PROJECT REPORT

Artificial Intelligence

Guided by:

Dr. Sanjeev Anand Sahu

Assistant Professor, Applied Mathematics, IIT(ISM) Dhanbad

Presented by-

Ayush Somani

16JE001852

Int. M. Tech. (Mathematics and Computing)

ACKNOWLEDGEMENT

I would like to express my special thanks of gratitude to my Mentor, Dr. S. A. Sahu sir (assistant professor, dept. of Applied Mathematics, IIT(ISM) Dhanbad) who gave me the golden opportunity to do this interesting and knowledgeable project on Artificial Intelligence, which also helped me in doing a lot of Research and I came to know about so many new things, I am really thankful to him.

Ayush Somani

Index

INTRODUCTION

According to the father of Artificial Intelligence, John McCarthy, it is “The science and engineering of making intelligent machines, especially intelligent computer programs”.

Artificial Intelligence is a way of making a computer, a computer-controlled robot, or a software think intelligently, in the similar manner the intelligent humans think.

AI is accomplished by studying how human brain thinks, and how humans learn, decide, and work while trying to solve a problem, and then using the outcomes of this study as a basis of developing intelligent software and systems.

Artificial Intelligence is the study of ideas that enable computers to be intelligent.

Artificial Intelligence is the part of computer science concerned with design of computer systems that exhibit human intelligence (From the Concise Oxford Dictionary).

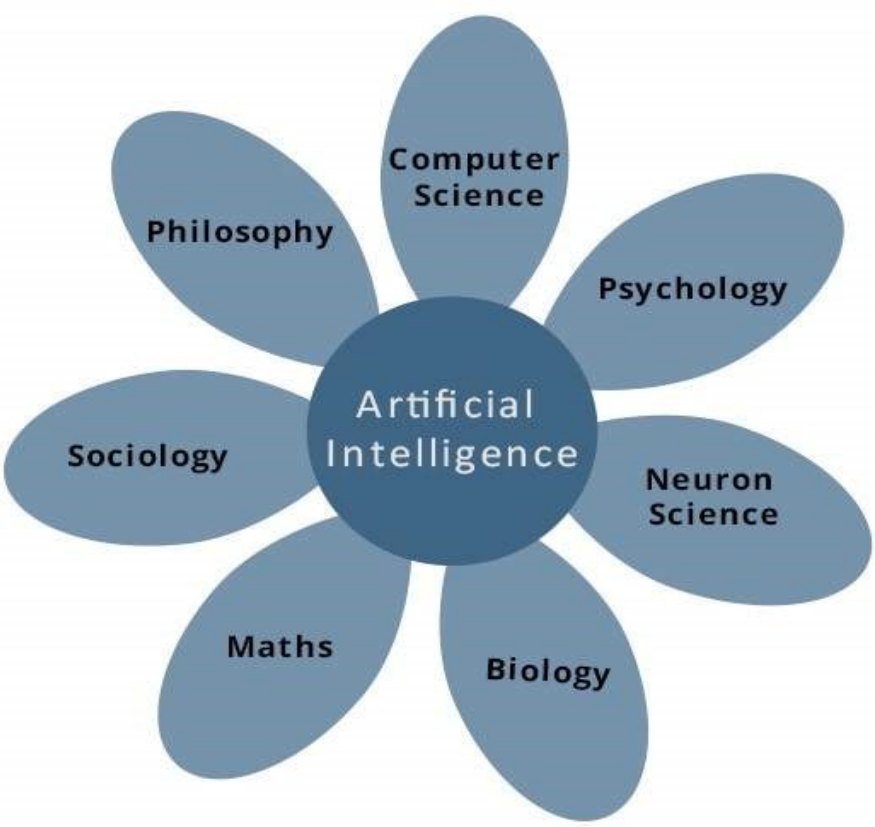
Artificial Intelligence will define the next generation of software solutions. It can be used to build smart apps that help organizations be more efficient and enrich people’s lives.

GOALS OF AI

To Create Expert Systems − The systems which exhibit intelligent behaviour, learn, demonstrate, explain, and advice its users.

To Implement Human Intelligence in Machines − Creating systems that understand, think, learn, and behave like humans.

Out of the following areas, one or multiple areas can contribute to build an intelligent system.



GOAL TREES

One of the most important features of human intelligence is reasoning. If we really want to make our systems mimic human intelligence, they must be capable of reasoning and problem solving.

The method for doing this is *GOAL TREES.*

GOAL TREE: - A Goal Tree, sometimes still referred to as Intermediate Objective Map or IO Map, is primarily a Logical Thinking Process tool, itself linked to the Theory of Constraints. The top of the tree deals with strategic planning while going down to its bottom links strategy to operations.

The root of the goal tree contains the goal that need to be achieved.

The next level contains the necessary condition to bring about the goal.

Now each next level breaks the complex goal into simpler steps which can be used to fulfil the goal. This is a typical goal tree.

RULE BASED EXPERT SYSTEM

In artificial intelligence, an expert system is a computer system that emulates the decision-making ability of a human expert. Expert systems are designed to solve complex problems by reasoning about knowledge, represented mainly as if–then rules rather than through conventional procedural code.
These mimic human's decision-making abilities based on specific predefined rules.

SEARCHING TECHNIQUES

Searching is the universal technique of problem solving in AI. There are some single-player games such as tile games, Sudoku, crossword, etc. The search algorithms help you to search for a particular position in such games.

Single Agent Pathfinding Problems

The games such as 3X3 eight-tile, 4X4 fifteen-tile, and 5X5 twenty four tile puzzles are single-agent-path-finding challenges. They consist of a matrix of tiles with a blank tile. The player is required to arrange the tiles by sliding a tile either vertically or horizontally into a blank space with the aim of accomplishing some objective.
The other examples of single agent pathfinding problems are Travelling Salesman Problem, Rubik's Cube, and Theorem Proving.

