Q9.
Implement the A* algorithm for finding the goal node for OR Graph.

DESCRIPTION:

A* is a cornerstone name of many AI systems and has been used since it was developed in 1968 by Peter Hart; Nils Nilsson and Bertram Raphael. It is the combination of Dijkstra's algorithm and Best first search. It can be used to solve many kinds of problems. A* search finds the shortest path through a search space to goal state using heuristic function. This technique finds minimal cost solutions and is directed to a goal state called A* search. In A*, the * is written for optimality purpose. The A* algorithm also finds the lowest cost path between the start and goal state, where changing from one state to another requires some cost. A* requires heuristic function to evaluate the cost of path that passes through the particular state. This algorithm is complete if the branching factor is finite and every action has fixed cost. A* requires heuristic function to evaluate the cost of path that passes through the particular state.The implementation of A* algorithm is 8-puzzle game

It can be defined by following formula.

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

Where

g(n): The actual cost path from the start state to the current state.h (n): The actual cost path from the current state to goal state. f (n): The actual cost path from the start state to the goal state.

For the implementation of A* algorithm we will use two arrays namely OPEN and CLOSE.

- **OPEN**: An array which contains the nodes that has been generated but has not been yet examined.
- **CLOSE:** An array which contains the nodes that have been examined.

ALGORITHM:

- Step 1: Place the starting node into OPEN and find its f (n) value.
- Step 2: Remove the node from OPEN, having smallest f (n) value. If it is a goal node then stop and return success.
- Step 3: Else remove the node from OPEN, find all its successors.
- Step 4: Find the f (n) value of all successors; place them into OPEN and place the removed node into CLOSE.

- Step 5: Go to Step-2.
- Step 6: Exit.

ADVANTAGES:

- It is complete and optimal.
- It is the best one from other techniques.
- It is used to solve very complex problems.
- It is optimally efficient, i.e. there is no other optimal algorithm guaranteed to expand fewer nodes than A*.

DISADVANTAGES:

- This algorithm is complete if the branching factor is finite and every action has fixed cost.
- The speed execution of A* search is highly dependant on the accuracy of the heuristic algorithm that is used to compute h (n). ● It has complexity problems.

IMPLEMENTATION:

CODE:

```
//9. Implement the A* algorithm for finding the goal node for OR Graph.

#include <list>
#include <algorithm>
#include <iostream>
#include <stdio.h>

class point
{ public: point( int a = 0, int b = 0 )
            { x = a; y = b;
            }
            bool operator ==( const point& o )
            {
                  return o.x == x && o.y == y;
            }
            point operator +( const point& o )
            { return point( o.x + x, o.y + y );
            } int x, y;
};

class map
{ public:
```

```
        map()
        { char t[8][8] = {
                        {0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 1, 1, 1, 0}, {0, 0, 1, 0, 0, 0, 1, 0},
                        {0, 0, 1, 0, 0, 0, 1, 0}, {0, 0, 1, 1, 1, 1, 1, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0}
                };
                w = h = 8; for( int r = 0;
                r < h; r++ )
                    for( int s = 0; s < w; s++ )
                        m[s][r] = t[r][s];
        }

        int operator() ( int x, int y )
        { return m[x][y];
        }
        char m[8][8];
        int w, h;
};

class node
{ public: bool operator == (const node& o )
        {
            return pos == o.pos;
        }
        bool operator == (const point& o )
        {
            return pos == o;
        }
        bool operator < (const node& o )
        {
            return dist + cost < o.dist + o.cost;
        }
        point pos, parent;
        int dist, cost;
};

class aStar
{ public: aStar()
        { neighbours[0] = point( -1, -1 );
                neighbours[1] = point( 1, -1 );
```

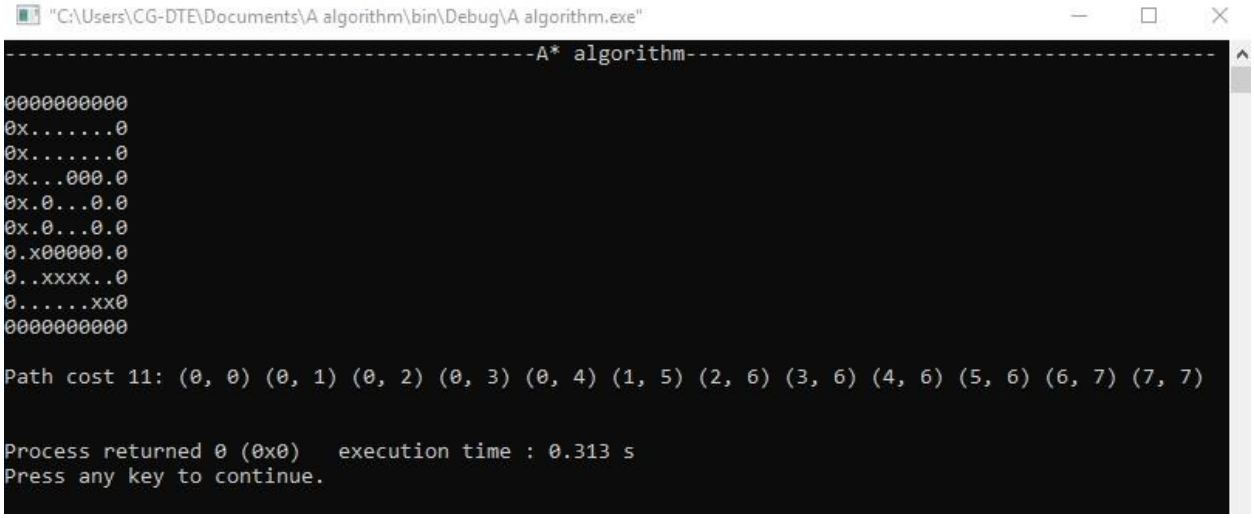
```
        neighbours[2] = point( -1, 1 );
        neighbours[3] = point( 1, 1 );
        neighbours[4] = point( 0, -1 );
        neighbours[5] = point( -1, 0 );
        neighbours[6] = point( 0, 1 );
        neighbours[7] = point( 1, 0 );
}

int calcDist( point& p )
{ int x = end.x - p.x, y = end.y - p.y; return(
        x * x + y * y );
}

bool isValid( point& p )
{
       return ( p.x >-1 && p.y > -1 && p.x < m.w && p.y < m.h );
}

bool existPoint( point& p, int cost )
{ std::list<node>::iterator i; i = std::find( closed.begin(),
        closed.end(), p ); if( i!= closed.end() )
        {
                if( ( *i ).cost + ( *i ).dist < cost )
                        return true;
                else
                { closed.erase( i ); return
                        false;
                }
        } i = std::find( open.begin(), open.end(), p
        ); if( i!= open.end() )
        {
                if( ( *i ).cost + ( *i ).dist < cost )
                   return true;
                else
                { open.erase( i ); return
                        false;
                }
        } return
        false;
}
```

```
bool fillOpen( node& n )
{ int stepCost, nc, dist; point
        neighbour; for( int x = 0;
        x < 8; x++ )
        { stepCost = x < 4 ? 1 : 1; neighbour = n.pos
                + neighbours[x]; if(neighbour ==
                end ) return true;
              if( isValid( neighbour ) && m( neighbour.x, neighbour.y ) != 1 )
                {
                        nc= stepCost + n.cost; dist =
                        calcDist( neighbour ); if( !existPoint(
                        neighbour, nc + dist ) )
                        {
                                node m;
                                m.cost = nc;
                                m.dist = dist;
                                m.pos = neighbour;
                                m.parent = n.pos;
                                open.push_back( m ); }
                }
        } return
        false;
}

bool search( point& s, point& e, map& mp )
{
        node n;
        end = e;
        start = s;
        m = mp;
        n.cost = 0;
        n.pos = s;
        n.parent = 0;
        n.dist = calcDist( s );
        open.push_back( n );
        while( !open.empty() )
        {
                //open.sort(); node n =
                open.front();
                open.pop_front();
                closed.push_back( n
```

```
                        ); if( fillOpen( n ) )
                        return true;
                } return
                false;
        }

        int path( std::list<point>& path )
        {
                path.push_front( end ); int cost = 1 + closed.back().cost;
                path.push_front( closed.back().pos ); point  parent =
                closed.back().parent; for( std::list<node>::reverse_iterator i
                = closed.rbegin(); i != closed.rend(); i++ )
                { if( ( *i ).pos == parent && !( ( *i ).pos == start ) )
                        { path.push_front( ( *i ).pos );
                                parent = ( *i ).parent;
                        }
                } path.push_front( start
                ); return cost;
        }
        map m;
        point end, start; point
        neighbours[8];
        std::list<node> open;
        std::list<node>
        closed;
};

int main( int argc, char* argv[] )
{
        map m;
        point s, e( 7, 7 );
        aStar as;
        std::cout<<"----------------------------------------A*algorithm----------------------
        ----------";
        std::cout<<"\n\n"; if(
        as.search( s, e, m ) )
        { std::list<point> path; int c =
                as.path( path ); for( int y
                = -1; y < 9; y++ )
                {
                        for( int x = -1; x < 9; x++ )
```

```
                    { if( x < 0 || y < 0 || x > 7 || y > 7 || m( x, y ) == 1 )
                            std::cout << '0';
                            else
                            {

                                    if( std::find( path.begin(), path.end(), point(
                                    x, y ) )!= path.end() ) std::cout << "x";
                                     else std::cout << ".";
                            }
                    } std::cout <<
                    "\n"; } std::cout
                    << "\nPath cost
                    " << c << ": ";
                    for(
                    std::list<point>::i
                    terator i =
                    path.begin(); i !=
                    path.end(); i++ )
            { std::cout<< "(" << ( *i ).x << ", " << ( *i ).y << ") ";
            }
    } std::cout <<
    "\n\n"; return 0;
}
```

