1. Program on uninformed(BFS) search methods.

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

vector<int> BFS(int vertex, vector<pair<int, int>> edges)
{
    // Write your code here
    vector<int> vis(vertex+1,0);
    vector<int> bfs;
    vector<int> adj[vertex+1];
    for(int i=0;i<edges.size();i++){
        int u=edges[i].first;
        int v=edges[i].second;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    sort(adj->begin(),adj->end());

    for(int i=0;i<vertex;i++){
        if(!vis[i]){

            queue<int> q;
            q.push(i);
            vis[i]=1;

            while(!q.empty()){
                int node=q.front();
                q.pop();
                bfs.push_back(node);

                for(auto it:adj[node]){
                    if(!vis[it]){
                        q.push(it);
                        vis[it]=1;
                    }
                }
            }
        }
    }

    return bfs;
}

int main(){
    vector<pair<int, int>> edges={{0, 1}, {0, 3}, {1, 2}, {2, 3}};
    vector<int> bfs=BFS(4,edges);


    for(auto it:bfs){
      cout << it << " " ;
   }
}
```

2. Program on informed(A*)search methods.

```
class Graph:
 # example of adjacency list (or rather map)
 # adjacency_list = {
 # 'A': [('B', 1), ('C', 3), ('D', 7)],
 # 'B': [('D', 5)],
 # 'C': [('D', 12)]
 # }
 def __init__(self, adjacency_list):
 self.adjacency_list = adjacency_list
 def get_neighbors(self, v):
 return self.adjacency_list[v]
 # heuristic function with equal values for all nodes
 def h(self, n):
 H = {
 'A': 1,
 'B': 1,
 'C': 1,
 'D': 1
 }
 return H[n]
 def a_star_algorithm(self, start_node, stop_node):
 # open_list is a list of nodes which have been visited, but who's neighbors
 # haven't all been inspected, starts off with the start node
 # closed_list is a list of nodes which have been visited
 # and who's neighbors have been inspected
 open_list = set([start_node])
 closed_list = set([])
 # g contains current distances from start_node to all other nodes
 # the default value (if it's not found in the map) is +infinity
 g = {}
 g[start_node] = 0
 # parents contains an adjacency map of all nodes
 parents = {}
 parents[start_node] = start_node
 while len(open_list) > 0:
 n = None
 # find a node with the lowest value of f() - evaluation function
 for v in open_list:
 if n == None or g[v] + self.h(v) < g[n] + self.h(n):
 n = v;
 if n == None:
 print('Path does not exist!')
 return None
 # if the current node is the stop_node
 # then we begin reconstructin the path from it to the start_node
 if n == stop_node:
 reconst_path = []
 while parents[n] != n:
 reconst_path.append(n)
 n = parents[n]
 reconst_path.append(start_node)
 reconst_path.reverse()
 print('Path found: {}'.format(reconst_path))
 return reconst_path
 # for all neighbors of the current node do
 for (m, weight) in self.get_neighbors(n):
 # if the current node isn't in both open_list and closed_list
 # add it to open_list and note n as it's parent
 if m not in open_list and m not in closed_list:
 open_list.add(m)
 parents[m] = n
 g[m] = g[n] + weight
 # otherwise, check if it's quicker to first visit n, then m
 # and if it is, update parent data and g data
 # and if the node was in the closed_list, move it to open_list
 else:
 if g[m] > g[n] + weight:
 g[m] = g[n] + weight
 parents[m] = n
 if m in closed_list:
 closed_list.remove(m)
 open_list.add(m)
 # remove n from the open_list, and add it to closed_list
 # because all of his neighbors were inspected
 open_list.remove(n)
 closed_list.add(n)
 print('Path does not exist!')
 return None
adjacency_list = {
 'A': [('B', 1), ('C', 3), ('D', 7)],
 'B': [('D', 5)],
 'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