Search Terminology

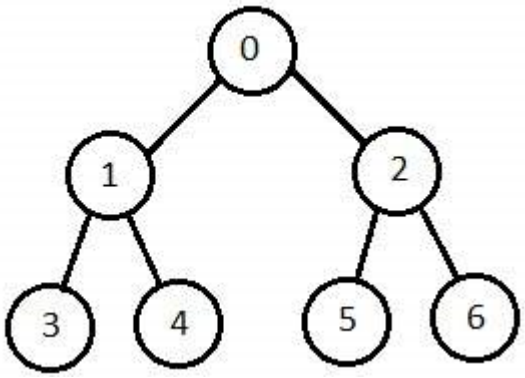
- **Problem Space** − It is the environment in which the search takes place. (A set of states and set of operators to change those states)
- **Problem Instance** − It is Initial state + Goal state.
- **Problem Space Graph** − It represents problem state. States are shown by nodes and operators are shown by edges.
- **Depth of a problem** − Length of a shortest path or shortest sequence of operators from Initial State to goal state.
- **Space Complexity** − The maximum number of nodes that are stored in memory.
- **Time Complexity** − The maximum number of nodes that are created.
- **Admissibility** − A property of an algorithm to always find an optimal solution.
- **Branching Factor** − The average number of child nodes in the problem space graph.
- **Depth** − Length of the shortest path from initial state to goal state.

1.Breadth-First Search

It starts from the root node, explores the neighboring nodes first and moves towards the next level neighbors. It generates one tree at a time until the solution is found. It can be implemented using FIFO queue data structure. This method provides shortest path to the solution.
If **branching factor** (average number of child nodes for a given node) = b and depth = d, then number of nodes at level d = b^d.
The total no of nodes created in worst case is $b + b^2 + b^3 + \ldots + b^d$.

Disadvantage − Since each level of nodes is saved for creating next one, it consumes a lot of memory space. Space requirement to store nodes is exponential.

Its complexity depends on the number of nodes. It can check duplicate nodes.



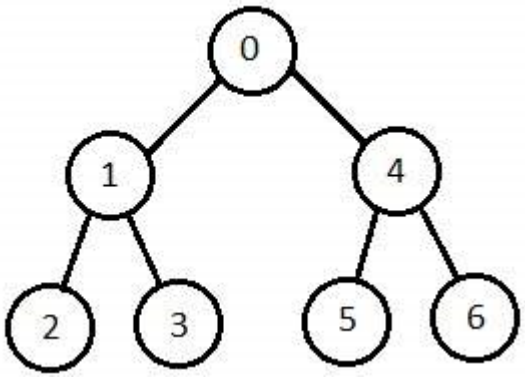
2.Depth-First Search

It is implemented in recursion with LIFO stack data structure. It creates the same set of nodes as Breadth-First method, only in the different order.

As the nodes on the single path are stored in each iteration from root to leaf node, the space requirement to store nodes is linear. With branching factor *b*and depth as *m*, the storage space is *bm.*

Disadvantage − This algorithm may not terminate and go on infinitely on one path. The solution to this issue is to choose a cut-off depth. If the ideal cut-off is *d*, and if chosen cut-off is lesser than *d*, then this algorithm may fail. If chosen cut-off is more than *d*, then execution time increases.

Its complexity depends on the number of paths. It cannot check duplicate nodes.



3.Uniform Cost Search

Sorting is done in increasing cost of the path to a node. It always expands the least cost node. It is identical to Breadth First search if each transition has the same cost.

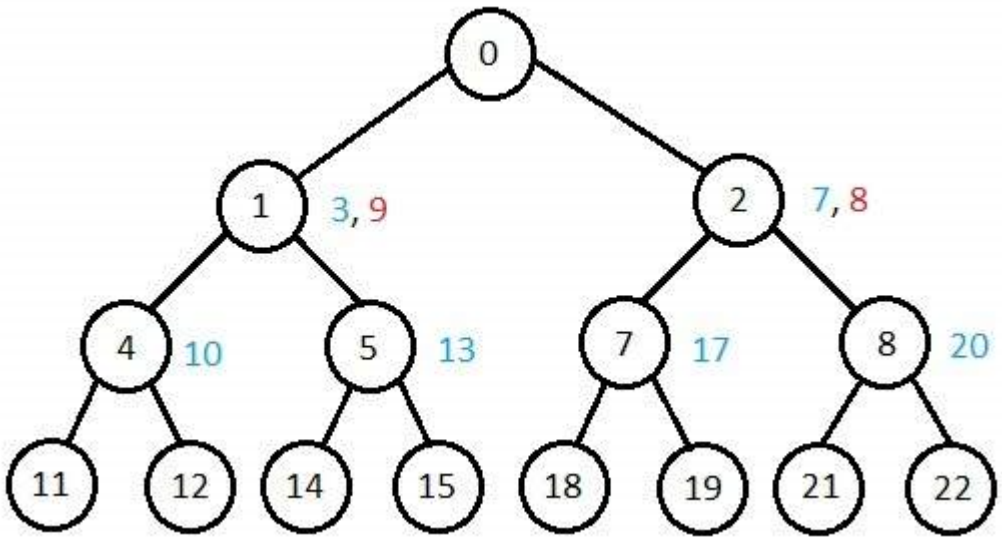
It explores paths in the increasing order of cost.

Disadvantage − There can be multiple long paths with the cost ≤ C*. Uniform Cost search must explore them all.

4.Iterative Deepening Depth-First Search

It performs depth-first search to level 1, starts over, executes a complete depth-first search to level 2, and continues in such way till the solution is found.

