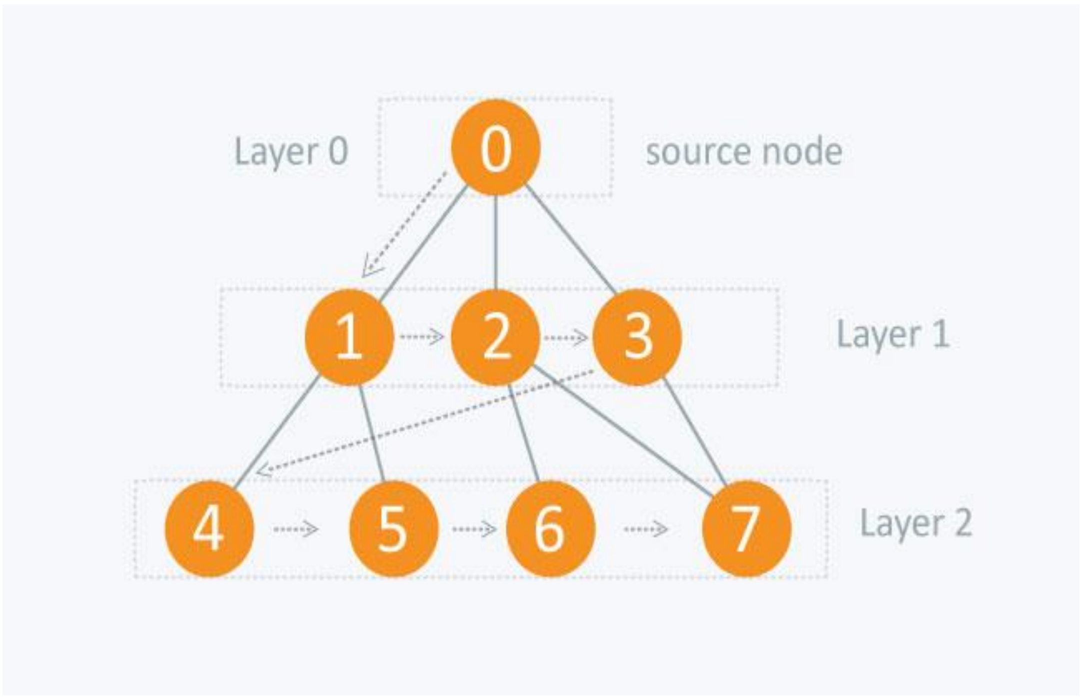
1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

BFS Algorithm:

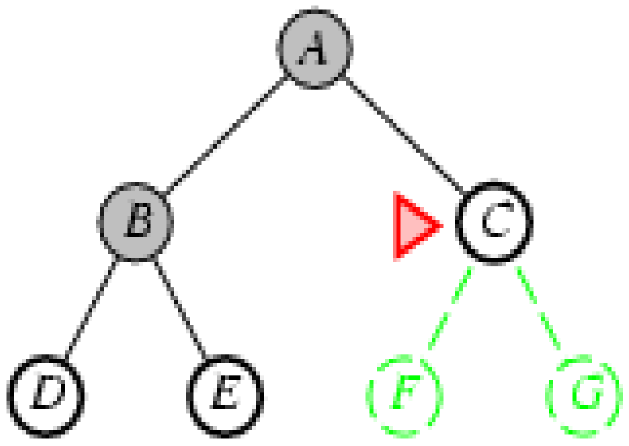
Pseudocode:

```
BFS (G, s)             //Where G is the graph and s is the source node
      let Q be queue.
      Q.enqueue( s ) //Inserting s in queue until all its neighbour
vertices are marked.

      mark s as visited.
      while ( Q is not empty)
           //Removing that vertex from queue,whose neighbour will be
visited now
           v  =  Q.dequeue( )

          //processing all the neighbours of v
          for all neighbours w of v in Graph G
               if w is not visited
                        Q.enqueue( w )             //Stores w in Q to
further visit its neighbour
                        mark w as visited.
```

Working of BFS:



Example: Initial Node: A Goal Node: C



Searching Strategies are evaluated along the following dimensions:

1. Completeness: does it always find a solution if one exists?
2. Time complexity: number of nodes generated
3. Space complexity: maximum number of nodes in memory
4. Optimality: does it always find a least-cost solution?

Properties of Breadth-first search:

1. **Complete**: - Yes: if b is finite.
2. **Time Complexity**: O(b^d+1)
3. **Space Complexity**: O(b^d+1)
4. **Optimal**: Yes

Advantages of Breadth-First Search:

1. Breadth first search will never get trapped exploring the useless path forever.
2. If there is a solution, BFS will definitely find it out.
3. If there is more than one solution then BFS can find the minimal one that requires less number of steps.