3. Write a program to implement Depth First Search

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

void dfs(int i, vector<bool> &vis, vector<int> adj[],vector<int> &store) {
    store.push_back(i);
    vis[i]=true;
    for(auto i:adj[i]){
        if(!vis[i]){
            dfs(i,vis,adj,store);
        }
    }
}

vector<vector<int>> depthFirstSearch(int V, int E, vector<vector<int>> &edges)
{
        vector<int> adj[V+1];
        for(int i=0;i<edges.size();i++){
            int u=edges[i][0];
            int v=edges[i][1];
            adj[u].push_back(v);
            adj[v].push_back(u);
        }

        vector<bool> visited(V,false);
        vector<vector<int>> storeDfs;

        for(int i=0;i<V;i++){
            if(!visited[i]){
                vector<int> nodes;
                dfs(i,visited,adj,nodes);
                storeDfs.push_back(nodes);
            }
        }
        return storeDfs;
}
int main(){
    int v=5;
    int e=4;
    vector<vector<int>> edges={{0,2},{0,1},{1,2},{3,4}};
    depthFirstSearch(v,e,edges);
}
```

4. Program for N queen Algorithm using Back tracking

```
# Python program to solve N Queen
# Problem using backtracking
global N
N = 4
def printSolution(board):
 for i in range(N):
 for j in range(N):
 print (board[i][j],end=' ')
 print()
# A utility function to check if a queen can
# be placed on board[row][col]. Note that this
# function is called when "col" queens are
# already placed in columns from 0 to col -1.
# So we need to check only left side for
# attacking queens
def isSafe(board, row, col):
 # Check this row on left side
 for i in range(col):
 if board[row][i] == 1:
 return False
 # Check upper diagonal on left side
 for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
 if board[i][j] == 1:
 return False
 # Check lower diagonal on left side
 for i, j in zip(range(row, N, 1), range(col, -1, -1)):
 if board[i][j] == 1:
 return False
 return True
def solveNQUtil(board, col):
 # base case: If all queens are placed
 # then return true
 if col >= N:
 return True
 # Consider this column and try placing
 # this queen in all rows one by one
 for i in range(N):
 if isSafe(board, i, col):
 # Place this queen in board[i][col]
 board[i][col] = 1
 # recur to place rest of the queens
 if solveNQUtil(board, col + 1) == True:
 return True
 # If placing queen in board[i][col
 # doesn't lead to a solution, then
 # queen from board[i][col]
 board[i][col] = 0
 # if the queen can not be placed in any row in
 # this column col then return false
 return False
# This function solves the N Queen problem using
# Backtracking. It mainly uses solveNQUtil() to
# solve the problem. It returns false if queens
# cannot be placed, otherwise return true and
# placement of queens in the form of 1s.
# note that there may be more than one
# solutions, this function prints one of the
# feasible solutions.
def solveNQ():
 board = [ [0, 0, 0, 0],
 [0, 0, 0, 0],
 [0, 0, 0, 0],
 [0, 0, 0, 0]
 ]
 if solveNQUtil(board, 0) == False:
 print ("Solution does not exist")
 return False
 printSolution(board)
 return True
# driver program to test above function
solveNQ()
```

5. Implementation for Bayes Belief Network
   Suppose we know the following probabilities:
   • P(rain) = 0.20
   • P(cloudy) = 0.40
   • P(cloudy | rain) = 0.85

   Write the program to calculate To calculate P(rain | cloudy).

```
# Define the probabilities
P_rain = 0.20
P_cloudy = 0.40
P_cloudy_given_rain = 0.85

# Calculate P(rain and cloudy)
P_rain_and_cloudy = P_cloudy_given_rain * P_rain

# Calculate P(cloudy)
P_not_rain = 1 - P_rain
P_cloudy_given_not_rain = 1 - P_cloudy_given_rain
P_not_rain_and_cloudy = P_cloudy_given_not_rain * P_not_rain
```

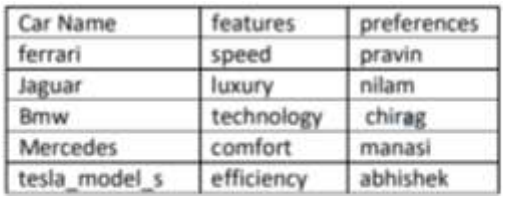
```
P_cloudy_total = P_rain_and_cloudy + P_not_rain_and_cloudy

# Calculate P(rain | cloudy)
P_rain_given_cloudy = P_rain_and_cloudy / P_cloudy_total

print("P(rain | cloudy) =", P_rain_given_cloudy)

# P(rain | cloudy) = 0.46875
```

6. Write Prolog code for following Data set



Car Name	features	preferences
ferrari	speed	pravin
Jaguar	luxury	nilam
Bmw	technology	chirag
Mercedes	comfort	manasi
tesla_model_s	efficiency	abhishek

```
% Define cars
car(ferrari).
car(jaguar).
car(bmw).
car(mercedes).
car(tesla_model_s).