It never creates a node until all lower nodes are generated. It only saves a stack of nodes. The algorithm ends when it finds a solution at depth *d*. The number of nodes created at depth *d* is b^d and at depth *d-1* is b^{d-1}.



Comparison of Various Algorithms Complexities

Let us see the performance of algorithms based on various criteria −

Criterion	Breadth First	Depth First	Uniform Cost	Interactive Deepening
Time	b^d	b^m	b^d	b^d
Space	b^d	b^m	b^d	b^d
Optimality	Yes	No	Yes	Yes
Completeness	Yes	No	Yes	Yes

Informed (Heuristic) Search Strategies

To solve large problems with large number of possible states, problem-specific knowledge needs to be added to increase the efficiency of search algorithms.

Heuristic Evaluation Functions

They calculate the cost of optimal path between two states. A heuristic function for sliding-tiles games is computed by counting number of moves that each tile makes from its goal state and adding these number of moves for all tiles.

1.A * Search

It is best-known form of Best First search. It avoids expanding paths that are already expensive, but expands most promising paths first.

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

- g(n) the cost (so far) to reach the node
- h(n) estimated cost to get from the node to the goal

- f(n) estimated total cost of path through n to goal. It is implemented using priority queue by increasing f(n).

2.Hill-Climbing Search

It is an iterative algorithm that starts with an arbitrary solution to a problem and attempts to find a better solution by changing a single element of the solution incrementally. If the change produces a better solution, an incremental change is taken as a new solution. This process is repeated until there are no further improvements.

function Hill-Climbing (problem), returns a state that is a local maximum.

```
inputs: problem, a problem
local variables: current, a node
                 neighbor, a node
current <-Make_Node(Initial-State[problem])
loop
  do neighbor <- a highest_valued successor of current
    if Value[neighbor] ≤ Value[current] then
    return State[current]
    current <- neighbor

end
```

Disadvantage − This algorithm is neither complete, nor optimal.

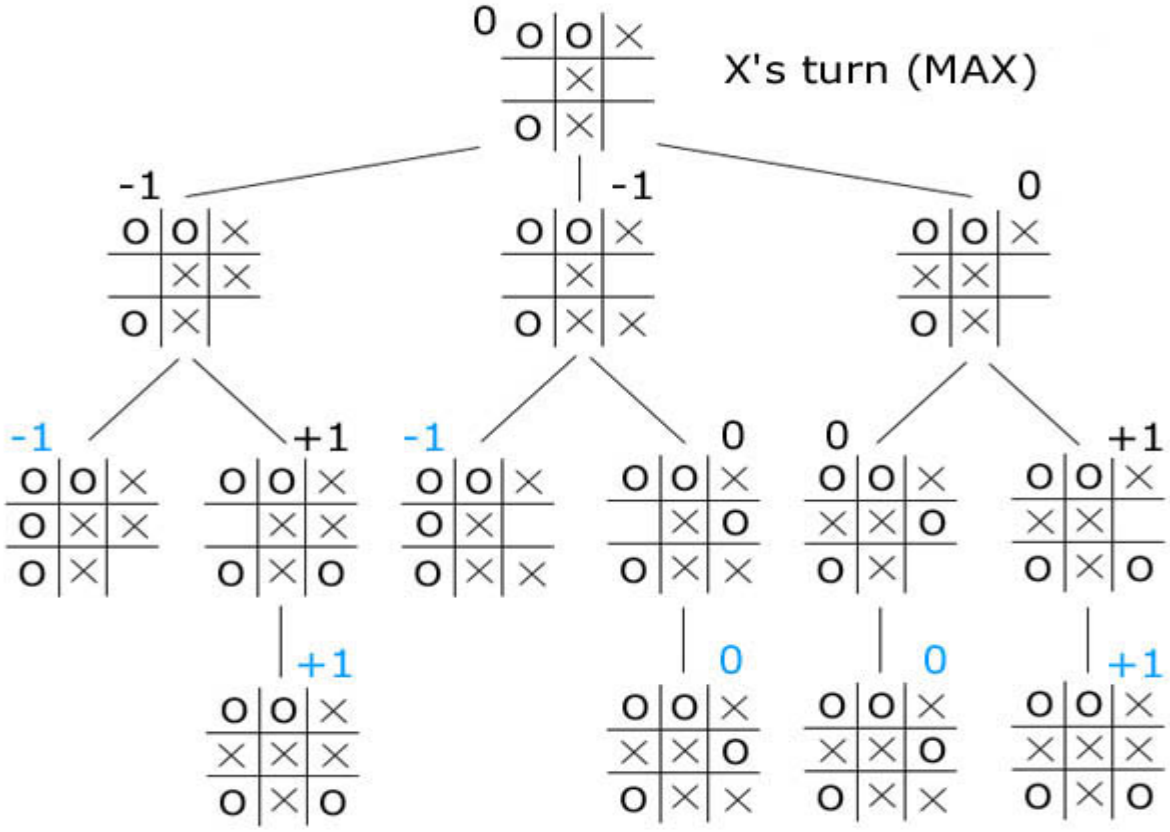
3.Local Beam Search

In this algorithm, it holds k number of states at any given time. At the start, these states are generated randomly. The successors of these k states are computed with the help of objective function. If any of these successors is the maximum value of the objective function, then the algorithm stops.

Otherwise the (initial k states and k number of successors of the states = 2k) states are placed in a pool. The pool is then sorted numerically. The highest k states are selected as new initial states. This process continues until a maximum value is reached.