Disadvantages of Breadth-First Search:

1. The main drawback of Breadth first search is its memory requirement. Since each level of the tree must be saved in order to generate the next level, and the amount of memory is proportional to the number of nodes stored, the space complexity of BFS is O(bd).
2. If the solution is farther away from the root, breath first search will consume lot of time.

Applications:

How to determine the level of each node in the given tree?

As you know in BFS, you traverse level wise. You can also use BFS to determine the level of each node.

Conclusion:

Comment on your implemented program and results you got.

Experiment No.4
Study and Implementation of Informed search method: A* Search algorithm.
Date of Performance:
Date of Submission:

Aim: Study and Implementation of A* search algorithm.

Objective: To study the informed searching techniques and its implementation for problem solving.

Theory:

A* (pronounced as "A star") is a computer algorithm that is widely used in path finding and graph traversal. The algorithm efficiently plots a walkable path between multiple nodes, or points, on the graph. However, the A* algorithm introduces a heuristic into a regular graph-searching algorithm, essentially planning ahead at each step so a more optimal decision is made.

A* is an extension of Dijkstra's algorithm with some characteristics of breadth-first search (BFS). Like Dijkstra, A* works by making a lowest-cost path tree from the start node to the target node. What makes A* different and better for many searches is that for each node, A* uses a function f(n)f(n)f(n) that gives an estimate of the total cost of a path using that node. Therefore, A* is a heuristic function, which differs from an algorithm in that a heuristic is more of an estimate and is not necessarily provably correct.
A* expands paths that are already less expensive by using this function:

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

where

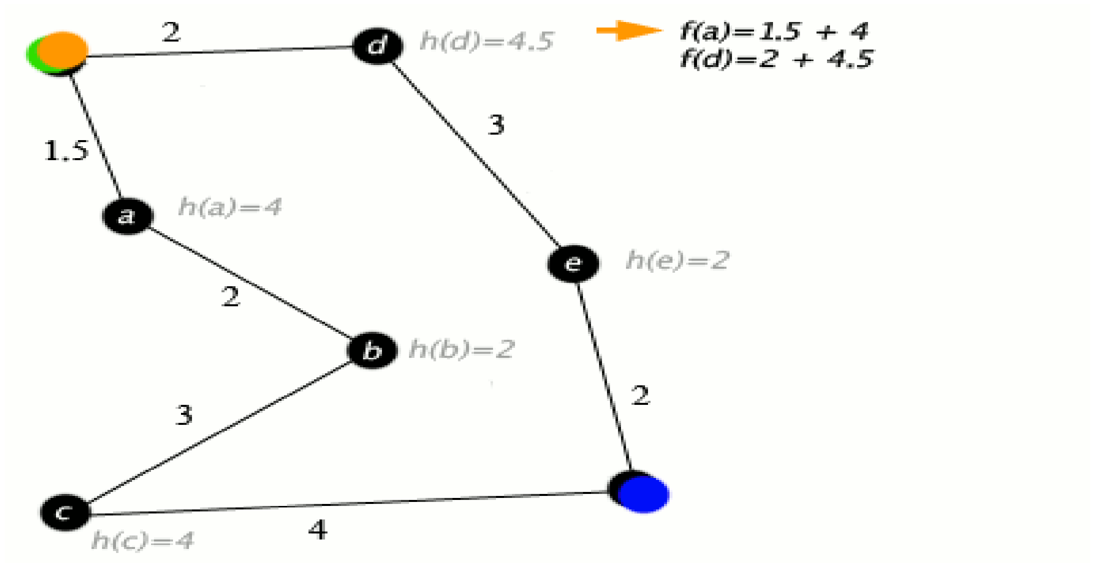
- f(n) = total estimated cost of path through node n
- g(n) = cost so far to reach node n
- h(n) = estimated cost from n to goal. This is the heuristic part of the cost function, so it is like a guess.

Pseudocode

The following pseudocode describes the algorithm:

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]

        total_path.prepend(current)

    return total_path


// A* finds a path from start to goal.

// h is the heuristic function. h(n) estimates the cost to reach goal from node n.

function A_Star(start, goal, h)

    // The set of discovered nodes that need to be (re-)expanded.