OUTPUT:



Q10.
Solve the classical missionary & cannibals problem by any search algorithm.

DESCRIPTION:

Three missionaries and three cannibals find themselves on one side of a river.they have agreed that they would all like to get to the other side.But the missionaries are not sure what else the cannibals have agreed to.so the missionaries want to manage the trip across the river in such a way that the number of missionaries on either side of the river is never less than the number of cannibals who are on the same side.the only boat available holds only two people at a time.How can everyone get across the river without the missionaries risking being eaten?So for solving the problem and to find out the solution on different states is called the Missionaries and Carnival Problem.

PROCEDURE:

Let us take an example. Initially a boatman, Grass, Tiger and Goat is present at the left bank of the river and want to cross it. The only boat available is one capable of carrying 2 objects of portions at a time. The condition of safe crossing is that at no time the tiger present with goat, the goat present with the grass at the either side of the river. How they will cross the river?

The objective of the solution is to find the sequence of their transfer from one bank of the river to the other using the boat sailing through the river satisfying these constraints.

COMMENTS:

- This problem requires a lot of space for its state implementation.
- It takes a lot of time to search the goal node.
- The production rules at each level of state are very strict.

IMPLEMENTATION:

CODE:

```
//10. Solve the classical missionary &cannibals problem by any search algorithm.
#include<bits/stdc++.h>
using namespace std;

struct StateSTR
{
```

```
        int Mlhs; //nr missionaries on LHS    of river
        int Clhs; //nr cannibals    on LHS    of river int
        pos; //boat on LHS (0) or RHS(1) of river int
        Mrhs; //nr missionaries on RHS    of river int
        Crhs; //nr cannibals    on RHS    of river
        StateSTR * parent; //pointer to parent state int
        opUsed; bool operator == (const StateSTR &
        rhs) const
        { return ((Mlhs == rhs.Mlhs) && (Clhs == rhs.Clhs) &&
                (Mrhs == rhs.Mrhs) && (Crhs == rhs.Crhs) &&(pos == rhs.pos));
        }
};

ostream & operator << (ostream & out, const StateSTR & s)
{ out << "Mlhs:" << s.Mlhs << endl; out << "Clhs:"
        << s.Clhs << endl; out << "Boat:" <<
        s.pos << endl; out << "Mrhs:" << s.Mrhs
        << endl; out << "Crhs:" << s.Crhs << endl
        << endl; return out;
}

bool validState(StateSTR * S)
{ if ((( * S).Clhs < 0) || (( * S).Clhs > 3)) return false; if ((( *
        S).Crhs < 0) || (( * S).Crhs > 3)) return false; if ((( *
        S).Mlhs < 0) || (( * S).Mlhs > 3)) return false; if ((( *
        S).Mrhs < 0) || (( * S).Mrhs > 3)) return false; if ((( *
        S).pos != 0) && (( * S).pos != 1)) return false; if ((((
        * S).Clhs > ( * S).Mlhs) && (( * S).Mlhs > 0)) ||
              ((( * S).Crhs > ( * S).Mrhs) && (( * S).Mrhs > 0)))
           return false;
        return true;
}
StateSTR * nextState(StateSTR * Z,const int j)
{
        StateSTR * S = new StateSTR();
        ( * S) = ( * Z);
        ( * S).opUsed = j;
        switch (j)
        {
                case 0:
                {
```

```
                ( * S).pos -= 1;
                ( * S).Mlhs += 0;
                ( * S).Clhs += 1;
                ( * S).Mrhs -= 0;
                ( * S).Crhs -= 1;
        }
        break;
        case 1:
        {
                ( * S).pos -= 1;
                ( * S).Mlhs += 0;
                ( * S).Clhs += 2;
                ( * S).Mrhs -= 0;
                ( * S).Crhs -= 2;
        }
        break;
        case 2:
        {
                ( * S).pos -= 1;
                ( * S).Mlhs += 1;
                ( * S).Clhs += 0;
                ( * S).Mrhs -= 1;
                ( * S).Crhs -= 0;
        }
        break;
        case 3:
        {
                ( * S).pos -= 1;
                ( * S).Mlhs += 2;
                ( * S).Clhs += 0;
                ( * S).Mrhs -= 2;
                ( * S).Crhs -= 0;
        }
        break;
        case 4:
        {
                ( * S).pos -= 1;
                ( * S).Mlhs += 1;
                ( * S).Clhs += 1;
                ( * S).Mrhs -= 1;
                ( * S).Crhs -= 1;
```

```
}
break;
case 5:
{
        ( * S).pos += 1;
        ( * S).Mrhs += 0;
        ( * S).Crhs += 1;
        ( * S).Mlhs -= 0;
        ( * S).Clhs -= 1;
}
break;
case 6:
{
        ( * S).pos += 1;
        ( * S).Mrhs += 0;
        ( * S).Crhs += 2;
        ( * S).Mlhs -= 0;
        ( * S).Clhs -= 2;
}
break;
case 7:
{
        ( * S).pos += 1;
        ( * S).Mrhs += 1;
        ( * S).Crhs += 0;
        ( * S).Mlhs -= 1;
        ( * S).Clhs -= 0;
}
break;
case 8:
{
        ( * S).pos += 1;
        ( * S).Mrhs += 2;
        ( * S).Crhs += 0;
        ( * S).Mlhs -= 2;
        ( * S).Clhs -= 0;
}
break;
case 9:
{
        ( * S).pos += 1;
```