GAME TREE

A game tree is a directed graph whose nodes are positions in a game and whose edges are moves.
These trees help in game theory and allow the computer to play efficiently like humans.
For example: Game tree for a tic tac toe game can be represented as given below:-

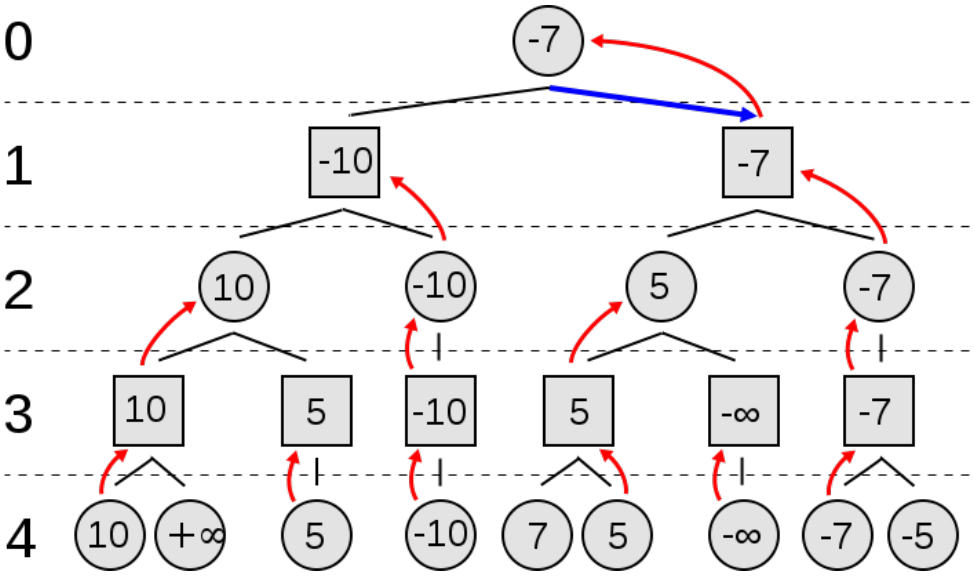


There are certain search algorithms that helps us in deciding the next optimal move for the player.
These are namely:-

- MINIMAX ALGORITHM
- ALPHA-BETA PRUNING

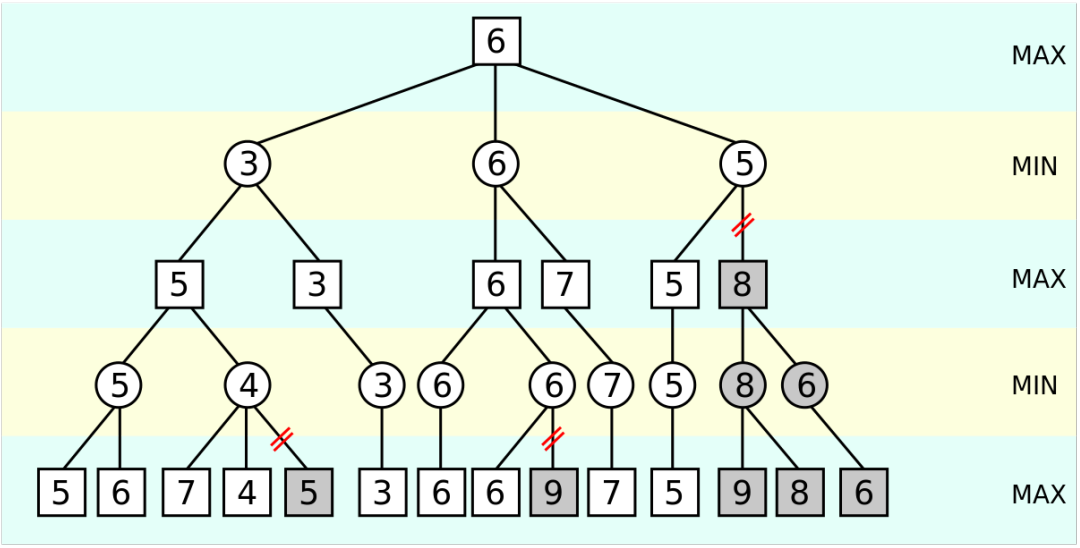
1.MINIMAX ALGORITHM

A minimax algorithm is a recursive algorithm for choosing the next move in an n-player game, usually a two-player game. A value is associated with each position or state of the game.



2.ALPHA-BETA PRUNING

Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree.



Constraint Satisfaction

These include problems which can be represented as a set of variables that can take values from a certain domain D and a set of constraint C.The solution to such problems is one in which all the variables have a value and also satisfy all the constraints.

Eg.: Map Colouring Problem

Set of variable x:{set of regions on the map}

Domain:{set of clours}

Set of constraints:{colour of neighbouring regions must not be same}

TYPES OF CONSTRAINTS:-

UNARY: When only one variable is involved in constraint i.e. x=Di where x is some variable and Di belongs to domain.

BINARY: When two variables are involved i.e. X1 != X where X1 and X2 both are variables.

HIGHER ORDER: When number of variables is more than 2. eg. cryptarithmetic

The constraint networks working on binary constraints are called ***binary constraint networks.***

CONSTRAINT PROPAGATION:-

The constraint propagation algorithm proceeds as follows. When a given variable is assigned a value, either directly by the user or by the system, the algorithm recomputes the possible value sets and assigned values of all its dependent variables. This process continues recursively until there are no more changes in the network. More specifically, when a variable X changes its value, the system evaluates the domain expression of each variable Y dependent on X. This may generate a new set of possible values for Y. If this set changes, the preference constraint is evaluated selecting one of the possible values as the new assigned value for Y. If this assigned value is different from the previous one, it causes the system to recompute the values for further downstream variables. Values that have been assigned by the user are always preferred as long as they are consistent with the constraints.

ARC CONSISTENCY ALGORITHM:-

A binary constraint is arc-consistent if every value of one variable has a value of the second variable such that they satisfy the constraint.