    // Initially, only the start node is known.

    openSet := {start}


    // For node n, cameFrom[n] is the node immediately preceding it on the cheapest path from
start to n currently known.

    cameFrom := an empty map


    // For node n, gScore[n] is the cost of the cheapest path from start to n currently known.

    gScore := map with default value of Infinity

    gScore[start] := 0


    // For node n, fScore[n] := gScore[n] + h(n).

    fScore := map with default value of Infinity

    fScore[start] := h(start)


    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        closedSet.Add(current)
        for each neighbor of current
            if neighbor in closedSet
                continue
            // d(current,neighbor) is the weight of the edge from current to neighbor
            // tentative_gScore is the distance from start to the neighbor through current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
                // This path to neighbor is better than any previous one. Record it!
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := gScore[neighbor] + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)

    // Open set is empty but goal was never reached
    return failure
```

An example of an A* algorithm in action where nodes are cities connected with roads and h(x) is the straight-line distance to target point:



Conclusion:

Comment on your implemented program and results you got.

Experiment No.5
Study and Implementation of Min-Max algorithm.
Date of Performance:
Date of Submission:

Aim: Study and Implementation of Min-Max algorithm.

Objective: To study the adversarial search technique and its implementation in game playing.

Theory:

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.
In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.
Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

Mini-Max Algorithm in Artificial Intelligence

- In decision-making and game theory, the mini-max algorithm is a recursive or backtracking method. It suggests the best move for the player, provided that the opponent is likewise playing well.
- In AI, the Min-Max algorithm is mostly employed for game play. Chess, checkers, tic-tac-toe, go, and other two-player games are examples. This Algorithm calculates the current state's minimax choice.
- The game is played by two players, one named MAX and the other named MIN, in this algorithm.
- Both players FIGHT it, since the opponent player receives the smallest benefit while they receive the greatest profit.
- Both players in the game are adversaries, with MAX selecting the maximum value and MIN selecting the minimum value.
- For the exploration of the entire game tree, the minimax method uses a depth-first search strategy.
- For the exploration of the entire game tree, the minimax method uses a depth-first search strategy.
- The minimax algorithm descends all the way to the tree's terminal node, then recursively backtracks the tree.

Pseudo-code for MinMax Algorithm:

```
function minimax(node, depth, maximizingPlayer) is
  if depth == 0 or node is a terminal node then
  return static evaluation of node

  if MaximizingPlayer then      // for Maximizer Player
  maxEva =- infinity
   for each child of node do
   eva= minimax(child, depth - 1, false)
  maxEva= max(maxEva,eva)       //gives Maximum of the values
  return maxEva