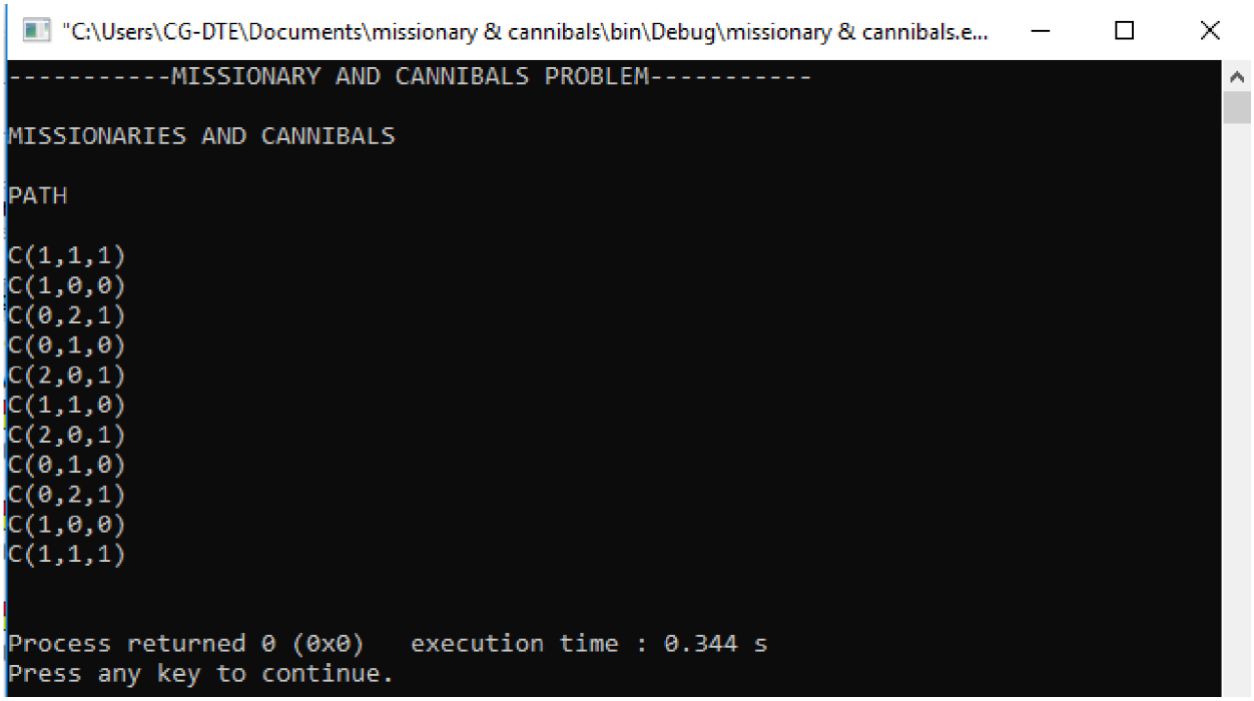
```
                    ( * S).Mrhs += 1;
                    ( * S).Crhs += 1;
                    ( * S).Mlhs -= 1;
                    ( * S).Clhs -= 1;
                }
                 break;
        }
        return S;
}

bool notFound(StateSTR * Y, list < StateSTR * > OPEN, list < StateSTR * >
CLOSED)
{ list < StateSTR * > ::iterator itr1 = OPEN.begin(); list <
        StateSTR * > ::iterator itr2 = CLOSED.begin(); for (;
        itr1 != OPEN.end(); itr1++) if (( * ( * itr1)) == ( * Y))
        break;
        for (; itr2 != CLOSED.end(); itr2++)
            if (( * ( * itr2)) == ( * Y)) break;
        if ((itr1 == OPEN.end()) && (itr2 == CLOSED.end()))
            return true;
        return false;
}

void addChildren(list < StateSTR * > & OPEN,list < StateSTR * > & CLOSED,
StateSTR * Y)
{
        StateSTR * tState;
        for (int i = 0; i < 10; i++)
        {
                tState = nextState(Y, i); if ((validState(tState))
                &&(notFound(tState, OPEN, CLOSED)))
                {
                        ( * tState).parent = Y;
                        OPEN.push_front(tState);
                }
                else delete
                    tState;
        }
        return;
}
```

```
void printOP(int n)
{
        switch (n)
        {
            case 0: cout << "C(0,1,0)" << endl;
                    break;
            case 1: cout << "C(0,2,0)" << endl;
                    Break; case 2: cout <<
            "C(1,0,0)" << endl; break;
            case 3: cout << "C(2,0,0)" << endl;
                    break;
            case 4: cout << "C(1,1,0)" << endl;
                    break;
            case 5: cout << "C(0,1,1)" << endl;
                    break;
            case 6: cout << "C(0,2,1)" << endl;
                    break;
            case 7: cout << "C(1,0,1)" << endl;
                    break;
            case 8: cout << "C(2,0,1)" << endl;
                    break;
            case 9: cout << "C(1,1,1)" << endl;
                    break;
        }
} int
main() {
        cout<<"-----------MISSIONARY AND CANNIBALS PROBLEM-----------
        \n"; cout << "\nMISSIONARIES AND CANNIBALS"; bool searchResult
        = false; stack < int > opsUsed;
        StateSTR START =
        {
           3,
           3,
           0,
           0,
           0,
           NULL,
           -1
        };

        StateSTR GOAL =
```

```
{
   0,
   0,
   1,
   3,
   3,
   NULL
};

StateSTR * X; StateSTR *
tempState; list < StateSTR *
> OPEN; list < StateSTR * >
CLOSED; OPEN.push_front(
& START);

while (!OPEN.empty())
{
            X = OPEN.front(); //stack-like operation
            OPEN.pop_front(); if (( * X) == GOAL)
            { searchResult = true; break;
            }
            else
            {
                        addChildren(OPEN, CLOSED, X);
                        CLOSED.push_back(X);
            }
}

//Display results if
(searchResult == true)
{
            cout << endl<<endl << "PATH" <<endl<< endl; for
            (StateSTR * p = X; p != NULL; p = ( * p).parent)
               opsUsed.push(( * p).opUsed);
}
while (!opsUsed.empty())
{
            printOP(opsUsed.top());
            opsUsed.pop();
}
```

```
        cout << endl;
        return 0;
}
```

OUTPUT:



Q11.

Solve the classical Travelling Salesman Problem of AI by heuristic approach.

DESCRIPTION:

The traveling salesman problem is a classic problem in combinatorial optimization. This problem is to find the shortest path that a salesman should take to traverse through a list of cities and return to the origin city. The list of cities and the distance between each pair are provided.

TSP is useful in various applications in real life such as planning or logistics. For example, a concert tour manager who wants to schedule a series of performances for the band must determine the shortest path for the tour to ensure reducing traveling costs and not making the band unnecessarily exhausted.

