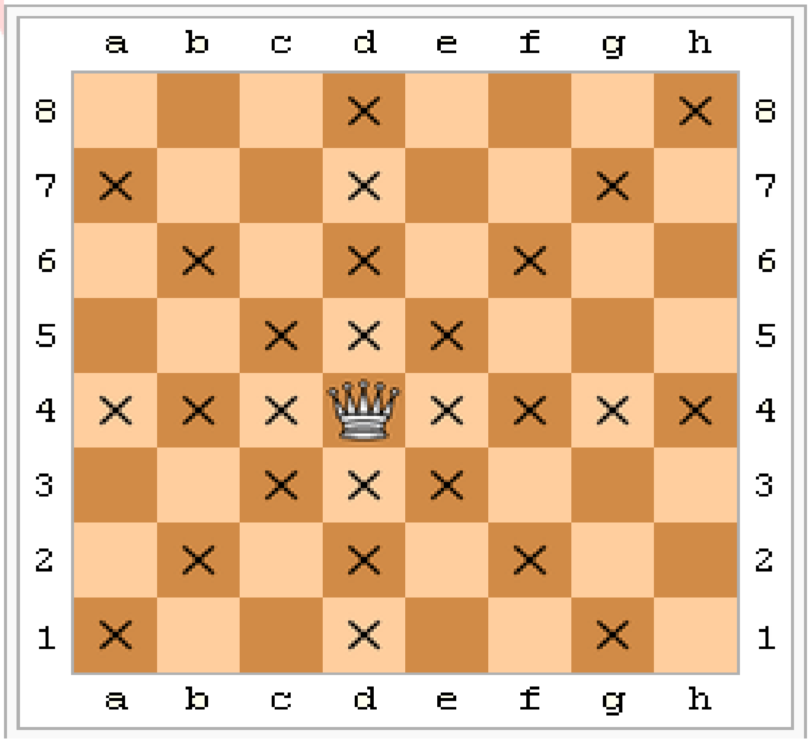
EXPERIMENT 1

AIM OF THE EXPERIMENT: To find the number of possible solutions of N-queens problem for a given N x N board.

THEORY:

In chess, a queen is considered to be the most powerful piece on the boards, because she can move as far as she pleases, horizontally, vertically, or diagonally.



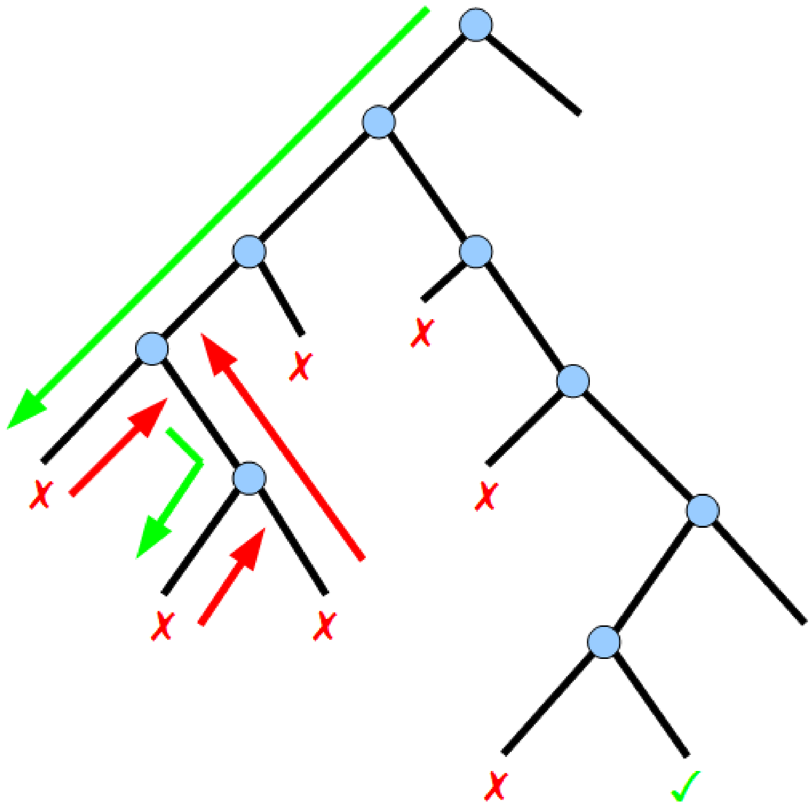
A standard chess board has 8 rows and 8 columns. The standard 8 by 8 Queen's problem asks how to place 8 queens on an ordinary chess board so that none of them can hit any other in one move.

A more general variation of the problem is the N by N queens problem which asks how to place N queens on a board with N rows and N columns.

In this experiment we are going to find the number of all possible solutions to a given N by N board.

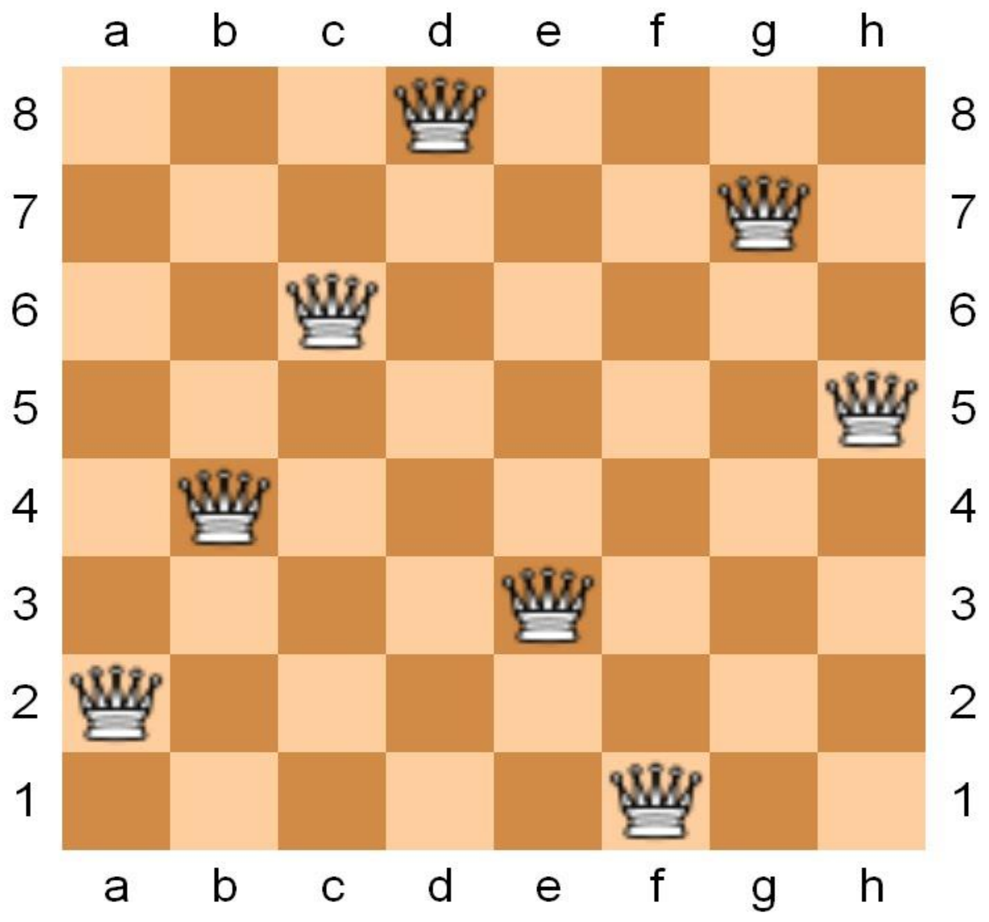
There can be many approaches to solve the N-queens problem, (one of them is brute force technique which has exponential time complexity) but the best approach to solve the given problem is through backtracking.