  else                          // for Minimizer player
   minEva =+ infinity
   for each child of node do
   eva= minimax(child, depth - 1, true)
   minEva = min(minEva, eva)        //gives minimum of the values
   return minEva
```

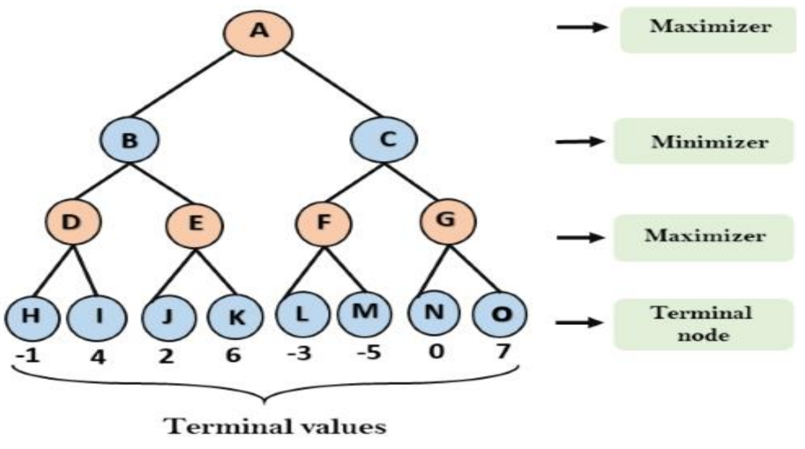
Initial call:

minimax(node, 3, true)

Working of Min-Max Algorithm:

- A simple example can be used to explain how the minimax algorithm works. We've included an example of a game-tree below, which represents a two-player game.
- There are two players in this scenario, one named Maximizer and the other named Minimizer.
- Maximizer will strive for the highest possible score, while Minimizer will strive for the lowest possible score.
- Because this algorithm uses DFS, we must go all the way through the leaves to reach the terminal nodes in this game-tree.
- The terminal values are given at the terminal node, so we'll compare them and retrace the tree till we reach the original state. The essential phases in solving the two-player game tree are as follows:

Step 1: The method constructs the whole game-tree and applies the utility function to obtain utility values for the terminal states in the first step. Let's assume A is the tree's initial state in the diagram below. Assume that the maximizer takes the first turn with a worst-case initial value of -infinity, and the minimizer takes the second turn with a worst-case initial value of +infinity.

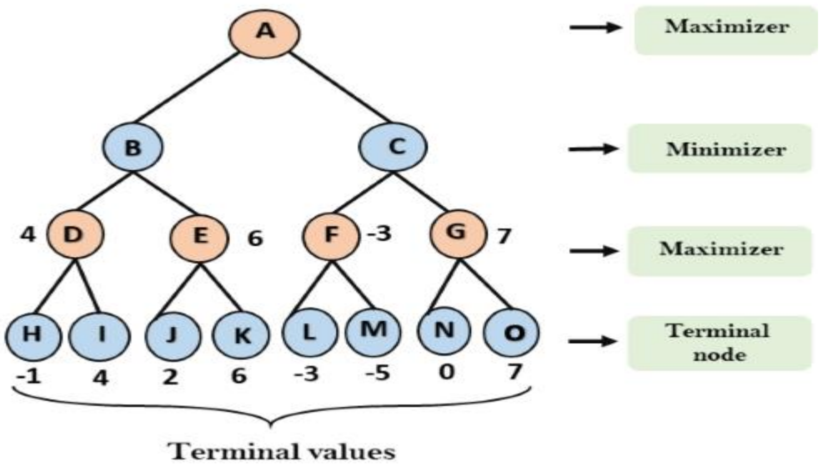


Example: Working of Min-Max Algorithm

step-1

Step 2: Next, we'll locate the Maximizer's utilities value, which is -, and compare each value in the terminal state to the Maximizer's initial value to determine the upper nodes' values. It will select the best option from all of them.

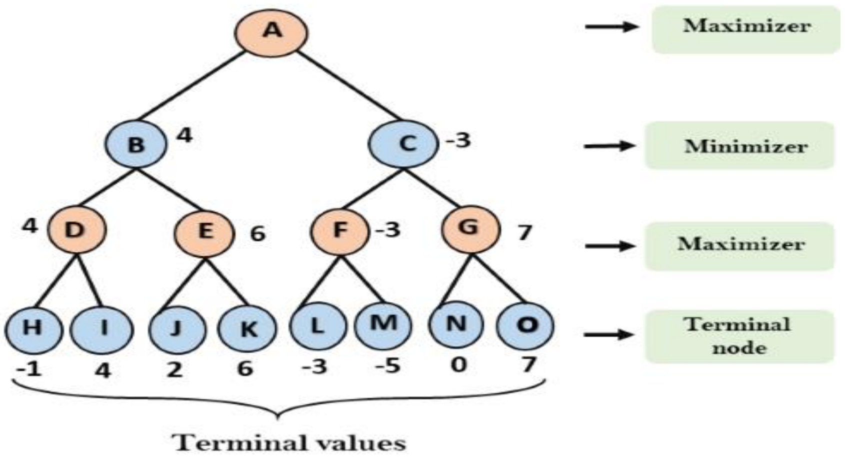
- **For node D max(-1,- -∞) => max(-1,4)= 4**
- **For Node E max(2, -∞) => max(2, 6)= 6**
- **For Node F max(-3, -∞) => max(-3,-5) = -3**
- **For node G max(0, -∞) = max(0, 7) = 7**



Step-2

Step 3: Now it's the minimizer's time, thus it'll compare all nodes' values with + and determine the 3rd layer node values.

- **For node B = min(4,6) = 4**
- **For node C = min (-3, 7) = -3**

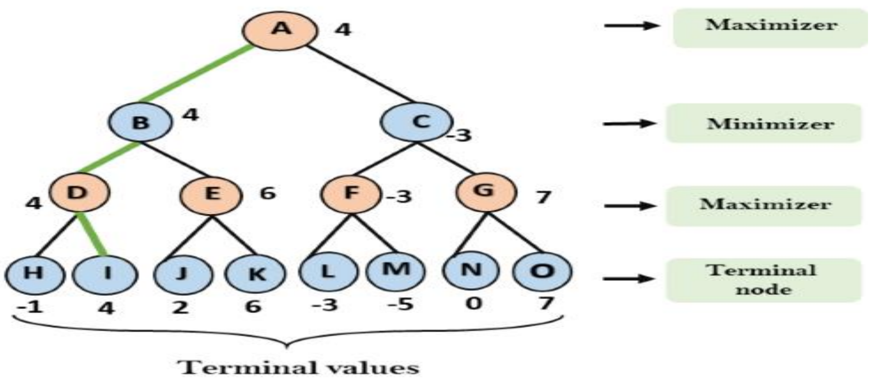


Step 3

In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α = -∞, and β = 3 will also be passed.

Step 4: Now it's Maximizer's turn, and it'll choose the maximum value of all nodes and locate the root node's maximum value. There are only four layers in this game tree, so we can go to the root node right away, but there will be more layers in real games.

For node A max(4, -3)= 4



Step 4

Properties of Mini-Max algorithm:

- **Complete –**The Min-Max algorithm is finished. In the finite search tree, it will undoubtedly locate a solution (if one exists).
- **Optimal-** If both opponents are playing optimally, the Min-Max algorithm is optimal.
- **Time complexity-** Because it executes DFS for the game-tree, the time complexity of the Min-Max algorithm is **O(b^m)**, where b is the game-branching tree's factor and m is the tree's maximum depth.
- **Space Complexity-** Mini-max method has a space complexity that is similar to DFS, which is **O(bm)**.

Limitation of the minimax Algorithm:

The biggest disadvantage of the minimax algorithm is that it becomes extremely slow while playing complex games like chess or go. This style of game contains a lot of branching, and the player has a lot of options to choose from. The minimax algorithm's drawback can be alleviated by using **alpha-beta pruning**, which we will explore in the next section. the depth to which the tree can grow.

Conclusion:

Comment on your implemented program and results you got.

Experiment No.6
Study and create knowledge base in Prolog.
Date of Performance:
Date of Submission:

Aim: Study and create knowledge base in Prolog.

Objective: To study and use AI programming language to create knowledge base.

Theory: Take any problem and represent the knowledge (facts) in prolog. Also you can use this for reasoning purpose.

Example: The problem of murder mystery.

Five persons Alice, her husband, brother, son and daughter

Event: One murder. One of the five is victim and one is Killer.

Rules:

1) Husband and Alice was not together on the night of murder.
2) The killer and victim were on the beach.
3) On the night of murder, one male and one female was in the bar.
4) The victim was twin and the counterpart was innocent.
5) The killer was younger than the victim.
6) One child was alone at home.

Code for Prolog problem of murder mystery in Artificial Intelligence