This is an NP-hard problem. In simple words, it means you can not guarantee to find the shortest path within a reasonable time limit. This is not unique to TSP though. In real-world optimization problems, you frequently encounter problems for which you must find sub-optimal solutions instead of optimal ones.

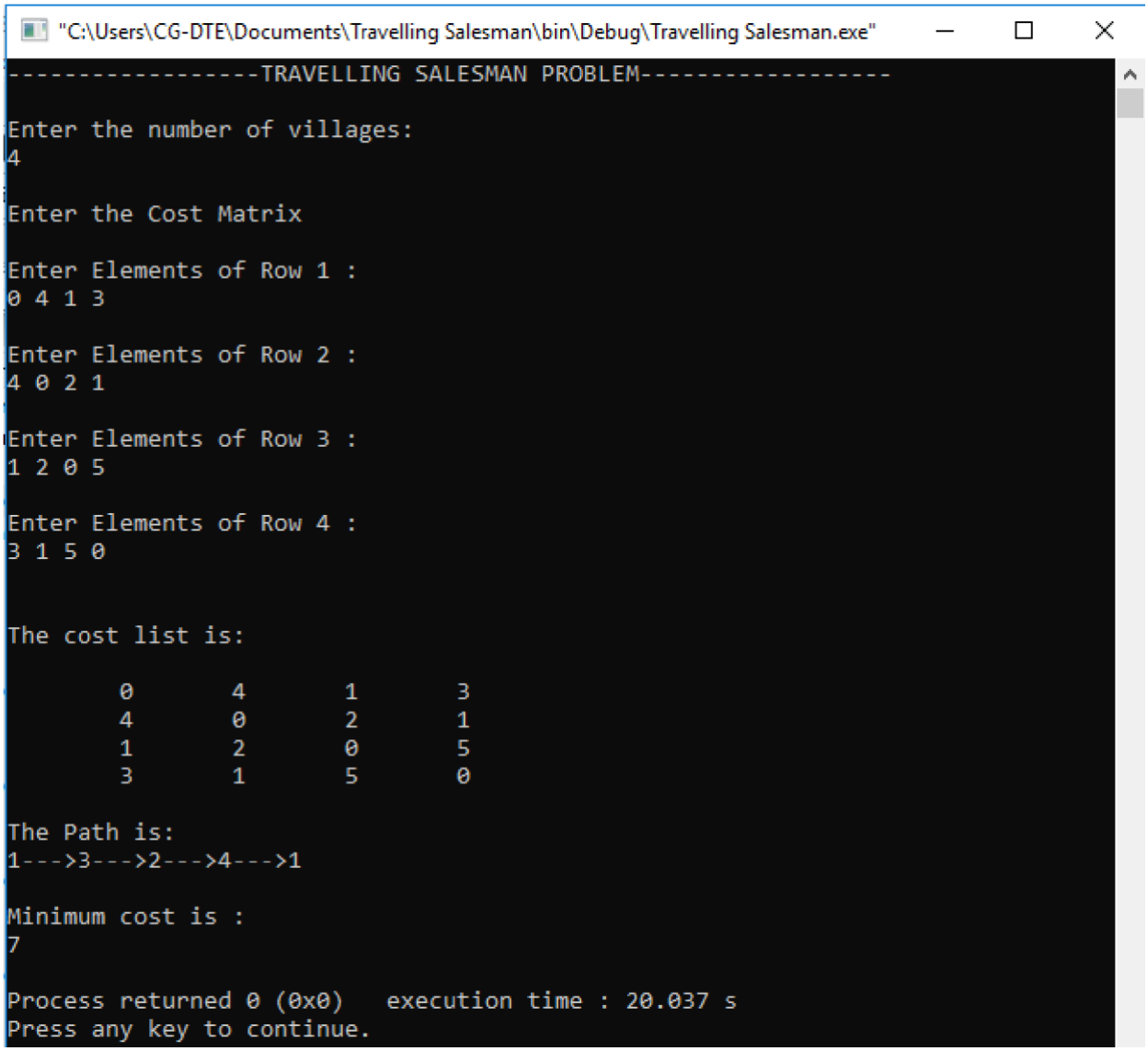
IMPLEMENTATION:

CODE:

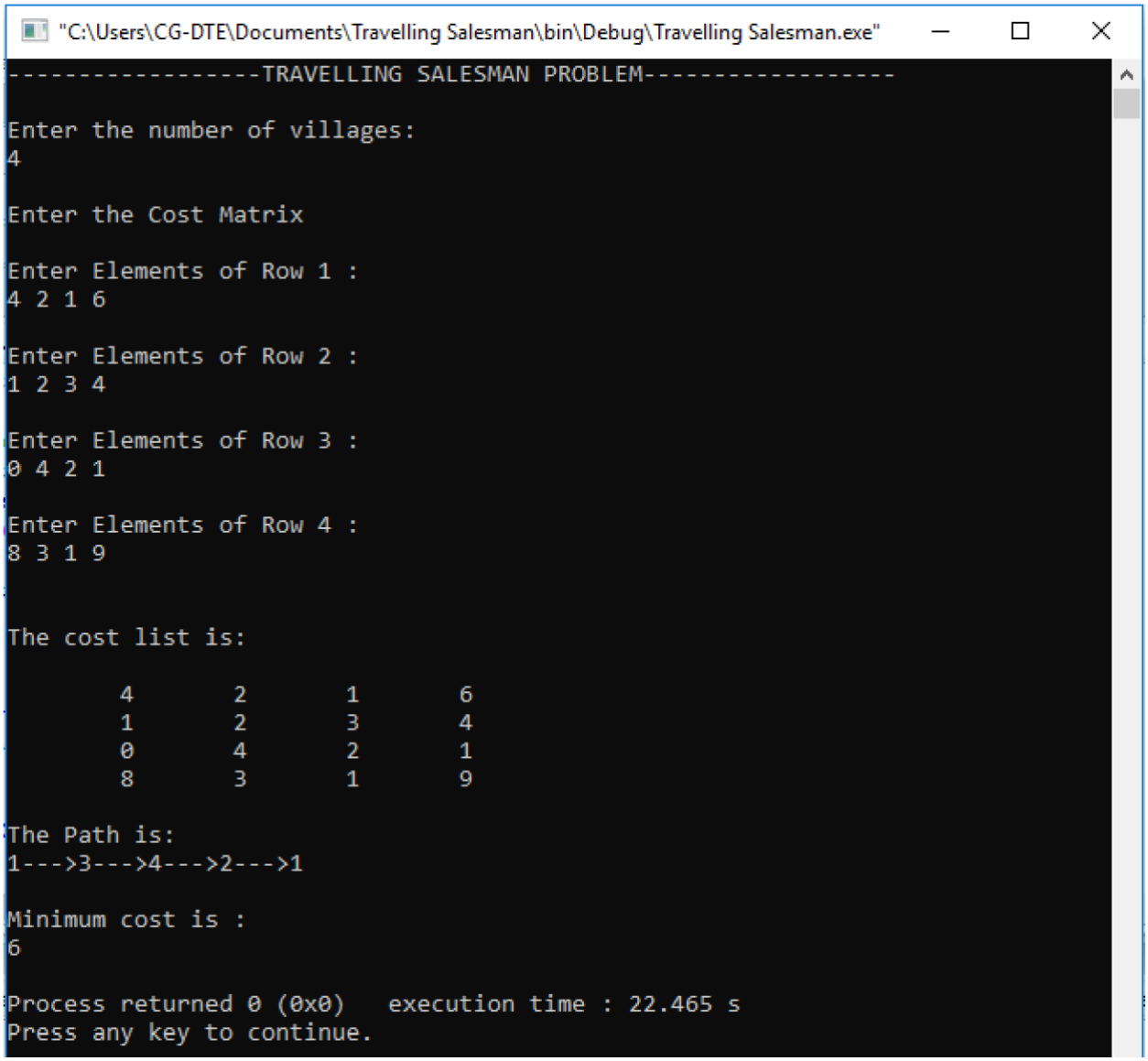
```
// Solve the classical Travelling Salesman Problem of AI by heuristic approach.
#include<stdio.h> #include
<iostream> using namespace std; int
ary[10][10],completed[10],n,cost=0;

void takeInput()
{ int i,j;
                cout<<"------------------TRAVELLING SALESMAN PROBLEM
                ------------------";
                cout<<"\n\nEnter the number of villages:\n";
                cin>>n; cout<<"\nEnter the Cost Matrix\n" ;
                for(i=0; i < n; i++)
                {
                        cout<<"\nEnter Elements of Row "<<i+1<<" :\n";
                        for( j=0; j < n; j++) cin>>ary[i][j];
                        completed[i]=0;
                } cout<<"\n\nThe cost list
                is:\n"; for( i=0; i < n; i++)
                {
                        cout<<"\n";
                        for(j=0; j < n; j++)
                            cout<<"\t"<<ary[i][j
                                    ];
                } }
    int least(int c)
    { int i,nc=999; int
                min=999,kmin;
                for(i=0; i < n; i++)
                { if((ary[c][i]!=0)&&(completed[i]==0))
                            if(ary[c][i]+ary[i][c] < min)
                            { min=ary[i][0]+ary[c][i];
                                    kmin=ary[c][i]; nc=i;
                            }
                }
                if(min!=999)
                    cost+=kmin;
                return nc;
    } void mincost(int
    city)
    { int i,ncity;
```

```
            completed[city]=1;
            cout<<city+1<<"--->";
            ncity=least(city);
            if(ncity==999)
            {
                        ncity=0;
                        cout<<ncity+1;
                        cost+=ary[city][ncity];
                        return;
            }
            mincost(ncity);
}

int main()
{
            takeInput(); cout<<"\n\nThe Path is:\n";
            mincost(0); //passing 0 because starting
            vertex cout<<"\n\nMinimum cost is :\n"<<cost;
            cout<<"\n"; return 0;
}
```