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

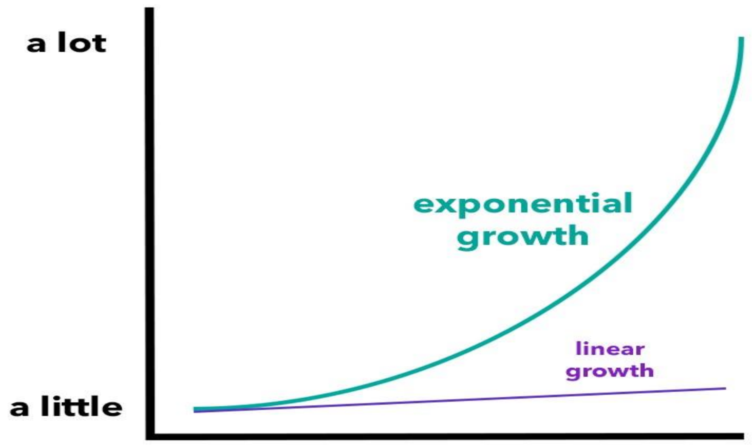


Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution. It is useless, for example, for locating a given value in an unordered table.
When it is applicable, however, backtracking is often much faster than brute force enumeration of all complete candidates, since it can eliminate many candidates with a single test.

The N-queens problem is a classic textbook example of the use of backtracking. In the common backtracking approach, the partial candidates are arrangements of k queens in the first k rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned.



The number of possible solutions for N-queens problem grows exponentially as the number of N grows, this is known as combinatorial explosion.



In mathematics, a combinatorial explosion is the rapid growth of the complexity of a problem due to how the combinatorics of the problem is affected by input, constraints, and bounds of the problem. Combinatorial explosion is sometimes used to justify the intractability of certain problems.

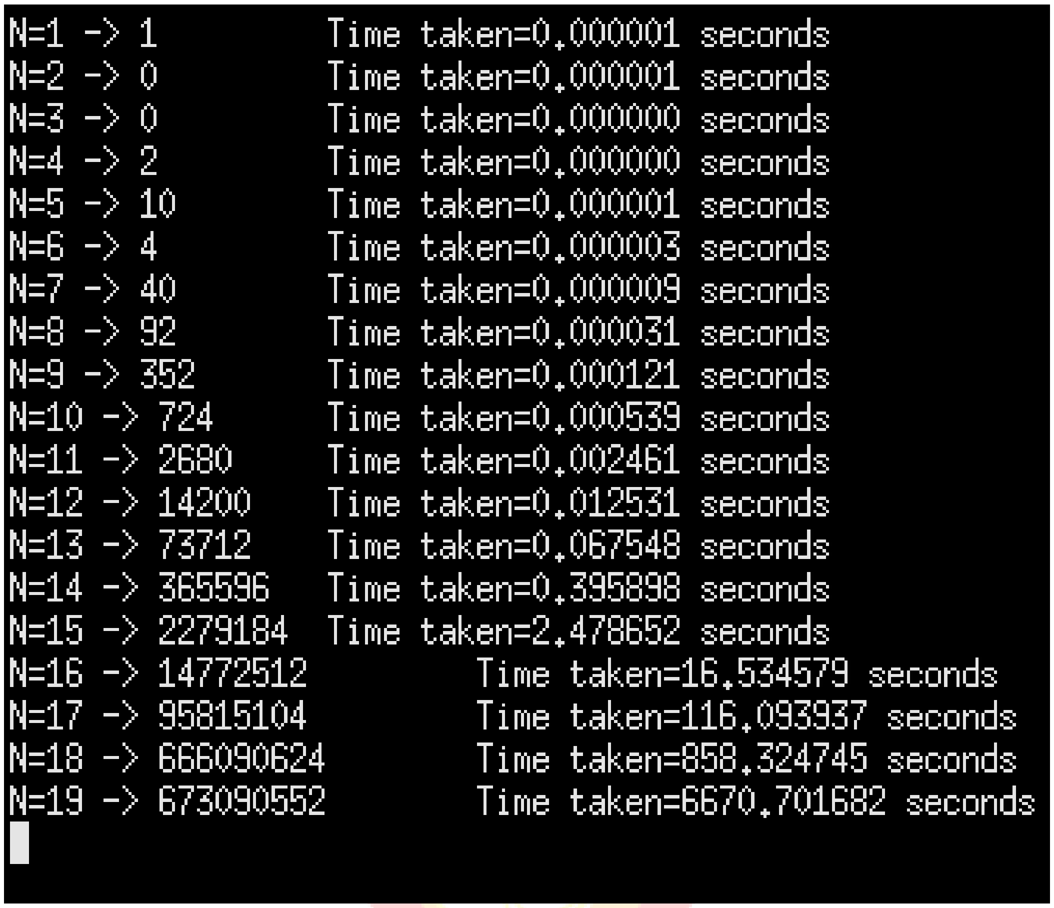
ALGORITHM:

1. Place the queens column wise, start from the left column.
2. If all queens are placed.
   i. Return true and increment the count variable.
3. Else
   i. Try all the rows in the current column.
   ii. Check if queen can be placed here safely, if yes, mark the current cell in solution matrix as 1 and try to solve the rest of the problem recursively.
   iii. If placing the queen in above steps leads to solution return true.
   iv. If placing the queen in above step does not lead to the solution, BACKTRACK, mark the current cell in solution matrix as 0 and return false.
4. If all rows are tried and nothing worked, return false.