```
predicates
%    pair(symbol,symbol)
     iskiller(symbol,symbol)
    male(symbol)
    female(symbol)
    isvictim(symbol)
    not_at_bar(symbol,symbol)
    not_at_beach(symbol,symbol)
    not_alone(symbol)
    twin(symbol,symbol)
    younger(symbol,symbol)
    child(symbol)

clauses
    male(husband).
    male(brother).
    male(son).

    female(alice).
    female(daughter).
```

```
twin(brother,alice).
twin(son,daughter).

child(son).
child(daughter).
```

Conclusion: Comment on programs you have implemented using prolog language for knowledge declaration.

Experiment No.7
Study and Implementation of unification algorithm in Prolog.
Date of Performance:
Date of Submission:

Aim: Study and Implementation of unification algorithm in Prolog.

Objective: To study about how to use AI Programming language (Prolog) for developing inferencing engine using Unification process and knowledge declared in Prolog.

Requirement: Turbo Prolog 2.0 or above / Windows Prolog.

Theory:

Unification is a process of making two different logical atomic expressions identical by finding a substitution. ... It takes two literals as input and makes them identical using substitution. Let $\Psi 1$ and $\Psi 2$ be two atomic sentences and σ be a unifier such that, $\Psi 1\sigma = \Psi 2\sigma$, then it can be expressed as UNIFY($\Psi 1$, $\Psi 2$).

For example, if one term is f(X, Y) and the second is f(g(Y, a), h(a)) (where upper case names are variables and lower case are constants) then the two terms can be unified by identifying X with g(h(a), a) and Y with h(a) making both terms look like f(g(h(a), a), h(a)). The unification can be represented by a pair of substitutions {X → g(h(a), a)} and {Y → h(a)}.

Unification Algorithm:

```
FUNCTION unify( t1, t2 ) RETURNS (unifiable : BOOLEAN, sigma :
SUBSTITUTION)
BEGIN
   IF t1 OR t2 is a variable THEN
      BEGIN
         let x be the variable and let t be the other term
         IF x == t THEN (unifiable, sigma) := (TRUE, NULL_SUBSTITUTION);
         ELSE IF x occurs in t THEN unifiable == FALSE;
         ELSE (unifiable, sigma) := (TRUE, {x <- t});
      END
   ELSE
      BEGIN
         assume t1 == f(x1, ..., xn) and t2 == g(y1, ... ym)
         IF f != g OR m != n THEN unifiable = FALSE;
         ELSE
            BEGIN
               k := 0;
               unifiable := TRUE;
               sigma := NULL_SUBSTITUTION;
               WHILE k < m AND unifiable DO
                  BEGIN
                     k := k + 1;
                     (unifiable, tau) := unify( sigma( xk ), sigma( yk ) );
                     IF unifiable THEN sigma := compose( tau, sigma );
                  END
            END
      END
   RETURN (unifiable, sigma);
END
```

Implementation Notes

1. To extract the name of a functor and its arguments, you may use the special built-in rules **functor/3**, **arg/3**, and "**=..**" . (Prolog allows overloading of rule names; the notation foo/2 denotes the foo rule that takes two arguments.) They are used as follows:
    1. functor(f(x,y),F,N) ==> F=f and N=2
    2. arg(1,f(x,y),A) ==> A=x
    3. f(x,y) =.. L ==> L = [f,x,y]

   Incidentally, an atom is treated as a 0-argument functor.

2. As an option, you may encode functors to be unified as lists in prefix notation. For example, f(x) would be encoded as [f, x]. For a more complicated example, the following function:

   f(3, g(x))

   would be encoded as:
   [f, 3, [g, x]]
   This notation doesn't look as nice, but it might make the implementation simpler.

3. You must choose how to distinguish *variables* from *atoms* in the expressions you are matching. For example, if **a** and **b** are constants, then unification of **a** and **b** should fail. However, if **A** and **B** are both variables, then unification should succeed, with the single substitution **A** -> **B**. A reasonable choice is that t, u, v, w, x, y, and z are variables, while all other letters are constants. In any case, please document your choice.

Testing Your Unifier

Here are some tests you should try before stopping work on your unifier. Harder tests are towards the bottom.

1. Two atoms should unify iff both atoms are the same. Two different atoms should fail to unify.
2. A variable should unify with anything that does not contain that variable. For example, x should unify with f(g(y),3,(h(a,z))), but not with f(x).
3. A variable should unify with itself. For example, x should unify with x.
4. Your algorithm should handle cases where a variable appears in multiple locations. For example, all of the following should unify:
    - f(x,x) = f(a,a)
    - f(x,g(x)) = f(a, g(x))
    - f(x, y) = f(y, x)

   And the following should *NOT* unify:

    - f(x,x) = f(a,b)
    - f(x,g(x)) = g(a, g(b))

5. When unifying functors, all arguments should unify. For example, g(h(1,2,3,4), 5) does not unify with g(h(1,8,3,4),5).
6. There are plenty of other things to try. These are just some examples to start.

Unification in Prolog:

The way in which Prolog matches two terms is called unification. The idea is similar to that of unification in logic: we have two terms and we want to see if they can be made to represent the same structure. For example, we might have in our database the single Polog clause:

parent(alan, clive).

and give the query:

|?- parent(X,Y).

We would expect X to be instantiated to alan and Y to be instantiated to clive when the query succeeds. We would say that the term parent(X,Y) unifies with the term parent(alan, clive) with X bound to alan and Y bound to clive. The unification algorithm in Prolog is roughly this:

df:un Given two terms and which are to be unified:

If and are constants (i.e. atoms or numbers) then if they are the same succeed. Otherwise fail.

If is a variable then instantiate to .

Otherwise, If is a variable then instantiate to .

Otherwise, if and are complex terms with the same arity (number of arguments), find the principal functor of and principal functor of . If these are the same, then take the ordered set of arguments of of and the ordered set of arguments of . For each pair of arguments and from the same position in the term, must unify with .

Otherwise fail.

For example: applying this procedure to unify foo(a,X) with foo(Y,b) we get:

```
foo(a,X) and foo(Y,b) are complex terms with the same arity (2).

The principal functor of both terms is foo.

The arguments (in order) of foo(a,X) are a and X.

The arguments (in order) of foo(Y,b) are Y and b.
```

So a and Y must unify , and X and b must unify.

Y is a variable so we instantiate Y to a.

X is a variable so we instantiate X to b.

The resulting term, after unification is foo(a,b).

The built in Prolog operator '=' can be used to unify two terms. Below are some examples of its use. Annotations are between ** symbols.