OUTPUT 1:


"C:\Users\CG-DTE\Documents\Travelling Salesman\bin\Debug\Travelling Salesman.exe"
------------------TRAVELLING SALESMAN PROBLEM------------------
Enter the number of villages:
4
Enter the Cost Matrix
Enter Elements of Row 1 :
0 4 1 3
Enter Elements of Row 2 :
4 0 2 1
Enter Elements of Row 3 :
1 2 0 5
Enter Elements of Row 4 :
3 1 5 0
The cost list is:
0 4 1 3
4 0 2 1
1 2 0 5
3 1 5 0
The Path is:
1--->3--->2--->4--->1
Minimum cost is :
7
Process returned 0 (0x0) execution time : 20.037 s
Press any key to continue.

OUTPUT 2:



Q 12.

Write a program to search any goal given an input graph using AO* algorithm in AO graph.

DESCRIPTION:

(And-Or) Graph The Depth first search and Breadth first search given earlier for OR trees or graphs can be easily adopted by AND-OR graph. The main difference lies in the way termination conditions are determined, since all goals following an AND nodes must be realized; where as a single goal node following an OR node will do. So for this purpose we are using AO* algorithm.

Like A* algorithm here we will use two arrays and one heuristic function.

OPEN: It contains the nodes that has been traversed but yet not been marked solvable or unsolvable.

CLOSE: It contains the nodes that have already been processed.
h(n): The distance from current node to goal node.

ALGORITHM:

- Step 1: Place the starting node into OPEN.
- Step 2: Compute the most promising solution tree say T0.
- Step 3: Select a node n that is both on OPEN and a member of T0. Remove it from OPEN and place it in CLOSE
- Step 4: If n is the terminal goal node then leveled n as solved and leveled all the ancestors of n as solved. If the starting node is marked as solved then success and exit.
- Step 5: If n is not a solvable node, then mark n as unsolvable. If starting node is marked as unsolvable, then return failure and exit.
- Step 6: Expand n. Find all its successors and find their h (n) value, push them into OPEN.
- Step 7: Return to Step 2. ● Step 8: Exit.

IMPLEMENTATION:

CODE:

```
#include<bits/stdc++.h>
using namespace std;
struct node
{
        int data; vector<
        vector<node* >* >v; bool
        mark; bool solved;
};
int edge_cost=0;

void insert(node* root)
{ cout<<"\nEnter data of node : "; cin>>root->data; cout<<"\nEnter number of
        OR nodes for value "<<root->data<<" : "; int or_no; cin>>or_no;
        for(int i=0; i<or_no; i++)
        {
                vector<node*>* ans=new vector<node*>; cout<<"\nEnter
                number of AND nodes for "<<i+1<<" or node for value "<<root-
                >data<<" : "; int and_no; cin>>and_no;
                for(int j=0; j<and_no; j++)
                {
```