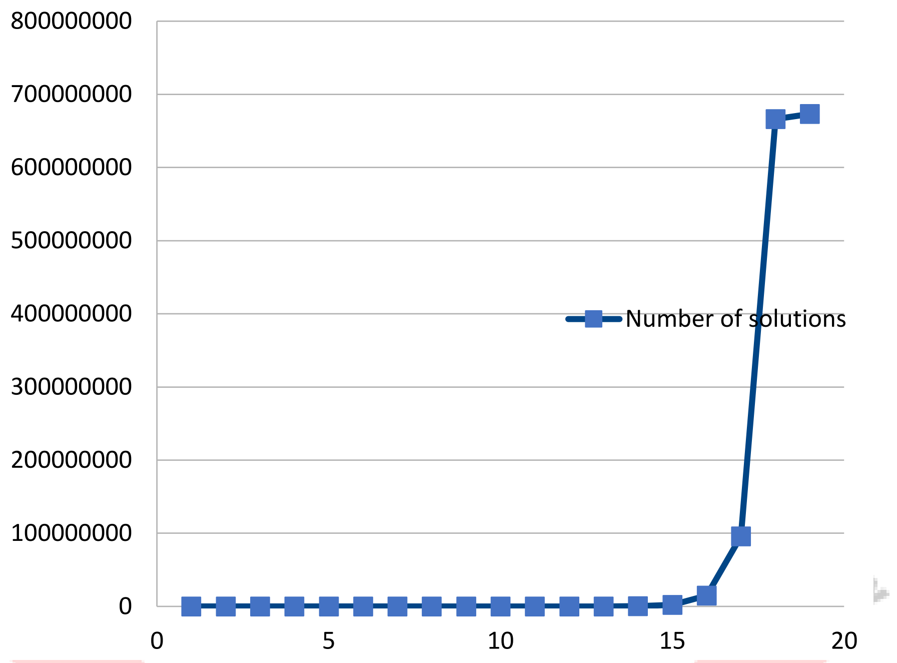
CODE:

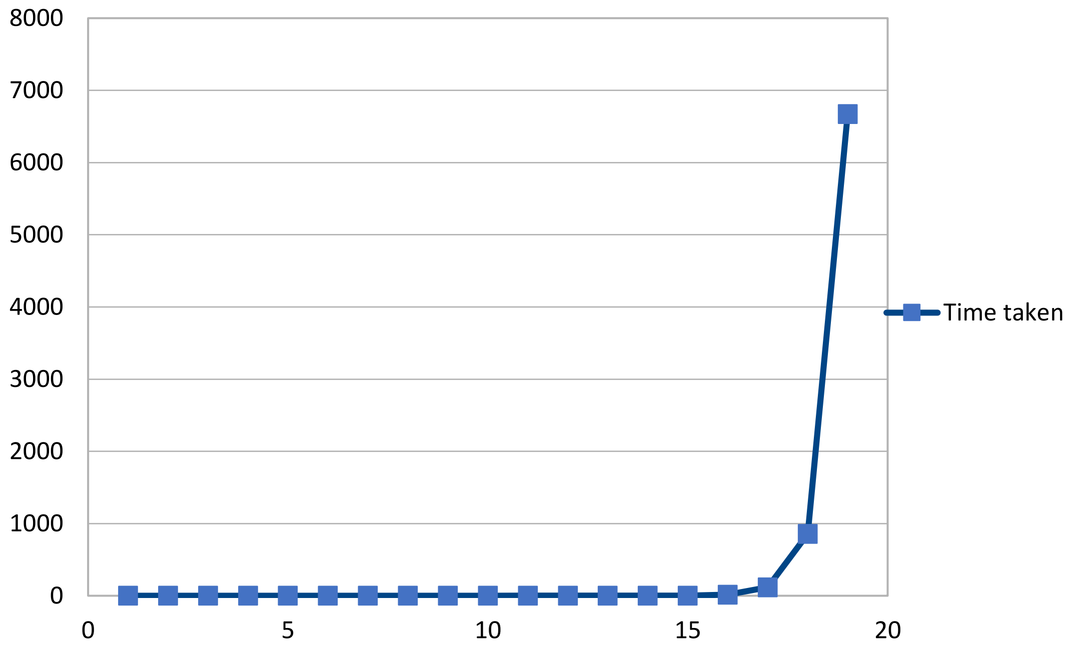
```
#include <stdio.h>
#include <time.h>
int  SIZE, MASK, COUNT;

void Backtrack(int y, int left, int down, int right)
{
   int  bitmap, bit;

   if (y == SIZE) {
      COUNT++;
   } else {
      bitmap = MASK & ~(left | down | right);
      while (bitmap) {
         bit = -bitmap & bitmap;
         bitmap ^= bit;
         Backtrack(y+1, (left | bit)<<1, down | bit, (right |                    bit)>>1);
      }
   }
}
int main(void)
{
   /*  <- N  */
   COUNT = 0;   /* result */
   clock_t begin, end;
   double time_spent = 0.0;
   for(SIZE = 1;SIZE <= 20; SIZE++)
   {
      begin = clock();
      MASK = (1 << SIZE) - 1;
      Backtrack(0, 0, 0, 0);
      end = clock();
      time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
      printf("N=%d -> %d\t Time taken=%lf seconds\n", SIZE, COUNT,           time_spent);
      COUNT = 0;
   }
   return 0;
}
```

RESULTS:



```
N=1 -> 1         Time taken=0.000001 seconds
N=2 -> 0         Time taken=0.000001 seconds
N=3 -> 0         Time taken=0.000000 seconds
N=4 -> 2         Time taken=0.000000 seconds
N=5 -> 10        Time taken=0.000001 seconds
N=6 -> 4         Time taken=0.000003 seconds
N=7 -> 40        Time taken=0.000009 seconds
N=8 -> 92        Time taken=0.000031 seconds
N=9 -> 352       Time taken=0.000121 seconds
N=10 -> 724      Time taken=0.000539 seconds
N=11 -> 2680     Time taken=0.002461 seconds
N=12 -> 14200    Time taken=0.012531 seconds
N=13 -> 73712    Time taken=0.067548 seconds
N=14 -> 365596   Time taken=0.395898 seconds
N=15 -> 2279184  Time taken=2.478652 seconds
N=16 -> 14772512         Time taken=16.534579 seconds
N=17 -> 95815104         Time taken=116.093937 seconds
N=18 -> 666090624        Time taken=858.324745 seconds
N=19 -> 673090552        Time taken=6670.701682 seconds
```


800000000
700000000
600000000
500000000
400000000
300000000
200000000
100000000
0
0
5
10
15
20
Number of solutions