```
| ?- a = a.         ** Two identical atoms unify **

yes

| ?- a = b.         ** Atoms don't unify if they aren't identical **

no

| ?- X = a.         ** Unification instantiates a variable to an atom **

     X=a

yes

| ?- X = Y.         ** Unification binds two differently named variables **

     X=_125451      ** to a single, unique variable name **

     Y=_125451

yes

| ?- foo(a,b) = foo(a,b).       ** Two identical complex terms unify **

yes

| ?- foo(a,b) = foo(X,Y).       ** Two complex terms unify if they are **

     X=a                              ** of the same arity, have the same
principal**

     Y=b                        ** functor and their arguments unify **

yes

| ?- foo(a,Y) = foo(X,b).       ** Instantiation of variables may occur **

     Y=b                        ** in either of the terms to be unified **

     X=a

yes

| ?- foo(a,b) = foo(X,X).       ** In this case there is no unification **
```

```
no                                ** because foo(X,X)  must have the same **
                                  ** 1st and 2nd arguments **
| ?- 2*3+4 = X+Y.       ** The term 2*3+4 has principal functor + **
     X=2*3                       **  and therefore unifies with X+Y with X
instantiated**
     Y=4                ** to 2*3 and Y instantiated to 4 **
yes
| ?- [a,b,c] = [X,Y,Z]. ** Lists unify just like other terms **
     X=a
     Y=b
     Z=c
yes
| ?- [a,b,c] = [X|Y].   ** Unification using the '|' symbol  can be used **
     X=a                 ** to find the head element, X, and tail list, Y,
**
     Y=[b,c]            ** of a list **
yes
| ?- [a,b,c] = [X,Y|Z]. ** Unification on lists doesn't have to be **
     X=a                ** restricted to finding the first head element **
     Y=b                ** In this case we find the 1st and 2nd elements **
     Z=[c]              ** (X and Y) and then the tail list (Z) **
yes
| ?- [a,b,c] = [X,Y,Z|T].         ** This is a similar example but here **
     X=a                          ** the first 3 elements are unified with **
     Y=b                          ** variables X, Y and Z, leaving the **
     Z=c                          ** tail, T, as an empty list [] **
     T=[]
Yes
```