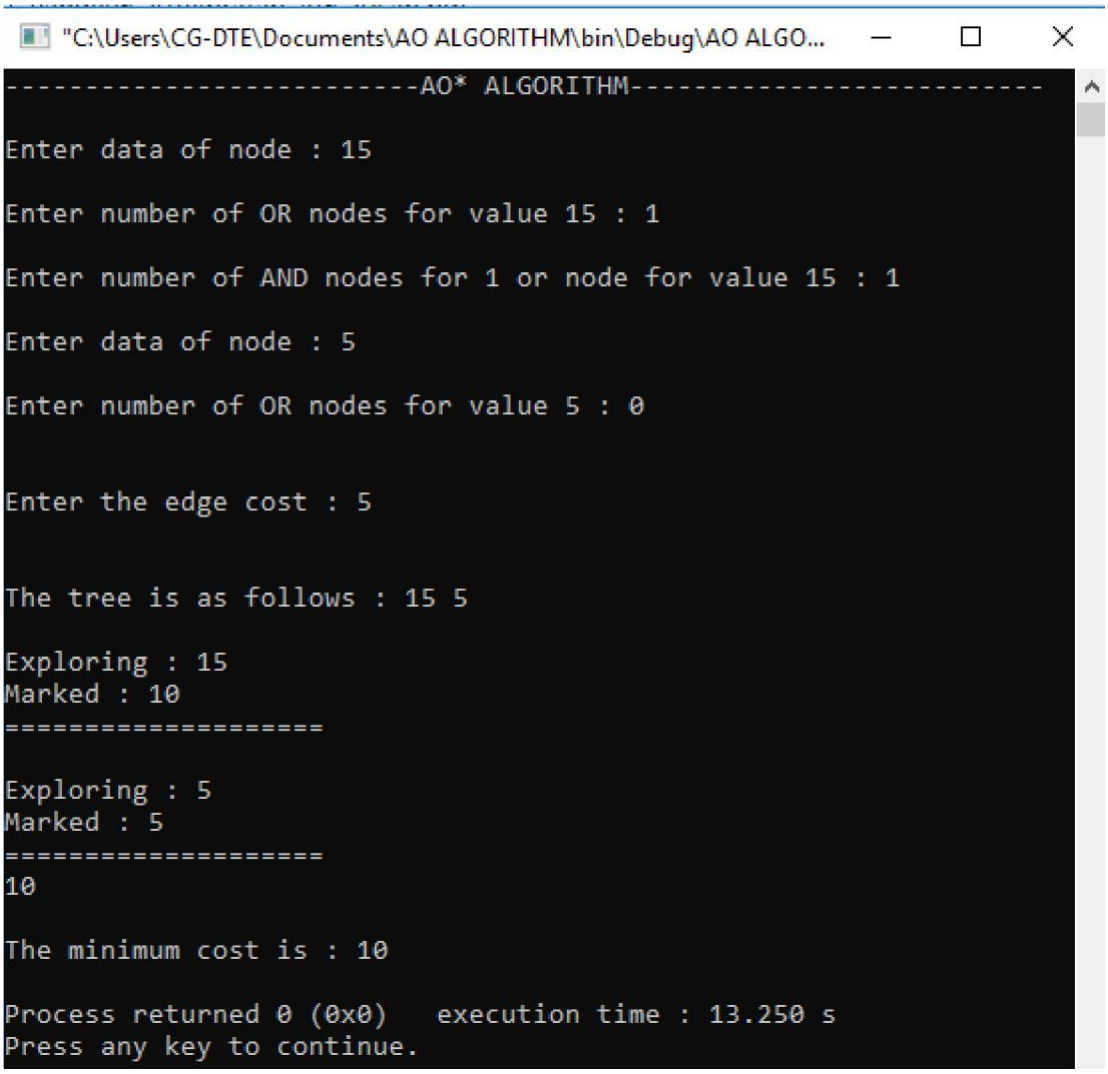
```
                    node* n=new node; n-
                    >solved=false; n-
                    >mark=false; insert(n);
                    (*ans).push_back(n);
                }
                root->v.push_back(ans);
        }
}
void aostar(node* root)
{
        vector<node*>* min_ans=new vector<node*>;
        (*min_ans).push_back(root); while(!root-
        >solved)
        {
                node* next_node=root;
                stack<node*>st; while(next_node &&
                next_node->mark)
                {
                        if((next_node->v).size()==0)
                        {
                                root->solved=true;
                                return;
                        }
                        int cost=INT_MAX; st.push(next_node);
                        for(unsigned int i=0; i<next_node->v.size();
                        i++)
                        {
                                vector<node*>*ans=(next_node->v)[i];
                                vector<node*> ans_v=*ans; int
                                temp_cost=0; for(unsigned int j=0;
                                j<(ans_v.size()); j++)
                                {
                                        node* n=ans_v[j]; temp_cost+=n-
                                        >data;
                                }
                                if(temp_cost<cost)
                                {
                                        min_ans=ans;
                                        cost=temp_cost;
                                }
                        }
```

```
                vector<node*> min_ans_v=*min_ans;
                next_node=NULL;
                for(unsigned int j=0; j<min_ans_v.size(); j++)
                {
                        if(min_ans_v[j]->mark)
                        {
                                next_node=min_ans_v[j];
                                break;
                        }
                }
        }
        vector<node*> min_ans_v=*min_ans;
        for(unsigned int j=0; j<min_ans_v.size(); j++)
        {
                node* n=min_ans_v[j];
                cout<<"Exploring : "<<n->data<<endl;
                int final_cost=INT_MAX; if(n-
                >v.size()==0)
                {
                        n->mark=true;
                }
                else
                { for(unsigned int i=0; i<n->v.size(); i++)
                        {
                                vector<node*>*ans=(n->v)[i];
                                vector<node*> ans_v=*ans;
                                int temp_cost=0;
                            for(unsigned int j=0; j<(ans_v.size()); j++)
                                {
                                        node* n=ans_v[j];
                                        temp_cost+=n->data;
                                        temp_cost+=edge_cost;
                                }
                                if(temp_cost<final_cost)
                                {
                                        final_cost=temp_cost;
                                }
                        }
                        n->data=final_cost;
                         n->mark=true;
                }
```

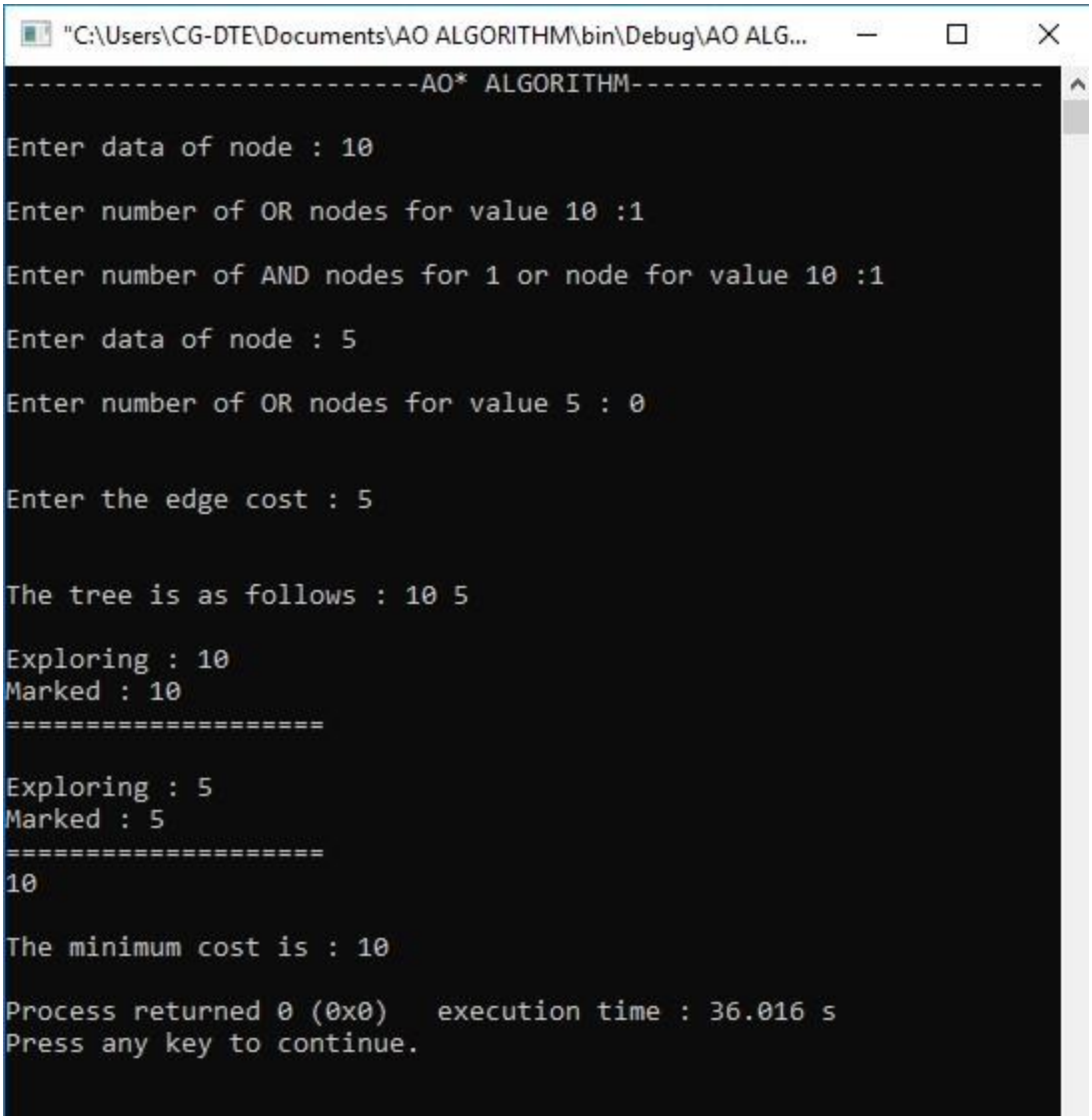
```
                cout<<"Marked : "<<n->data<<endl;
            } for(int i=0; i<20;
            i++) cout<<"=";
            cout<<endl;
            while(!st.empty())
            {
                node* n=st.top(); cout<<n->data<<" ";
                st.pop(); int final_cost=INT_MAX;
                for(unsigned int i=0; i<n->v.size();
                i++)
                {
                    vector<node*>*ans=(n->v)[i];
                    vector<node*> ans_v=*ans; int
                    temp_cost=0; for(unsigned int j=0;
                    j<(ans_v.size()); j++)
                    {
                        node* n=ans_v[j]; temp_cost+=n-
                        >data; temp_cost+=edge_cost;
                    }
                    if(temp_cost<final_cost)
                    {
                        min_ans=ans;
                        final_cost=temp_cost;
                    }
                }
                n->data=final_cost;
            }
            cout<<endl;
            next_node=root;
        }
}
void print(node* root)
{ if(root)
        {
            cout<<root->data<<" ";
            vector<vector<node*>* >vec=root->v;
            for(unsigned int i=0; i<(root->v).size(); i++)
            {
                vector<node*>* ans=(root->v)[i];
                vector<node*> ans_v=*ans;
                for(unsigned int j=0; j<ans_v.size(); j++)
                {
```

```
                                        node* n=ans_v[j];
                                        print(n);
                                    }
                                }
                            }
                            return;
}

int main()
{  cout<<"-------------------------AO*  ALGORITHM-------------------------\n";  node*
                root=new node; root->solved=false;

                root->mark=false; insert(root);
                cout<<endl; cout<<"\nEnter the
                edge cost : "; cin>>edge_cost;
                cout<<endl; cout<<"\nThe tree is
                as follows : "; print(root);
                cout<<endl<<endl; aostar(root); cout<<"\nThe
                minimum cost is : "<<root->data<<endl; return 0;
}
```