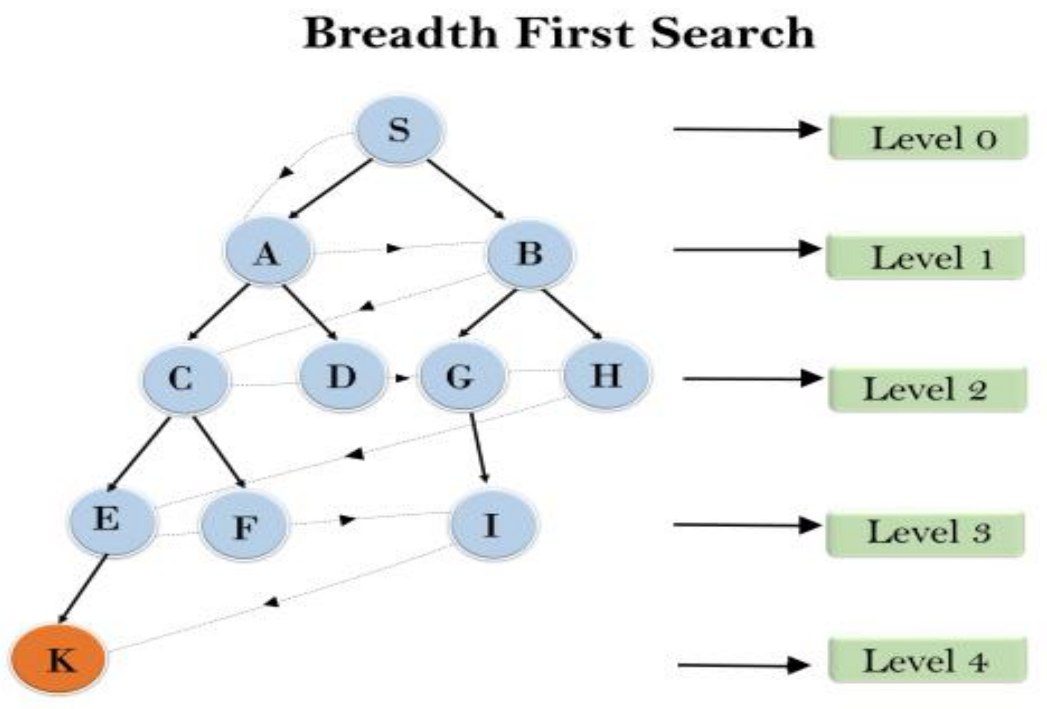

8000
7000
6000
5000
4000
3000
2000
1000
0
0
5
10
15
20
Time taken

EXPERIMENT 2

AIM OF THE EXPERIMENT: To implement graph traversal techniques breadth first search and depth first search.

THEORY:

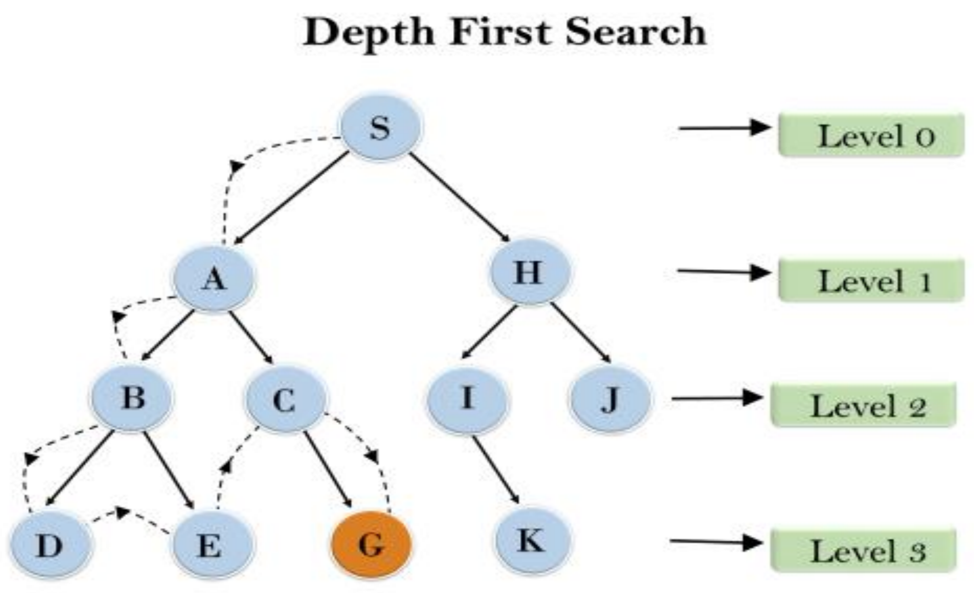
Breadth-first search: Breadth-first search is an algorithm for traversing or searching tree or graph data structure. It starts at tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.
It uses the opposite strategy as depth-first search, which instead explores the node branch as far as possible before being forced to backtrack and expand other nodes.



Advantages of breadth first search:

1. Breadth first search will not get trapped exploring a blind alley. This contrast with depth first searching, which may follow a single, unfruitful path for a very long time, perhaps forever, before the path actually terminates in a state that has no successors. This is a particular problem in depth-first search if there are loops (i.e., a state has a successor that is also one of its ancestors) unless special care is expended to test for such a situation.
2. If there is a solution, then breadth-first search is guaranteed to find it. Furthermore, if there are multiple solutions, then a minimal solution (i.e. one that requires the minimum number of steps) will be found. This is guaranteed by the fact that longer paths are never explored until all shorter ones have already been examined. This contrast with depth first search, which may find a long path to a solution in one part of the tree, when a shorter path exists in some other, unexplored part of the tree.

Depth-first search: Depth first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node



(selecting some arbitrary node as the root node in case of a graph) and explores as far as possible along each branch before backtracking.

Advantages of depth-first search:

1. Depth first search requires less memory since only the nodes on the current path are stored. This contrasts with breadth-first search, where all of the tree that has so far been generated must be stored.
2. By chance (or if care is taken in ordering the alternative successor states), depth-first search may find a solution without examining much of the search space at all. This contrasts with breadth-first search in which all parts if the tree must be examined to level n before any nodes on level n + 1 can be examined. This is particularly significant if many acceptable solutions exist. Depth-first search can stop when one of them is found.

ALGORITHM:

Breadth-first search:

1. Create a variable called NODE-LIST and set it to the initial state.
2. Until a goal state is found or NODE-LIST is empty:
   a) Remove the first element fro NODE-LIST and call it E. If NODE-LIST was empty, quit.
   b) For each way that each rule can match the state described in E do:
      i. Apply the rule to generate a new state.
      ii. If the new state is goal state, quit and return this state.
      iii. Otherwise, add the new state to the end of NODE-LIST.