% Define car features
feature(ferrari, speed).
feature(jaguar, luxury).
feature(bmw, technology).
feature(mercedes, comfort).
feature(tesla_model_s, efficiency).

% Define car preferences
preference(speed, pravin).
preference(luxury, nilam).
preference(technology, chirag).
preference(comfort, manasi).
preference(efficiency, abhishek).

% Define a rule for finding a person's preferred car
preferred_car(Person, Car) :-
 preference(Feature, Person), % Get the person's preferred feature
 feature(Car, Feature). % Find a car that has that feature
```

7. Write code for Planning Programming

```
tab = []
result = []
goalList = ["a", "b", "c", "d", "e"]
def parSolution(N):
 for i in range(N):
 if goalList[i] != result[i]:
 return False
 return True
def Onblock(index, count):
 if count == len(goalList)+1:
 return True
 block = tab[index]
 result.append(block)
 print(result)
 if parSolution(count):
 print("Pushed a result solution ")
 # remove block from tab
 tab.remove(block)
 Onblock(0, count + 1)
 else:
 print("result solution not possible, back to the tab")
 # pop out if no partial solution
 result.pop()
 Onblock(index+1, count)
def Ontab(problem):
 # check if everything in stack is on the tab
 if len(problem) != 0:
 tab.append(problem.pop())
 Ontab(problem)
 # if everything is on the tab the we return true
 else:
 return True
def goal_stack_planing(problem):
 # pop problem and put in tab
 Ontab(problem)
 # print index and number of blocks on result stack
 if Onblock(0, 1):
 print(result)
if __name__ == "__main__":
 problem = ["c", "a", "e", "d", "b"]
 print("Goal Problem")
 for k, j in zip(goalList, problem):
 print(k+" "+j)
 goal_stack_planing(problem)
 print("result Solution")
 print(result)
```

8. Write code for program to find maximum score that maximizing player can get.