There are certain algorithms which help in checking consistency of arcs:-

AC1 algorithm

REVISE$((x_i), x_j)$

input: a subnetwork defined by two variables $X = \{x_i, x_j\}$, a distinguished variable x_i, domains: D_i and D_j, and constraint R_{ij}
output: D_i, such that, x_i arc-consistent relative to x_j

1. **for** each $a_i \in D_i$
2. **if** there is no $a_j \in D_j$ such that $(a_i, a_j) \in R_{ij}$
3. **then** delete a_i from D_i
4. **endif**
5. **endfor**

AC-1$(\mathcal{R})$

input: a network of constraints $\mathcal{R} = (X, D, C)$
output: $\mathcal{R}'$ which is the loosest arc-consistent network equivalent to $\mathcal{R}$

1. **repeat**
2. **for** every pair $\{x_i, x_j\}$ that participates in a constraint
3. Revise$((x_i), x_j)$ (or $D_i \leftarrow D_i \cap \pi_i(R_{ij} \bowtie D_j)$)
4. Revise$((x_j), x_i)$ (or $D_j \leftarrow D_j \cap \pi_j(R_{ij} \bowtie D_i)$)
5. **endfor**
6. **until** no domain is changed

AC3 Algorithm:-

REVISE$((x_i), x_j)$

input: a subnetwork defined by two variables $X = \{x_i, x_j\}$, a distinguished variable x_i, domains: D_i and D_j, and constraint R_{ij}
output: D_i, such that, x_i arc-consistent relative to x_j

1. **for** each $a_i \in D_i$
2. **if** there is no $a_j \in D_j$ such that $(a_i, a_j) \in R_{ij}$
3. **then** delete a_i from D_i
4. **endif**
5. **endfor**

AC-3$(\mathcal{R})$

input: a network of constraints $\mathcal{R} = (X, D, C)$
output: $\mathcal{R}'$ which is the largest arc-consistent network equivalent to $\mathcal{R}$

1. **for** every pair $\{x_i, x_j\}$ that participates in a constraint $R_{ij} \in \mathcal{R}$
2. $queue \leftarrow queue \cup \{(x_i, x_j), (x_j, x_i)\}$
3. **endfor**
4. **while** $queue \neq \{\}$
5. select and delete (x_i, x_j) from $queue$
6. $Revise((x_i), x_j)$
7. **if** $Revise((x_i), x_j)$ causes a change in D_i
8. **then** $queue \leftarrow queue \cup \{(x_k, x_i), i \neq k\}$
9. **endif**
10. **endwhile**

PROGRAMS

8-Square Puzzle

Code:-

```
#include <bits/stdc++.h>
using namespace std;
#define N 3
struct Node
{
   Node* parent;
   int mat[N][N];
   int x, y;
   int cost;
   int level;
};
int printMatrix(int mat[N][N])
{
   for (int i = 0; i < N; i++)
   {
      for (int j = 0; j < N; j++)
         printf("%d ", mat[i][j]);
      printf("\n");
   }
}
Node* newNode(int mat[N][N], int x, int y, int newX,
          int newY, int level, Node* parent)
{
   Node* node = new Node;
   node->parent = parent;
   memcpy(node->mat, mat, sizeof node->mat);
   swap(node->mat[x][y], node->mat[newX][newY]);
   node->cost = INT_MAX;
   node->level = level;
   node->x = newX;
   node->y = newY;
   return node;
}
int row[] = { 1, 0, -1, 0 };
int col[] = { 0, -1, 0, 1 };
int calculateCost(int initial[N][N], int final[N][N])
{
   int count = 0;
   for (int i = 0; i < N; i++)
     for (int j = 0; j < N; j++)
       if (initial[i][j] && initial[i][j] != final[i][j])
          count++;
   return count;
}
int isSafe(int x, int y)
{
   return (x >= 0 && x < N && y >= 0 && y < N);
}
void printPath(Node* root)
{
   if (root == NULL)
      return;
   printPath(root->parent);
   printMatrix(root->mat);

   printf("\n");
}
struct comp
{
```

```
    bool operator()(const Node* lhs, const Node* rhs) const
    {
        return (lhs->cost + lhs->level) > (rhs->cost + rhs->level);
    }
};
void solve(int initial[N][N], int x, int y,int final[N][N])
{
    priority_queue<Node*, std::vector<Node*>, comp> pq;
    Node* root = newNode(initial, x, y, x, y, 0, NULL);
    root->cost = calculateCost(initial, final);
    pq.push(root);
    while (!pq.empty())
    {
        Node* min = pq.top();
        pq.pop();
        if (min->cost == 0)
        {
            printPath(min);
            return;
        }


        for (int i = 0; i < 4; i++)
        {
            if (isSafe(min->x + row[i], min->y + col[i]))
            {
                Node* child = newNode(min->mat, min->x,min->y, min->x + row[i],min->y + col[i],min->level + 1, min);
                child->cost = calculateCost(child->mat, final);
                pq.push(child);
            }
        }
    }
}
int main()
{
    int initial[N][N];
    int final1[N][N];
    int x;int y;
    cout<<"Enter initial state 0 for empty space\n";
    for(int i=0;i<N;i++)
        for(int j=0;j<N;j++)
        {
            cin>>initial[i][j];
            if(initial[i][j]==0)
            {
                x=i;
                y=j;
            }
        }
    cout<<"Enter the final state \n";
    for(int i=0;i<N;i++)
        for(int j=0;j<N;j++)
            cin>>final1[i][j];
    cout<<"\n";
    solve(initial, x, y, final1);
    return 0;}
```