```
| ?- [a,b,c] = [a|[b|[c|[]]]].  ** Prolog is quite happy to unify these **
yes                             ** because they are just notational **
                                ** variants of the same Prolog term **
```

Conclusion:

Comment on explanation feature of decision taken by your prolog program.

Experiment No.8
Study and Implementation of Bayes Belief Network
Date of Performance:
Date of Submission:

Aim: Study and Implementation of Bayes Belief Network

Objective: To study about how to use Bayes Belief Network in reasoning process.

Theory:

Bayesian Belief Network or Bayesian Network or Belief Network is a Probabilistic Graphical Model (PGM) that represents conditional dependencies between random variables through a Directed Acyclic Graph (DAG). Bayesian Networks are applied in many fields. The main objective of these networks is trying to understand the structure of causality relations.

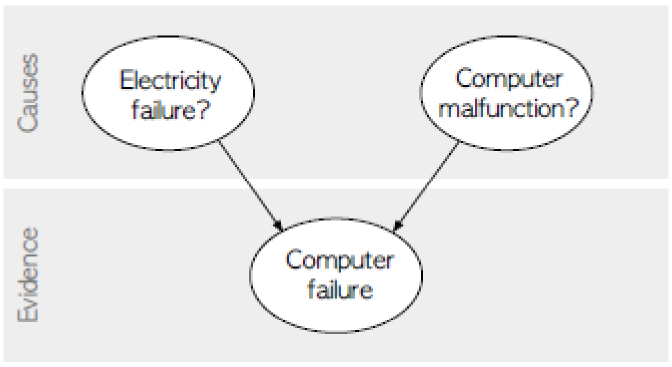
For example, disease diagnosis, optimized web search, spam filtering, gene regulatory networks, etc.

Bayesian Belief Network is a graphical representation of different probabilistic relationships among random variables in a particular set. It is a classifier with no dependency on attributes i.e it is condition independent. Due to its feature of joint probability, the probability in Bayesian Belief Network is derived, based on a condition — P(attribute/parent) i.e probability of an attribute, true over parent attribute.

A Bayesian network represents the causal probabilistic relationship among a set of random variables, their conditional dependences, and it provides a compact representationof a joint probability distribution. It consists of two major parts: a directed acyclic graph and a set of conditional probability distributions. The directed acyclic graph is a set of random variables represented by nodes. For health measurement, a node may be a health domain, and the states of the node would be the possible responses to that domain. If there exists a causal probabilistic dependence between two random variables in the graph, the corresponding two nodes are connected by a directed edge, while the directed edge from a node A to a node B indicates that the random variable A causes the random variable B. Since the directed edges represent a static causal probabilistic dependence, cycles are not allowed in the graph. A conditional probability distribution is defined for each node in the graph. In other words, the conditional probability distribution of a node (random variable) is defined for every possible outcome of the preceding causal node(s).

Example 1:

Suppose we attempt to turn on our computer, but the computer does not start (observation/evidence). We would like to know which of the possible causes of computer failure is more likely. In this simplified illustration, we assume only two possible causes of this misfortune: electricity failure and computer malfunction. The corresponding directed acyclic graph is depicted in figure.



The two causes in this banal example are assumed to be independent (there is no edge between the two causal nodes), but this assumption is not necessary in general. Unless there is a cycle in the graph, Bayesian networks are able to capture as many causal relations as it is necessary to credibly describe the real-life situation. Since a directed acyclic graph represents a hierarchical arrangement, it is unequivocal to use terms such as parent, child, ancestor, or descendant for certain node.

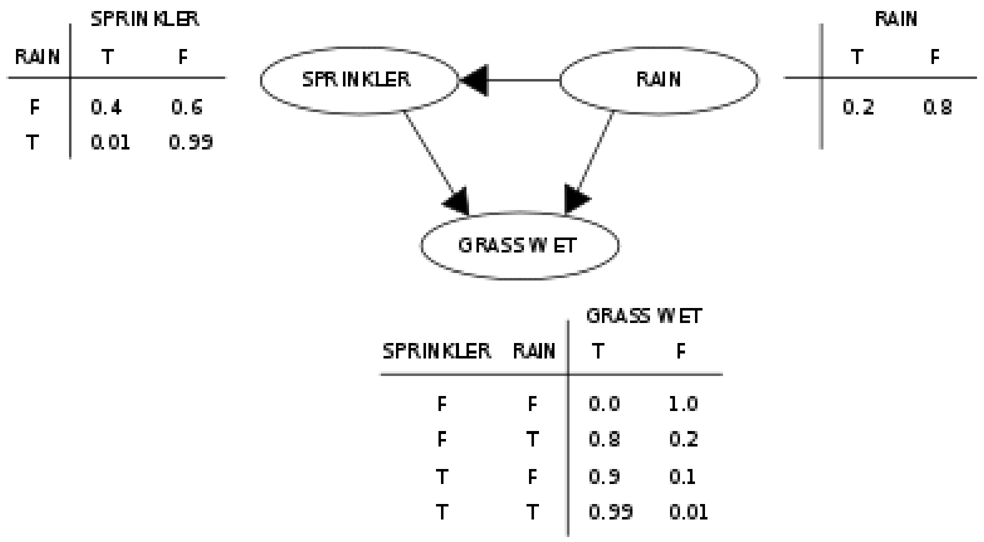
In figure, both electricity failure and computer malfunction are ancestors and parents of computer failure; analogically computer failure is a descendant and a child of both electricity failure and computer malfunction.

The goal is to calculate the posterior conditional probability distribution of each of the possible unobserved causes given the observed evidence, i.e. P [Cause | Evidence]. However, in practice we are often able to obtain only the converse conditional probability distribution of observing evidence given the cause, P [Evidence j Cause]. The whole concept of Bayesian networks is built on Bayes theorem, which helps us to express the conditional probability distribution of cause given the observed evidence using the converse conditional probability of observing evidence given the cause:

$$P[\text{Cause} \mid \text{Evidence}] = P[\text{Evidence} \mid \text{Cause}] \cdot \frac{P[\text{Cause}]}{P[\text{Evidence}]}$$

Any node in a Bayesian network is always conditionally independent of its all non-descendants given that node's parents. Hence, the joint probability distribution of all random variables in the graph factorizes into a series of conditional probability distributions of random variables given their parents. Therefore, we can build a full probability model by only specifying the conditional probability distribution in every node

Example 2: A Bayesian network with conditional probability tables



	SPRINKLER	
RAIN	T	F
F	0.4	0.6
T	0.01	0.99

RAIN	
T	F
0.2	0.8

		GRASS WET	
SPRINKLER	RAIN	T	F
F	F	0.0	1.0
F	T	0.8	0.2
T	F	0.9	0.1
T	T	0.99	0.01

Conclusion:

Comment on decision taken by your program for the taken problem.

Experiment No.9
Case Study on an Expert System in healthcare domain.
Date of Performance:
Date of Submission:

Aim: Case Study on an Expert System in healthcare domain.

Objective:

1. To develop an analysis and design ability in students to develop the AI applications in healthcare.
2. Also to develop technical writing skill in students.

Theory:

1. This assignment asks students to study and understood recent AI applications.

2. Write your own report on the design of Expert system application for healthcare domain.

Conclusion:

Comment on your AI healthcare system components.

Experiment No.10
Case Study on Natural Language Processing Application.
Date of Performance:
Date of Submission:

Aim: Case Study on Natural Language Processing Application.

Objective:

3. To develop an analysis and design ability in students to develop the real world NLP application.
4. Also to develop technical writing skill in students.

Theory:

1. This assignment asks students to study and understood recent AI applications.

2. Write your own report on the design components of NLP application system.

Conclusion:

Comment on your Components/steps used in NLP application.