Depth-first search:

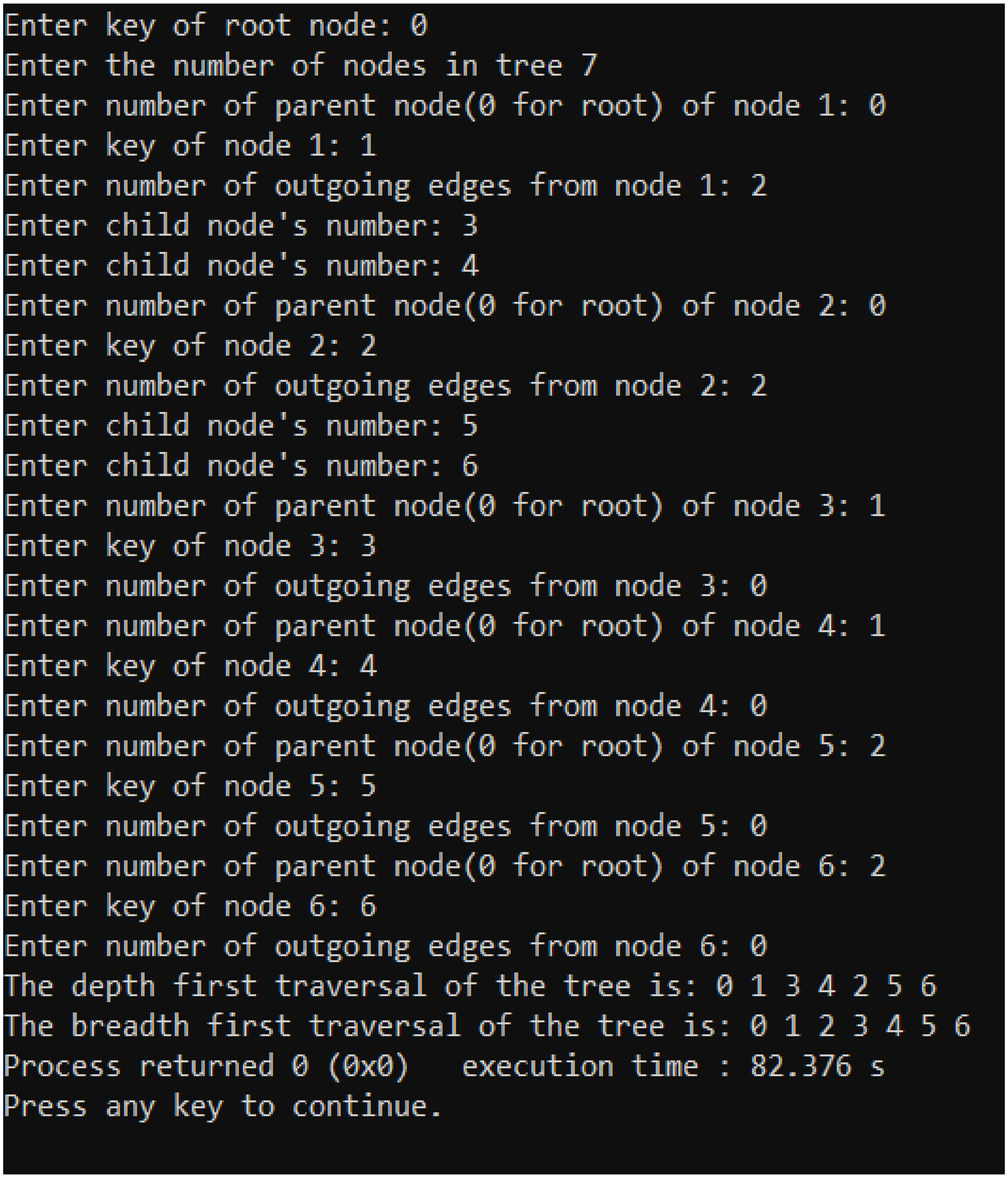
1. If the initial state is goal state, quit and return success.
2. Otherwise, do the following until success or failure is signaled:
   a) Generate a successor, E, of the initial state. If there are no more successors, signal failure.
   b) Call Depth-first search with E as initial state.
   c) If success is returned, signal success. Otherwise continue in this loop.

CODE:

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

struct node
{
   int key;
   vector<node*> next;
   node* back;
};

class tree
{
   node root;
   vector<node> otherNodes;
public:
   tree();
   void initialize(node &currNode, int i);
   void dfs();
   node pop(vector<node> &vect);
   void dfsUtil(node toPush);
   void bfs();
   void bfsUtil(node toEnqueue);
};

tree::tree()
{
   root.back = NULL;
   cout<<"Enter key of root node: ";
   int key;
   cin>>key;
   root.key = key;
   cout<<"Enter the number of nodes in tree ";
   int num;
   cin>>num;
   if(num == 1)
      root.next.push_back(NULL);
```

```
    else
    {
        for(int i = 0; i < num; i++)
        {
            node tempNode;
            otherNodes.push_back(tempNode);
        }
        for(int i = 0; i < num - 1; i++)
        {
            initialize(otherNodes[i], i+1);
        }
    }
}

void tree::initialize(node &currNode, int i)
{
    cout<<"Enter number of parent node(0 for root) of node                    "<<i<<": ";
    int parent;
    cin>>parent;
    if(parent == 0)
    {
        currNode.back = &root;
        root.next.push_back(&currNode);
    }
    else
        currNode.back = &otherNodes[parent - 1];
    cout<<"Enter key of node "<<i<<": ";
    int key;
    cin>>key;
    currNode.key = key;
    int outgoing;
    cout<<"Enter number of outgoing edges from node                           "<<i<<": ";
    cin>>outgoing;
    for(int j = 0; j < outgoing; j++)
    {
        int p;
        cout<<"Enter child node's number: ";
        cin>>p;
        if(p == 0)
            currNode.next.push_back(NULL);
        else
            currNode.next.push_back(&otherNodes[p-1]);
    }
}

void tree::dfs()
{
    dfsUtil(root);
}

node tree::pop(vector<node> &vect)
{
    node temp = vect.back();
    vect.pop_back();
    return temp;
}

void tree::dfsUtil(node toPush)
{
    static vector<node> stack;
    stack.push_back(toPush);
    cout<<toPush.key<<" ";
    for(int i = 0; i < toPush.next.size(); i++)
    {
        if(toPush.next[i] != NULL)
            dfsUtil(*(toPush.next[i]));
        else
        {
            dfsUtil(pop(stack));
        }
    }
}

void tree::bfs()
{
    bfsUtil(root);
}

void tree::bfsUtil(node toEnqueue)
{
    static vector<node*> queue;
    queue.push_back(&toEnqueue);
    while(!queue.empty())
    {
        node temp = *(queue.front());
        queue.erase(queue.begin());
        for(int i = 0; i < temp.next.size(); i++)
        {
            queue.push_back(temp.next[i]);
        }
        cout<<temp.key<<" ";
    }
}

int main()
{
    tree T;
    cout<<"The depth first traversal of the tree is: ";
    T.dfs();
    cout<<endl<<"The breadth first traversal of the tree is: ";
    T.bfs();
    return 0;
}
```

RESULTS:



```
Enter key of root node: 0
Enter the number of nodes in tree 7
Enter number of parent node(0 for root) of node 1: 0
Enter key of node 1: 1
Enter number of outgoing edges from node 1: 2
Enter child node's number: 3
Enter child node's number: 4
Enter number of parent node(0 for root) of node 2: 0
Enter key of node 2: 2
Enter number of outgoing edges from node 2: 2
Enter child node's number: 5
Enter child node's number: 6
Enter number of parent node(0 for root) of node 3: 1
Enter key of node 3: 3
Enter number of outgoing edges from node 3: 0
Enter number of parent node(0 for root) of node 4: 1
Enter key of node 4: 4
Enter number of outgoing edges from node 4: 0
Enter number of parent node(0 for root) of node 5: 2
Enter key of node 5: 5
Enter number of outgoing edges from node 5: 0
Enter number of parent node(0 for root) of node 6: 2
Enter key of node 6: 6
Enter number of outgoing edges from node 6: 0
The depth first traversal of the tree is: 0 1 3 4 2 5 6
The breadth first traversal of the tree is: 0 1 2 3 4 5 6
Process returned 0 (0x0)   execution time : 82.376 s
Press any key to continue.
```

EXPERIMENT 3

AIM OF THE EXPERIMENT: To implement Tic-Tac-Toe using minimax algorithm.

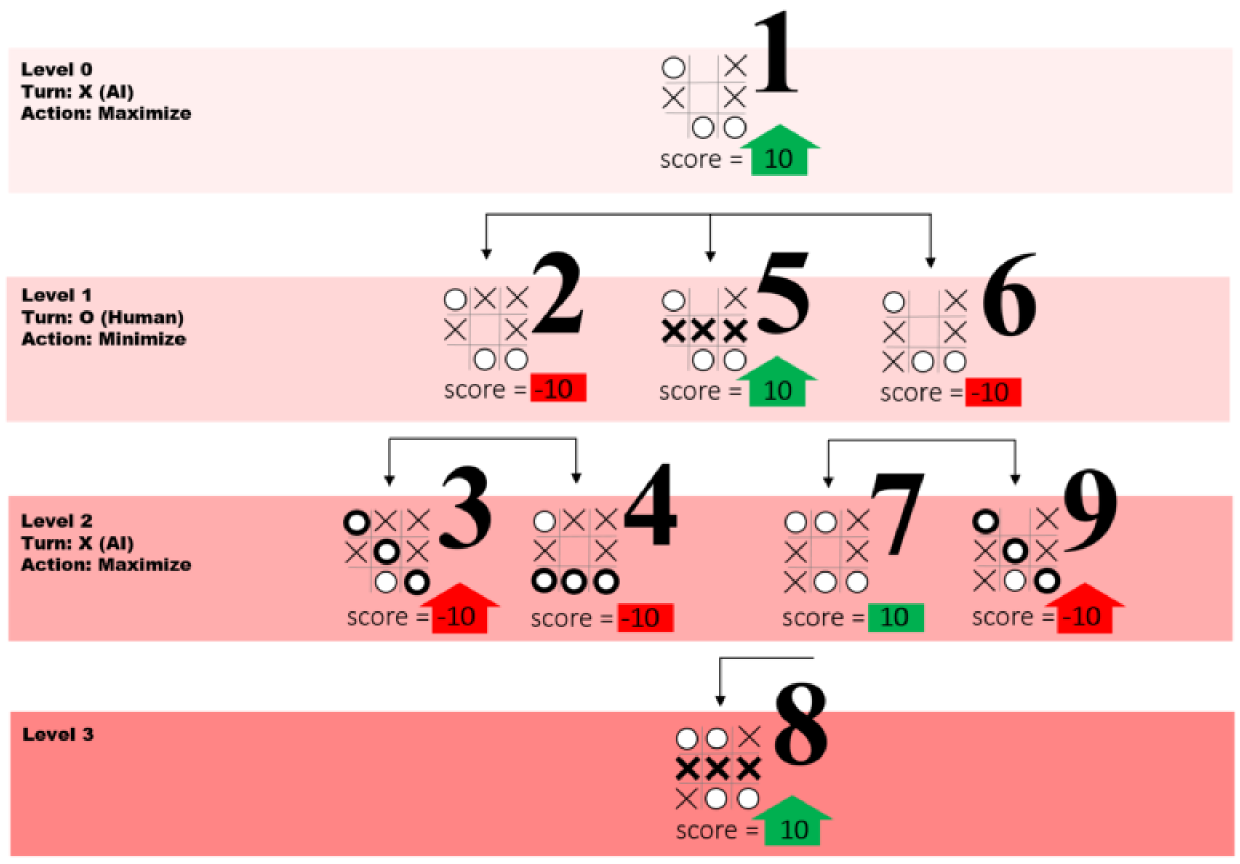
THEORY:

Minimax is a decision rule used in artificial intelligence, decision theory, game theory, statistics and philosophy for minimizing the possible loss for a worst case (maximum loss) scenario. When dealing with gains, it is referred to as ‘maximin’ - to maximize the minimum gain. Originally formulated for two-player zero-sum game-theory, covering both the cases where players take alternate moves and those where they make simultaneous moves, it has also been extended to more complex games and to general decision-making in the presence of uncertainty.

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 minimizer tries to get the highest score possible while the maximizer tries to do the opposite and get the lowest score possible.

Every board has a value associated with it. In a given state if the maximizer has upper hand the, 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.


Level 0
Turn: X (AI)
Action: Maximize
1
score = 10
Level 1
Turn: O (Human)
Action: Minimize
2
score = -10
5
score = 10
6
score = -10
Level 2
Turn: X (AI)
Action: Maximize
3
score = -10
4
score = -10
7
score = 10
9
score = -10
Level 3
8
score = 10

ALGORITHM:

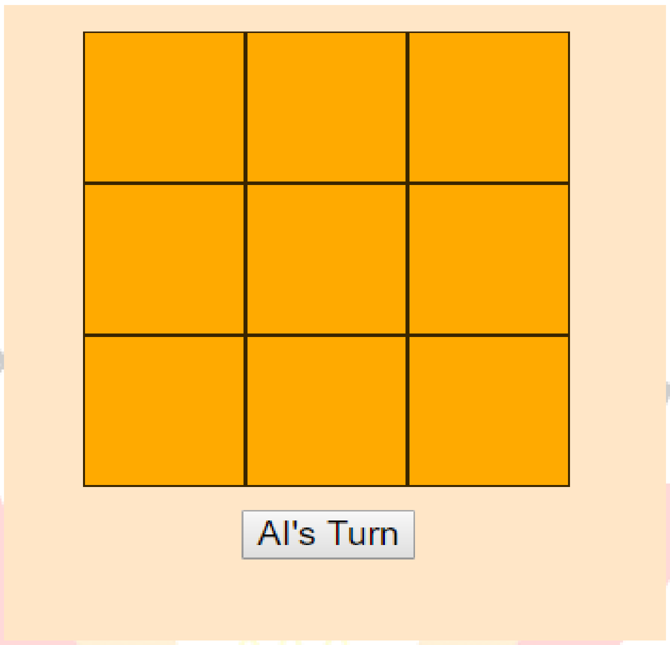
```
# @player is the turn taking player
def score(game)
  if game.win?(@player)
    return 10
  elsif game.win?(@opponent)
    return -10
  else
    return 0
  end
end
def minimax(game)
  return score(game) if game.over?
  scores = [] # an array of scores
  moves = []  # an array of moves