9- TIC-TAC-TOE:-

Code:

```
#include<bits/stdc++.h>
using namespace std;
#define COMPUTER 1
#define HUMAN 2
#define SIDE 3
#define COMPUTERMOVE 'o'
#define HUMANMOVE 'x'
 struct Move
{
   int row, col;
};
char player = 'o', opponent = 'x';
string name;
void showBoard(char board[][SIDE])
{
   printf("\n\n");

   printf("\t\t\t  %c | %c  | %c  \n", board[0][0],
                     board[0][1], board[0][2]);
   printf("\t\t\t--------------\n");
   printf("\t\t\t  %c | %c  | %c  \n", board[1][0],
                     board[1][1], board[1][2]);
   printf("\t\t\t--------------\n");
   printf("\t\t\t  %c | %c  | %c  \n\n", board[2][0],
                     board[2][1], board[2][2]);

   return;
}
void showInstructions()
{
   printf("\t\t\t  Tic-Tac-Toe\n\n");
   printf("Choose a cell numbered from 0 to 8 as below"
         " and play\n\n");

   printf("\t\t\t  0 | 1  | 2  \n");
   printf("\t\t\t--------------\n");
   printf("\t\t\t  3 | 4  | 5  \n");
   printf("\t\t\t--------------\n");
   printf("\t\t\t  6 | 7  | 8  \n\n");

   printf("-\t-\t-\t-\t-\t-\t-\t-\t-\n\n");

   return;
}
void declareWinner(int whoseTurn)
{
   if (whoseTurn == COMPUTER)
      printf("COMPUTER has won\n");
   else
     cout<<name<<" has won\n";
   return;
}
bool rowCrossed(char board[][SIDE])
{
   for (int i=0; i<SIDE; i++)
   {
      if (board[i][0] == board[i][1] &&
         board[i][1] == board[i][2] &&
         board[i][0] != ' ')
         return (true);
   }
   return(false);
}
bool columnCrossed(char board[][SIDE])
```

```
{
   for (int i=0; i<SIDE; i++)
   {
      if (board[0][i] == board[1][i] &&
         board[1][i] == board[2][i] &&
         board[0][i] != ' ')
         return (true);
   }
   return(false);
}
bool diagonalCrossed(char board[][SIDE])
{
   if (board[0][0] == board[1][1] &&
      board[1][1] == board[2][2] &&
      board[0][0] != ' ')
      return(true);

   if (board[0][2] == board[1][1] &&
      board[1][1] == board[2][0] &&
       board[0][2] != ' ')
      return(true);

   return(false);
}
bool gameOver(char board[][SIDE])
{
   return(rowCrossed(board) || columnCrossed(board)
         || diagonalCrossed(board) );
}
bool isMovesLeft(char board[3][3])
{
   for (int i = 0; i<3; i++)
      for (int j = 0; j<3; j++)
         if (board[i][j]==' ')
            return true;
   return false;
}
int evaluate(char b[3][3])
{
   for (int row = 0; row<3; row++)
   {
      if (b[row][0]==b[row][1] &&
         b[row][1]==b[row][2])
      {
         if (b[row][0]==player)
            return +10;
         else if (b[row][0]==opponent)
            return -10;
      }
   }
   for (int col = 0; col<3; col++)
   {
      if (b[0][col]==b[1][col] &&
         b[1][col]==b[2][col])
      {
         if (b[0][col]==player)
            return +10;

         else if (b[0][col]==opponent)
            return -10;
      }
   }
   if (b[0][0]==b[1][1] && b[1][1]==b[2][2])
   {
      if (b[0][0]==player)
         return +10;
      else if (b[0][0]==opponent)
         return -10;
```

```
    }

    if (b[0][2]==b[1][1] && b[1][1]==b[2][0])
    {
        if (b[0][2]==player)
            return +10;
        else if (b[0][2]==opponent)
            return -10;
    }
    return 0;
}
int minimax(char board[3][3], int depth, bool isMax)
{
    int score = evaluate(board);
    if (score == 10)
        return score;
    if (score == -10)
        return score;
    if (isMovesLeft(board)==false)
        return 0;
    if (isMax)
    {
        int best = -1000;
        for (int i = 0; i<3; i++)
        {
            for (int j = 0; j<3; j++)
            {
                if (board[i][j]==' ')
                {
                    board[i][j] = player;
                    best = max( best,
                        minimax(board, depth+1, !isMax) );
                    board[i][j] = ' ';
                }
            }
        }
        return best;
    }
    else
    {
        int best = 1000;
        for (int i = 0; i<3; i++)
        {
            for (int j = 0; j<3; j++)
            {
                if (board[i][j]==' ')
                {
                    board[i][j] = opponent;
                    best = min(best,
                            minimax(board, depth+1, !isMax));
                    board[i][j] = ' ';
                }
            }
        }
        return best;
    }
}
Move findBestMove(char board[3][3])
{
    int bestVal = -1000;
    Move bestMove;
    bestMove.row = -1;
    bestMove.col = -1;
    for (int i = 0; i<3; i++)
    {
        for (int j = 0; j<3; j++)
        {
            if (board[i][j]==' ')
```

```
        {
          board[i][j] = player;
          if(gameOver(board)==true)
          {
            board[i][j]=' ';
            bestMove.row=i;
            bestMove.col=j;
            return bestMove;
          }
          int moveVal = minimax(board, 0, false);
          board[i][j] = ' ';
          if (moveVal > bestVal)
          {
            bestMove.row = i;
            bestMove.col = j;
            bestVal = moveVal;
          }
        }
      }
    }


    return bestMove;
}
void playTicTacToe(int whoseTurn)
{
    char board[SIDE][SIDE];
    for(int i=0;i<SIDE;i++)
      for(int j=0;j<SIDE;j++)
        board[i][j]=' ';
    showInstructions();
    int moveIndex = 0, x, y;
    while (gameOver(board) == false &&
          moveIndex != SIDE*SIDE)
    {
      if (whoseTurn == COMPUTER)
      {

        Move troy=findBestMove(board);
        x = troy.row;
        y = troy.col;
        board[x][y] = COMPUTERMOVE;
        printf("COMPUTER has put a %c in cell %d\n",
              COMPUTERMOVE, x*3+y);
        showBoard(board);
        moveIndex ++;
        whoseTurn = HUMAN;
      }

      else if (whoseTurn == HUMAN)
      {
        int bgy;
        label:cout<<"Enter the cell number where you want to play   ";

        cin>>bgy;
        cout<<endl;
        x=bgy/3;
        y=bgy%3;
        if(bgy>=9)
        {
          cout<<"The cell you selected doesnot exist.\n\n Please select a valid cell number....(0-8)\n\n";
          goto label;
        }
        else if(board[x][y]!=' ')
        {
          cout<<"The cell you selected is already occupied. Please try again...."<<endl;
          goto label;
        }
```

```
            board[x][y] = HUMANMOVE;
            cout<<name<<" has put a "<<HUMANMOVE<<" in cell "<<bgy<<endl;
            showBoard(board);
            moveIndex ++;
            whoseTurn = COMPUTER;
         }
      }
      if (gameOver(board) == false &&
            moveIndex == SIDE * SIDE)
         printf("It's a draw\n");
      else
      {
         if (whoseTurn == COMPUTER)
            whoseTurn = HUMAN;
         else if (whoseTurn == HUMAN)
            whoseTurn = COMPUTER;
         declareWinner(whoseTurn);
      }
      return;
}
int main()
{
      cout<<"WELCOME USER.....ENJOY PLAYING THE GAME----------\n\n\n_____ALL THE BEST_____\n\n\nCreator:- AYUSH SOMANI\n\n";
      cout<<"ENTER YOUR NAME  ";
      cin>>name;
      char WISH='Y';
      while(WISH=='Y'||WISH=='y')
      {
         cout<<"SELECT YOUR TURN.....   1 for first and 2 for second     ";
         int t;
         cin>>t;
         cout<<"\n\n\n\n";
         int whoseturn;
         if(t==1)
            whoseturn=HUMAN;
         else
            whoseturn=COMPUTER;
         playTicTacToe(whoseturn);
         cout<<"\n\n\n\nDO YOU WANT TO CONTINUE\n\n\n\nTYPE Y for yes or any other key to abort   ";
         cin>>WISH;
         cout<<"\n\n\n\n";
      }
      cout<<"THANK YOU FOR PLAYING THE GAME.....\n\nHOPE YOU ENJOYED THE GAME\n\n\n";
      return (0);
}
```