OUTPUT 1:


"C:\Users\CG-DTE\Documents\AO ALGORITHM\bin\Debug\AO ALGO...
----------AO* ALGORITHM----------
Enter data of node : 15
Enter number of OR nodes for value 15 : 1
Enter number of AND nodes for 1 or node for value 15 : 1
Enter data of node : 5
Enter number of OR nodes for value 5 : 0
Enter the edge cost : 5
The tree is as follows : 15 5
Exploring : 15
Marked : 10
==========
Exploring : 5
Marked : 5
==========
10
The minimum cost is : 10
Process returned 0 (0x0) execution time : 13.250 s
Press any key to continue.

OUTPUT 2:



Q13.

Implement the MINIMAX algorithm for any game playing.

DESCRIPTION:

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.

Let us combine minimax and evaluation function to write a proper Tic-Tac-Toe **AI** (**A**rtificial **I**ntelligence) that plays a perfect game.This AI will consider all possible scenarios and makes the most optimal move.

Finding the Best Move :

We shall be introducing a new function called findBestMove(). This function evaluates all the available moves using minimax() and then returns the best move the maximizer can make. The pseudocode is as follows :

```
function findBestMove(board):
    bestMove = NULL for each
    move in board :
        if current move is better than bestMove
            bestMove = current move
    return bestMove
```

Minimax :

To check whether or not the current move is better than the best move we take the help of minimax() function which will consider all the possible ways the game can go and returns the best value for that move, assuming the opponent also plays optimally

The code for the maximizer and minimizer in the minimax() function is similar to findBestMove(), the only difference is, instead of returning a move, it will return a value. Here is the pseudocode :

```
function minimax(board, depth, isMaximizingPlayer):
    if current board state is a terminal state :
        return value of the board
    if isMaximizingPlayer :
        bestVal = -INFINITY for
        each move in board :
            value = minimax(board, depth+1, false)
            bestVal = max( bestVal, value)
        return bestVal


    else :
        bestVal = +INFINITY for
        each move in board :
            value = minimax(board, depth+1, true)
            bestVal = min( bestVal, value)
        return bestVal
```

Checking for GameOver state :

To check whether the game is over and to make sure there are no moves left we use isMovesLeft() function. It is a simple straightforward function which checks whether a move is available or not and returns true or false respectively.Pseudocode is as follows:

```
function isMovesLeft(board):
    for each cell in board:
        if current cell is empty:
            return true
    return false
```

Making our AI smarter :

One final step is to make our AI a little bit smarter. Even though the following AI plays perfectly, it might choose to make a move which will result in a slower victory or a faster loss. Lets take an example and explain it.

Assume that there are 2 possible ways for X to win the game from a give board state.

Move A : X can win in 2 move

Move B : X can win in 4 moves

Our evaluation function will return a value of +10 for both moves A and B. Even though the move A is better because it ensures a faster victory, our AI may choose B sometimes. To overcome this problem we subtract the depth value from the evaluated score. This means that in case of a victory it will choose a the victory which takes least number of moves and in case of a loss it will try to prolong the game and play as many moves as possible. So the new evaluated value will be

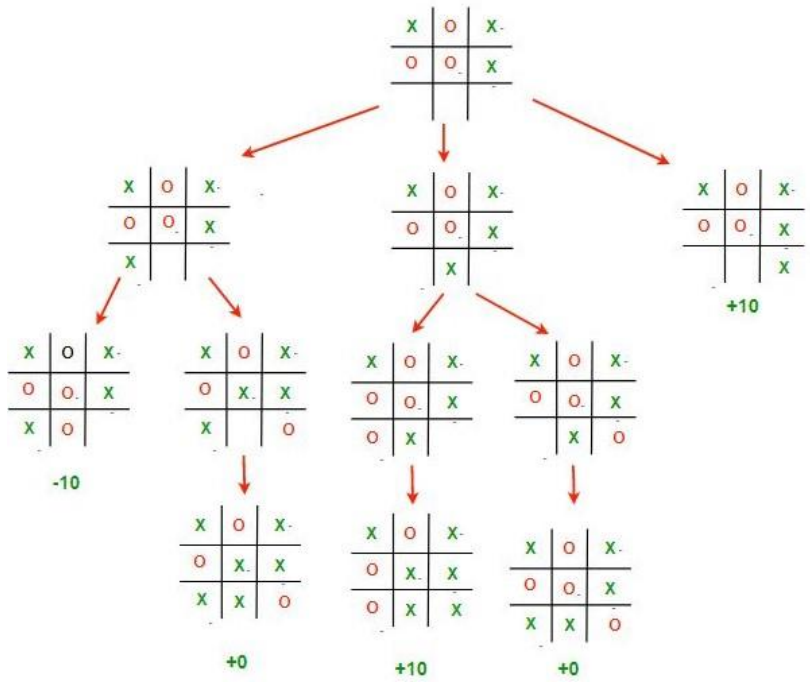
Move A will have a value of +10 – 2 = 8

Move B will have a value of +10 – 4 = 6

Now since move A has a higher score compared to move B our AI will choose move A over move B. The same thing must be applied to the minimizer. Instead of subtracting the depth we add the depth value as the minimizer always tries to get, as negative a value as possible. We can subtract the depth either inside the evaluation function or outside it. Anywhere is fine. I have chosen to do it outside the function. Pseudocode implementation is as follows.

```
if maximizer has won: return
    WIN_SCORE – depth

else if minimizer has won: return
    LOOSE_SCORE + depth
```