  # Populate the scores array, recursing as needed
  game.get_available_moves.each do |move|
    possible_game = game.get_new_state(move)
    scores.push minimax(possible_game)
    moves.push move
  end

  # Do the min or the max calculation
  if game.active_turn == @player
    # This is the max calculation
    max_score_index = scores.each_with_index.max[1]
    @choice = moves[max_score_index]
    return scores[max_score_index]
  else
    # This is the min calculation
    min_score_index = scores.each_with_index.min[1]
    @choice = moves[min_score_index]
    return scores[min_score_index]
  end
end
```

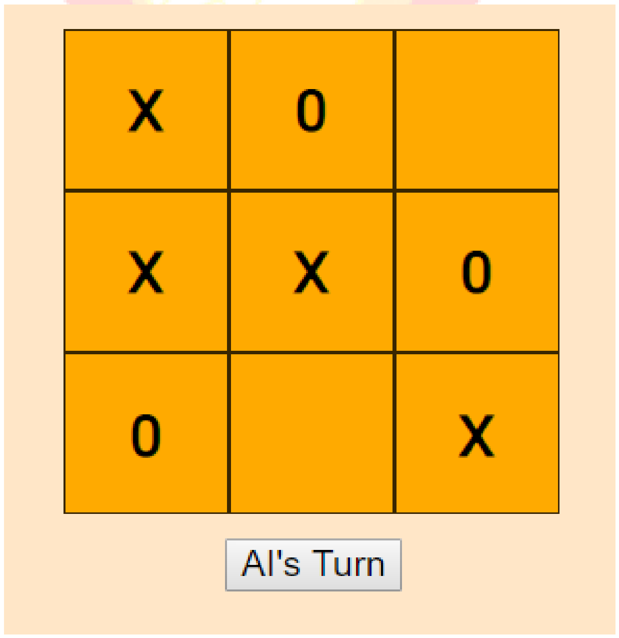
CODE:

```
let boxArray = [];
for (let i = 1; i < 10; ++i) {
    let str = "tictac";
    str += i;
    boxArray.push(document.getElementById(str));
}
let board = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]
];
function makeBoard() {
    for (let i = 0; i < 3; ++i) {
        for (let j = 0; j < 3; ++j) {
            if (boxArray[3 * i + j].innerHTML === " ") {
                board[i][j] = 0;
            }
            else if (boxArray[3 * i + j].innerHTML === "X") {
                board[i][j] = 1;
            }
            else if (boxArray[3 * i + j].innerHTML === "O") {
                board[i][j] = -1;
            }
        }
    }
}
let turnCount = 1;
function XO() {
    if (turnCount % 2 === 1) {
        event.srcElement.innerHTML = 'X';
        ++turnCount;
    }
    else {
        event.srcElement.innerHTML = 'O';
        ++turnCount;
    }
    makeBoard();
    if (hasOwon(board))
        window.alert("YOU WON!!");
    else if (isFull(board))
        window.alert("DRAW!!");
    makeBoard();
}
for (let i = 0; i < boxArray.length; ++i) {
    boxArray[i].addEventListener('click', XO);
}
function hasXwon(board) {
    let check = new Set();
    for (let i = 0; i < 3; ++i) {
        check.add(board[i][i]);
    }
    if (check.size === 1 && check.has(1)) {
        return true;
    }
    check.clear();
    for (let i = 0; i < 3; ++i) {
        let j = 2 - i;
        check.add(board[i][j]);
    }
    if (check.size === 1 && check.has(1)) {
        return true;
    }
    check.clear();
    for (let i = 0; i < 3; ++i) {
        for (let j = 0; j < 3; ++j) {
            check.add(board[i][j]);
        }
        if (check.size === 1 && check.has(1)) {
            return true;
        }
        check.clear();
    }
    for (let i = 0; i < 3; ++i) {
        for (let j = 0; j < 3; ++j) {
            check.add(board[j][i]);
        }
        if (check.size === 1 && check.has(1)) {
            return true;
        }
        check.clear();
    }
}
function hasOwon(board) {
    let check = new Set();
    for (let i = 0; i < 3; ++i) {
        check.add(board[i][i]);
    }
    if (check.size === 1 && check.has(-1)) {
        return true;
    }
    check.clear();
    for (let i = 0; i < 3; ++i) {
        let j = 2 - i;
        check.add(board[i][j]);
    }
    if (check.size === 1 && check.has(-1)) {
        return true;
    }
    check.clear();
    for (let i = 0; i < 3; ++i) {
        for (let j = 0; j < 3; ++j) {
            check.add(board[i][j]);
        }
        if (check.size === 1 && check.has(-1)) {
            return true;
        }
        check.clear();
    }
    for (let i = 0; i < 3; ++i) {
        for (let j = 0; j < 3; ++j) {
            check.add(board[j][i]);
        }
        if (check.size === 1 && check.has(-1)) {
            return true;
        }
        check.clear();
    }
```