10- HIDATO PUZZLE:-

Code:-

```
#include<bits/stdc++.h>
using namespace std;
struct node{
   int val;
   unsigned char neighbors;
};
class hSolver{
public:
   hSolver()
   {
            dx[0] = -1; dx[1] = 0; dx[2] = 1; dx[3] = -1; dx[4] = 1; dx[5] = -1; dx[6] = 0; dx[7] = 1;
            dy[0] = -1; dy[1] = -1; dy[2] = -1; dy[3] = 0; dy[4] = 0; dy[5] = 1; dy[6] = 1; dy[7] = 1;
   }
   void solve( vector<string>& puzz, int max_wid )
   {
            if( puzz.size() < 1 ) return;
            wid = max_wid; hei = static_cast<int>( puzz.size() ) / wid;
            int len = wid * hei, c = 0; max = 0;
            arr = new node[len]; memset( arr, 0, len * sizeof( node ) );
            weHave = new bool[len + 1]; memset( weHave, 0, len + 1 );

            for( vector<string>::iterator i = puzz.begin(); i != puzz.end(); i++ )
            {
               if( ( *i ) == "*" ) {    arr[c++].val = -1; continue; }
               arr[c].val = atoi( ( *i ).c_str() );
               if( arr[c].val > 0 ) weHave[arr[c].val] = true;
               if( max < arr[c].val ) max = arr[c].val;
               c++;
            }

            solveIt(); c = 0;
            for( vector<string>::iterator i = puzz.begin(); i != puzz.end(); i++ )
            {
               if( ( *i ) == "." )
               {
                        ostringstream o; o << arr[c].val;
                        ( *i ) = o.str();
               }
               c++;
            }
            delete [] arr;
            delete [] weHave;
   }

private:
   bool search( int x, int y, int w )
   {
            if( w == max ) return true;

            node* n = &arr[x + y * wid];
            n->neighbors = getNeighbors( x, y );
            if( weHave[w] )
            {
               for( int d = 0; d < 8; d++ )
               {
                        if( n->neighbors & ( 1 << d ) )
                        {
                           int a = x + dx[d], b = y + dy[d];
                           if( arr[a + b * wid].val == w )
                           if( search( a, b, w + 1 ) ) return true;
                        }
               }
               return false;
            }
```

```
            for( int d = 0; d < 8; d++ )
            {
              if( n->neighbors & ( 1 << d ) )
              {
                        int a = x + dx[d], b = y + dy[d];
                        if( arr[a + b * wid].val == 0 )
                        {
                          arr[a + b * wid].val = w;
                          if( search( a, b, w + 1 ) ) return true;
                          arr[a + b * wid].val = 0;
                        }
              }
            }
            return false;
  }

  unsigned char getNeighbors( int x, int y )
  {
            unsigned char c = 0; int m = -1, a, b;
            for( int yy = -1; yy < 2; yy++ )
              for( int xx = -1; xx < 2; xx++ )
              {
                        if( !yy && !xx ) continue;
                        m++; a = x + xx, b = y + yy;
                        if( a < 0 || b < 0 || a >= wid || b >= hei ) continue;
                        if( arr[a + b * wid].val > -1 ) c |= ( 1 << m );
              }
            return c;
  }

  void solveIt()
  {
            int x, y; findStart( x, y );
            if( x < 0 ) { cout << "\nCan't find start point!\n"; return; }
            search( x, y, 2 );
  }
  void findStart( int& x, int& y ){
            for( int b = 0; b < hei; b++ )
              for( int a = 0; a < wid; a++ )
                        if( arr[a + wid * b].val == 1 ) { x = a; y = b; return; }
            x = y = -1;
  }
  int wid, hei, max, dx[8], dy[8];
  node* arr;
  bool* weHave;
};
int main( int argc, char* argv[] ){
  int wid;
  //string p = ". 33 35 . . * * * . . 24 22 . * * * . . . 21 . . * * . 26 . 13 40 11 * * 27 . . . 9 . 1 * * * . . 18 . . * * * * * . 7 . . * * * * * * 5 ."; wid = 8;
  //string p = "54 . 60 59 . 67 . 69 . . 55 . . 63 65 . 72 71 51 50 56 62 . * * * * . . . 14 * * 17 . * 48 10 11 * 15 . 18 . 22 . 46 . * 3 . 19 23 . . 44 . 5 . 1 33 32 . . 43 7 . 36 .
27 . 31 42 . . 38 . 35 28 . 30"; wid = 9;
  //string p = ". 58 . 60 . . 63 66 . 57 55 59 53 49 . 65 . 68 . 8 . . 50 . 46 45 . 10 6 . * * * . 43 70 . 11 12 * * * 72 71 . . 14 . * * * 30 39 . 15 3 17 . 28 29 . . 40 . . 19 22 .
. 37 36 . 1 20 . 24 . 26 . 34 33"; wid = 9;
  istringstream iss( p ); vector<string> puzz;
  copy( istream_iterator<string>( iss ), istream_iterator<string>(), back_inserter<vector<string> >( puzz ) );
  hSolver s; s.solve( puzz, wid );
  int c = 0;
  for( vector<string>::iterator i = puzz.begin(); i != puzz.end(); i++ ){
            if( ( *i ) != "*" && ( *i ) != "." ){
              if( atoi( ( *i ).c_str() ) < 10 ) cout << "0";
              cout << ( *i ) << " ";
            }
            else cout << "   ";
            if( ++c >= wid ) { cout << endl; c = 0; }
  }
  cout << endl << endl;
  return 0;
}
```