EXPLANATION: GAME TREE



IMPLEMENTATION:

CODE:

```
// C++ program to find the next optimal move for a player
#include<bits/stdc++.h>
using namespace std;

struct Move
{
        int row, col;
};

char player = 'x', opponent = 'o';

// This function returns true if there are moves remaining on the board.
// It returns false if there are no moves left to play.
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;
}
```

```
// This is the evaluation function
int evaluate(char b[3][3])
{
        // Checking for Rows for X or O victory.
        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;
                }
        }

        // Checking for Columns for X or O victory.
        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;
                }
        }
            // Checking for Diagonals for X or O victory.
            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;
            }

            // Else if none of them have won then return 0
            return 0;
```

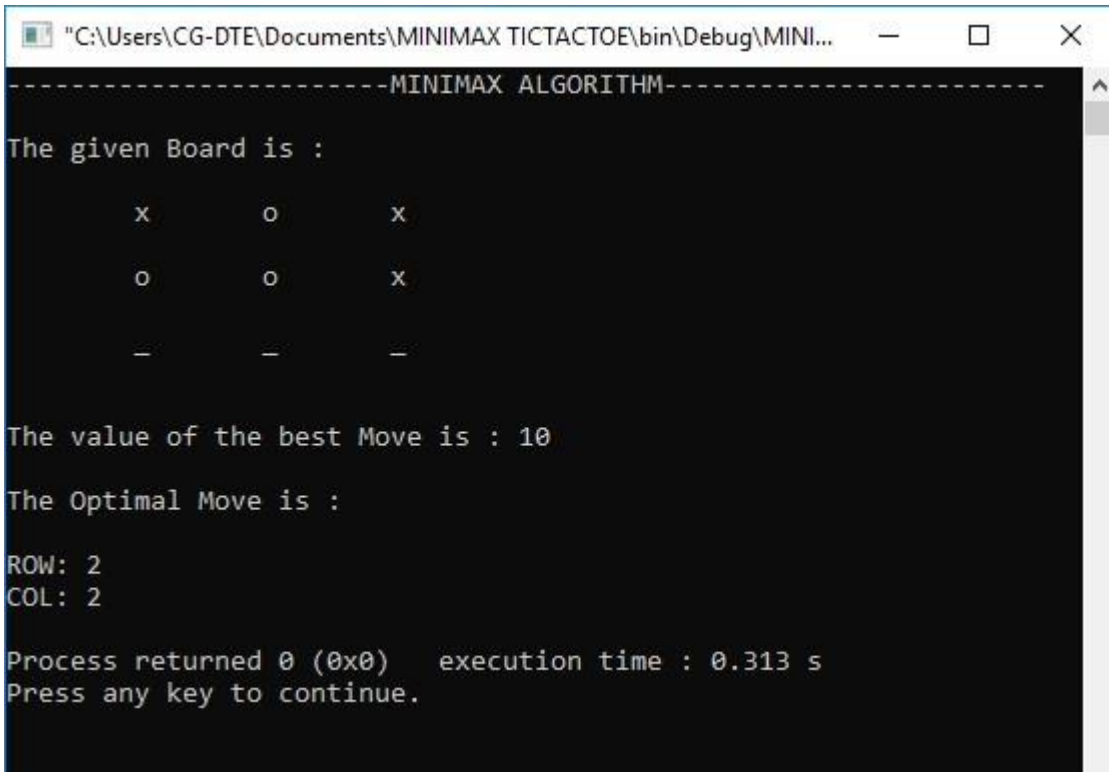
```
}

// This is the minimax function. It considers all the possible ways the game can go and returns the value of the board int minimax(char board[3][3], int depth, bool isMax)
{
        int score = evaluate(board);

        // If Maximizer has won the game return his/her evaluated score if (score == 10) return score;

        // If Minimizer has won the game return his/her evaluated score if (score == -10) return score;

        // If there are no more moves and no winner then it is a tie
        if (isMovesLeft(board)==false) return 0;

        // If this maximizer's move
        if (isMax)
        { int best = -1000;

                // Traverse all cells
                for (int i = 0; i<3; i++)
                { for (int j = 0; j<3; j++) {
                        // Check if cell is empty
                        if (board[i][j]=='_') {
                                // Make the move
                                board[i][j] = player;

                                // Call minimax recursively and
                                //choose the maximum value
                                best = max( best, minimax(board,
                                depth+1, !isMax) );

                                // Undo the move
                                board[i][j] = '_';
                        }
                }
                }
                return best;
        }
```

```
                // If this minimizer's move
                else
                {
                        int best = 1000;

                        // Traverse all cells
                        for (int i = 0; i<3; i++)
                        { for (int j = 0; j<3; j++) {
                                        // Check if cell is empty
                                        if (board[i][j]=='_')
                                        {
                                                // Make the move
                                                board[i][j] = opponent;

                                                // Call minimax recursively and
                                                // choose the minimum value
                                                best = min(best, minimax(board,
                                                depth+1, !isMax));

                                                // Undo the move
                                                board[i][j] = '_';
                                        }
                                }
                        }
                        return best;
                }
}

// This will return the best possible move for the player
Move findBestMove(char board[3][3])
{
                int bestVal = -1000;
                Move bestMove;
                bestMove.row = -1;
                bestMove.col = -1;

                // Traverse all cells, evaluate minimax function for all empty cells.
                // And return the cell with optimal value.
                for (int i = 0; i<3; i++)
                { for (int j = 0; j<3; j++) {
```

```
                                        // Check if cell is
                                        empty if
                                        (board[i][j]=='_') {
                                                // Make the move
                                                board[i][j] = player;

                                               //compute evaluation function for this move
                                                int moveVal = minimax(board, 0, false);

                                                // Undo the move
                                                board[i][j] = '_';

                                                // If the value of the current move is
                                                more than the best value, then update
                                                best if (moveVal > bestVal)
                                                {
                                                        bestMove.row = i;
                                                        bestMove.col = j;
                                                        bestVal = moveVal;
                                                }
                                        }
                                }
                        }

                        printf("\nThe value of the best Move is : %d\n\n", bestVal);
                        return bestMove;
}

// Driver code
int main()
{
                char board[3][3] =
                {
                        { 'x', 'o', 'x' },
                        { 'o', 'o', 'x' },
                        { '_', '_', '_' }
                };

                cout<<"-----------------------MINIMAX ALGORITHM-----------------------";
                cout<<"\n\n"; cout << "The given Board is :\n\n"; for (int i = 0; i < 3;
                ++i)
```

```
        { for (int j = 0; j < 3; ++j)
                { cout << '\t' << board[i][j] << ' ';
                }
                cout << endl<<endl;
         }

         Move bestMove = findBestMove(board);
         printf("The Optimal Move is :\n\n");
         printf("ROW: %d\n", bestMove.row);
         printf("COL: %d\n", bestMove.col );
         return 0;
}
```

OUTPUT:



END