```
# A simple Python3 program to find
# maximum score that
# maximizing player can get
import math
def minimax (curDepth, nodeIndex,
maxTurn, scores,
targetDepth):
# base case :targetDepth reached
if (curDepth == targetDepth):
return scores[nodeIndex]
if (maxTurn):
return max(minimax(curDepth + 1, nodeIndex * 2,
False, scores, targetDepth),
minimax(curDepth + 1, nodeIndex * 2 + 1,
False, scores, targetDepth))
else:
return min(minimax(curDepth + 1, nodeIndex * 2,
True, scores, targetDepth),
minimax(curDepth + 1, nodeIndex * 2 + 1,
True, scores, targetDepth))
# Driver code
scores = [3, 5, 2, 9, 12, 5, 23, 23]
treeDepth = math.log(len(scores), 2)
print("The optimal value is : ", end = "")
print(minimax(0, 0, True, scores, treeDepth))
```

9. Write the PEAS for Part Picking Robot.

   - Performance Measure: The performance of the Part Picking Robot is measured by the number of correct parts picked and placed in the correct location per unit time.

   - Environment: The environment of the Part Picking Robot consists of a warehouse or factory where parts are stored, a conveyor belt or similar mechanism to transport parts to the robot, a loading dock or similar mechanism for the robot to place the picked parts, and a computer system or other interface for the robot to receive instructions and send status updates.

   - Actuators: The actuators of the Part Picking Robot include a robotic arm with a gripper, sensors to detect the position of the conveyor belt and parts, and a communication interface to send and receive instructions.

   - Sensors: The sensors of the Part Picking Robot include cameras or other vision sensors to detect the location and orientation of the parts, proximity sensors or other sensors to detect the position of the conveyor belt and loading dock, and communication interfaces to receive instructions and send status updates.

   - Goal: The goal of the Part Picking Robot is to efficiently and accurately pick parts from the conveyor belt and place them in the correct location on the loading dock, following the instructions provided by the computer system or other interface.

10. Write PEAS for Automation CAR

    - Performance Measure: The performance of the Automation Car is measured by its efficiency in completing its tasks, such as the number of trips made per unit time, the time taken to complete each trip, and the fuel efficiency.

    - Environment: The environment of the Automation Car consists of roads, traffic signals, other vehicles, pedestrians, and various obstacles. The car receives inputs from sensors such as cameras, LIDAR, and radar, and communicates with traffic control systems, navigation systems, and other cars on the road.

    - Actuators: The actuators of the Automation Car include the steering system, accelerator and brake pedals, gear shift, headlights, turn signals, and other controls. These are controlled by a computer system that receives input from various sensors.

    - Sensors: The sensors of the Automation Car include cameras, LIDAR, radar, ultrasonic sensors, and other sensors to detect obstacles, traffic signals, road markings, and other vehicles. The car may also have GPS and other navigation sensors to determine its location and direction.

- Goal: The goal of the Automation Car is to efficiently and safely transport passengers and cargo to their destinations, while obeying traffic rules, avoiding obstacles, and minimizing fuel consumption. The car should be able to handle different driving scenarios, such as highways, city streets, and rural roads, and adapt to changing traffic and road conditions.

11. Write PEAS for Chess

- Performance Measure: The performance of the Chess AI is measured by its ability to win games against human or computer opponents, and the quality of its moves. The AI should strive to maximize the number of games won, while minimizing the number of losses and draws.

- Environment: The environment of the Chess AI consists of a chessboard, pieces, and a set of rules governing the movement and capture of the pieces. The AI receives input from a human or computer opponent who makes their moves in turn, and responds with its own moves. The AI may also have access to databases of previous chess games and opening variations.

- Actuators: The actuators of the Chess AI include selecting and moving the pieces on the chessboard, as well as performing other actions such as resigning or offering a draw. The AI may also have access to other game-related functions, such as recording moves and time management.

- Sensors: The sensors of the Chess AI include a representation of the current state of the chessboard and the positions of the pieces, as well as the moves made by the opponent. The AI may also have access to other data sources such as databases of opening variations, endgame tables, and evaluation functions.

- Goal: The goal of the Chess AI is to win the game by placing the opponent's king in checkmate, while avoiding its own checkmate. The AI should be able to play at various levels of difficulty, from beginner to advanced, and adapt to the opponent's playing style and strategy. The AI should also be able to learn and improve over time, by analyzing previous games and identifying weaknesses in its own play.

12. Write PEAS for Wumpus world Problem

- Performance Measure: The performance of the agent in the Wumpus world problem is measured by its ability to find the gold and exit the world safely. The agent should strive to maximize its score, which is calculated based on the number of actions taken, the amount of time spent, and the risks encountered.

The agent should also minimize its penalty, which is incurred for actions such as firing an arrow, falling into a pit, or being killed by the Wumpus.

- Environment: The environment of the Wumpus world problem consists of a two-dimensional grid of cells, each of which can be either safe, contain a pit, contain gold, or be occupied by the Wumpus. The agent can move in the four cardinal directions and fire an arrow in a straight line. The agent receives input from its sensors, which provide information about the adjacent cells, such as the presence of a breeze, a stench, or glitter.

- Actuators: The actuators of the agent in the Wumpus world problem include moving to an adjacent cell, firing an arrow, picking up the gold, and exiting the world. The agent may also choose to do nothing, wait for a certain amount of time, or take some other action depending on the state of the world.

- Sensors: The sensors of the agent in the Wumpus world problem include a percept that consists of the presence or absence of a breeze, a stench, or glitter in the current cell. The agent can also perceive the state of the adjacent cells, such as whether they contain a pit, the Wumpus, or gold.

13. Draw the State space for Vacuum world