11- N-QUEENS:-

Code:-

```
#include<bits/stdc++.h>
using namespace std;
#define N 4
void printSolution(int board[N][N])
{
   static int k = 1;
   printf("%d-\n",k++);
   for (int i = 0; i < N; i++)
   {
      for (int j = 0; j < N; j++)
         printf(" %d ", board[i][j]);
      printf("\n");
   }
   printf("\n");
}
bool isSafe(int board[N][N], int row, int col)
{
   int i, j;
   for (i = 0; i < col; i++)
      if (board[row][i])
         return false;
   for (i=row, j=col; i>=0 && j>=0; i--, j--)
      if (board[i][j])
         return false;
   for (i=row, j=col; j>=0 && i<N; i++, j--)
      if (board[i][j])
         return false;

   return true;
}
bool solveNQUtil(int board[N][N], int col)
{
   if (col == N)
   {
      printSolution(board);
      return true;
   }
   bool res = false;
   for (int i = 0; i < N; i++)
   {
      if ( isSafe(board, i, col) )
      {
         board[i][col] = 1;
         res = solveNQUtil(board, col + 1) || res;
         board[i][col] = 0;
      }
   }
   return res;
}
void solveNQ()
{
   int board[N][N];
   memset(board, 0, sizeof(board));

   if (solveNQUtil(board, 0) == false)
   {
      printf("Solution does not exist");
      return ;
   }

   return ;
}
int main()
{
```

```
    solveNQ();
    return 0;
}
```

N-QUEENS (PROLOG)

Code:-

```
same1([],[]).
same1([Head|Tail],[Head1|Tail1]):-Head=Head1,same1(Tail,Tail1).
different(X,Y):-X>Y.
different(X,Y):-Y>X.
check(A,[],R,N).
check(A,[BH|BT],R,N):-BH>0,N1 is N+1,BH<N1,R1 is A+R,R3 is A-R,different(BH,R1),different(BH,R3),different(BH,A),R2 is R+1,check(A,BT,R2,N).
solution([],N).
solution([Head|Tail],N):-Head>0,N1 is N+1,Head<N1,check(Head,Tail,1,N),solution(Tail,N).
fill1(4,[4,3,2,1]).
fill1(N,[Head|Tail]):-Head=N,N1 is N-1,fill1(N1,Tail).
answer(R,Z,N):-permutation(R,Q),solution(Q,N),same1(Z,Q).
nqueen(0,[]).
nqueen(N,Z):-fill1(N,R),answer(R,Z,N).
```

Matrix Multiplication(PROLOG)

Code:-

```
length1([],0).
length1([X|T],N):-length1(T,N1),N is N1+1.
empty([]).
empty([Head|Tail]):-length1(Head,0),empty(Tail).
multrow(D1,D2,X3,R):-empty(D1),empty(D2),X3=R.
multrow([Head|Tail],[[F1|F2]|F3],X3,R):-R1 is Head*F1+R,multrow(Tail,F3,X3,R1).
same1([],[]).
same1([X1|T1],[X2|T2]):-X1=X2,same1(T1,T2).
remove([],[]).
remove([[F1|F2]|F3],[Head|Tail]):-same1(F2,Head),remove(F3,Tail).
mult(X,M,L):-empty(M),empty(L).
mult(X,[[F1|F2]|F3],[X3|Y3]):-multrow(X,[[F1|F2]|F3],X3,0),remove([[F1|F2]|F3],Z1),mult(X,Z1,Y3).
multiplymat([],S,[]).
multiplymat([X|Y],S,[X5|Y5]):-mult(X,S,X5),multiplymat(Y,S,Y5).
```