```
   (1,1) = Dirty       (2,1) = Dirty
                 |            |
                 |            |
(1,2) = Dirty --------|------- (2,2) = Dirty
                 |            |
                 |            |
          (1,1) = Clean      (2,1) = Clean
```

In this diagram, the two-dimensional grid represents the two locations of the vacuum cleaner, with the x-axis indicating the horizontal position and the y-axis indicating the vertical position. Each location can be in one of two possible states: clean or dirty. The possible states of the vacuum cleaner and the environment are as follows:

(1,1) = Dirty, (1,2) = Dirty
(1,1) = Dirty, (1,2) = Clean
(1,1) = Clean, (1,2) = Dirty
(1,1) = Clean, (1,2) = Clean
(2,1) = Dirty, (1,2) = Dirty
(2,1) = Dirty, (1,2) = Clean
(2,1) = Clean, (1,2) = Dirty
(2,1) = Clean, (1,2) = Clean

Each state is a combination of the locations of the vacuum cleaner and the environment. For example, the first state corresponds to the vacuum cleaner being in location (1,1) and both locations being dirty. The state space can be used to represent the possible transitions between states and to find a solution to the Vacuum World problem using search algorithms such as DFS, BFS, or A*.

14. Give Environment types for Taxi driving

- City environment: A busy city environment with multiple roads, intersections, traffic lights, and other vehicles on the road. This type of environment can be challenging for a taxi driver to navigate due to heavy traffic and congestion.

- Rural environment: A rural environment with winding roads and fewer vehicles on the road. This type of environment can be less challenging than a city environment, but the driver needs to be careful of potential hazards such as animals crossing the road.

- Mountainous environment: A mountainous environment with steep inclines and narrow roads. This type of environment requires a skilled driver with good control over the vehicle.

- Extreme weather environment: An environment with extreme weather conditions such as heavy rain, snow, or fog. This type of environment requires the driver to exercise caution and adjust their driving style accordingly.

- Nighttime environment: A nighttime environment with limited visibility and low light conditions. This type of environment requires the driver to be extra careful and use headlights to navigate.

- Tourist destination environment: An environment with multiple tourist attractions, which can lead to heavy traffic and congested roads. This type of environment requires the driver to be patient and skilled in navigating through crowds of pedestrians and other vehicles.

15. Give Environment types for Cross word

- Physical environment: A traditional crossword puzzle that is printed on paper and solved with a pen or pencil. This type of environment is familiar to most people and can be completed anywhere, such as at home, on a plane, or in a coffee shop.

- Digital environment: A crossword puzzle that is available on a digital platform such as a computer, tablet, or smartphone. This type of environment can be more convenient than a physical puzzle as it can be accessed from anywhere with an internet connection.

- Collaborative environment: A crossword puzzle that is completed with a partner or in a group. This type of environment can be social and collaborative, with participants working together to solve clues and fill in the grid.

- Competitive environment: A crossword puzzle competition where participants compete to solve the puzzle in the fastest time or with the highest accuracy. This type of environment can be challenging and rewarding for skilled puzzle solvers.

- Language-specific environment: A crossword puzzle that is specific to a particular language, such as English, Spanish, or French. This type of environment can help language learners practice their vocabulary and comprehension skills.

- Educational environment: A crossword puzzle that is designed to teach or reinforce a specific subject or topic, such as history, science, or literature. This type of environment can be useful for students and educators as a tool for learning and assessment.

16. Give Environment types for Chess

- Physical environment: A traditional chess board and pieces set up on a table, with players sitting across from each other. This type of environment is familiar to most chess players and can be played anywhere with a flat surface.

- Digital environment: A digital chess board that can be accessed on a computer, tablet, or smartphone. This type of environment can be convenient for playing online or practicing against computer opponents.

- Competitive environment: A chess tournament or competition where players compete against each other for a prize or recognition. This type of environment can be challenging and rewarding for skilled chess players.

- Training environment: A chess lesson or training session where players work on improving their skills and strategies. This type of environment can be useful for beginners and advanced players alike.

- Educational environment: A chess lesson or program that is designed to teach chess to children or students. This type of environment can be useful for developing critical thinking skills and problem-solving abilities.

- Virtual reality environment: A virtual reality chess game that simulates a physical chess board and allows players to move the pieces and interact with the game in a more immersive way. This type of environment can be fun and engaging for players who enjoy exploring new technologies.