```
}
function isFull(board) {
    let check = new Set();
    for (let i = 0; i < 3; i++) {
        for (let j = 0; j < 3; ++j) {
            check.add(board[i][j]);
        }
    }
    if (check.has(0))
        return false;
    else
        return true;
}
function nextMoves(board, XorY) {
    let moves = [];
    for (let i = 0; i < 3; ++i) {
        for (let j = 0; j < 3; ++j) {
            if (board[i][j] !== 0)
                continue;
            else {
                let copyBoard = board.map(inner => inner.slice());
                if (XorY)
                    copyBoard[i][j] = 1;
                else
                    copyBoard[i][j] = -1;
                moves.push(copyBoard);
            }
        }
    }
    return moves;
}
let choiceBoard;
function minimax(board, depth, isXturn) {
    if (hasXwon(board))
        return 10 - depth;
    else if (hasOwon(board))
        return depth - 10;
    else if (isFull(board))
        return 0;
    else {
        ++depth;
        let moves = [];
        let score = [];
        let newMoves = [];
        newMoves = nextMoves(board, isXturn);
        for (let i = 0; i < newMoves.length; ++i) {
            score.push(minimax(newMoves[i], depth, !isXturn));
            moves.push(newMoves[i]);
        }
        if (isXturn) {
            let index = score.indexOf(Math.max(...score));
            choiceBoard = moves[index];
            return score[index];
        }
        else {
            let index = score.indexOf(Math.min(...score));
            choiceBoard = moves[index];
            return score[index];
        }
    }
}
function turnBoard(choiceBoard) {
    ++turnCount;
    for (let i = 0; i < 3; ++i) {
        for (let j = 0; j < 3; ++j) {
            board[i][j] = choiceBoard[i][j];
            if (board[i][j] === 1)
                boxArray[3 * i + j].innerHTML = "X";
            else if (board[i][j] === -1)
                boxArray[3 * i + j].innerHTML = "0";
        }
    }
}
function AIplay() {
    minimax(board, 0, true);
    turnBoard(choiceBoard);
    if (hasXwon(board))
        window.alert("AI won!!");
    else if (isFull(board))
        window.alert("DRAW!!");
}
const AI = document.getElementById("AI");
AI.addEventListener('click', AIplay);
```

RESULTS:





EXPERIMENT 4

AIM OF THE EXPERIMENT: To implement a software project in Prolog.

THEORY:

Prolog is a logic programming language associated with artificial intelligence and computational linguistics.
Prolog has its roots in first-order logic, a formal logic, and unlike many other programming languages, Prolog is intended primarily as a declarative programming language: the program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a query over these relations.
The language was first conceived by Alain Colmerauer and his group in Marseilles, France, in the early 1970s and the first prolog system was developed in 1972 by Colmerauer with Philippe Roussel.
Prolog was one of the first logic programming languages, and remains the most popular among such languages today, with several free and commercial implementations available. The language has been used for theorem proving, expert systems, term rewriting, type systems, and automated planning, as well as its original intended field of use, natural language processing. Modern Prolog environments support the creation of graphical user interface, as well as administrative and networked applications.
Prolog is well-suited for specific tasks that benefit from rule-based logical queries such as searching databases, voice control systems, and filling templates.
In Prolog, programming logic is expressed in terms of relations, and a computation is initiated by running a query over these relations. Relations and queries are constructed using Prolog’s single data type, the term. Relations are defined by clauses. Given a query, the Prolog engine attempts to find a resolution refutation of the negated query. If the negated query can be refuted i.e., an instantiation for all free variables can be found that makes

the union of clauses and the singleton set consisting of the negated query false, it follows that the original query, with the found instantiation applied, is a logical consequence of the program. This makes Prolog (and other logic programming languages) particularly useful for database, symbolic mathematics, and language parsing applications. Because Prolog allows impure predicates, checking the truth value of certain special predicates may have some deliberate side effects, such as printing a value to screen. Because of this, the programmer is permitted to use some amount of conventional imperative programming when the logical paradigm is inconvenient. It has a purely logical subset, called “pure Prolog”, as well as a number of extra logical features.

Data Types

Prolog’s single data type is the term. Terms are either atoms, numbers, variables or compound terms.

- An atom is a general-purpose name with no inherent meaning. Examples of atoms include x, red, ‘Taco’, and ‘some atom’.
- Numbers can be floats or integers. ISO standard compatible Prolog systems can check the Prolog flag “bounded”. Most of the major Prolog systems support arbitrary length integer numbers.
- Variables are denoted by a string consisting of letters, numbers and underscore characters, and beginning with an upper case letter or underscore. Variables closely resemble variables in logic in that they are placeholders for arbitrary terms.
- A compound term is composed of an atom called a “functor” and a number of arguments, which are again terms.

Searching a Maze

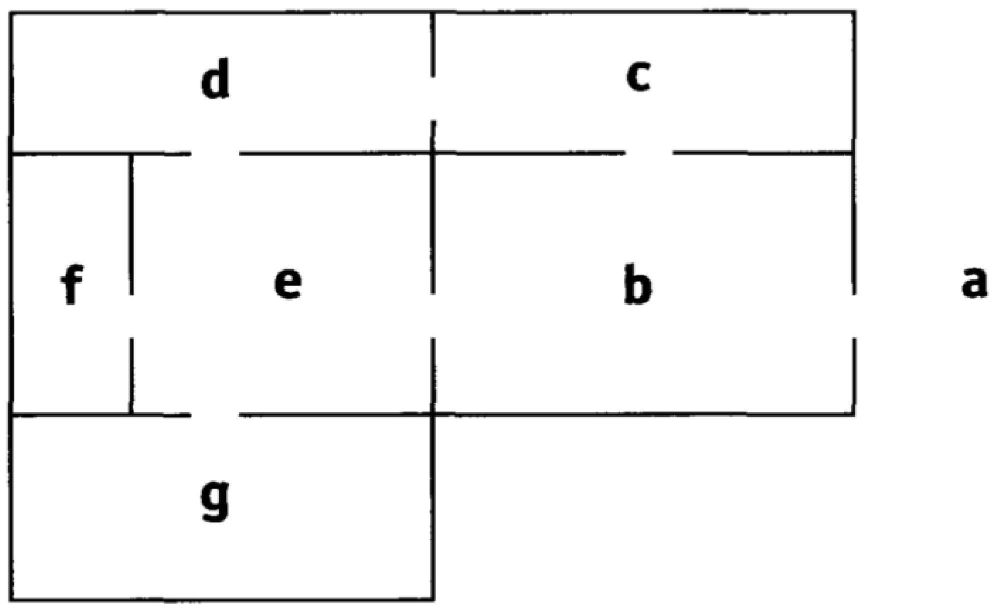
It is a dark and stormy night. As you drive down a lonely country road, your car breaks down, and you stop in front of a splendid palace. You go to the door, find it open, and begin looking for a telephone. How do you search the palace without getting lost? How do you know that you have searched every room? Also, what is the shortest path to the telephone? It is for such situations that maze-searching methods have been devised.

In many computer programs, such as those for searching mazes, it is useful to keep lists of information, and search the list if some information is needed at later time. For example, if we decide to search the palace for a telephone, we might need to keep a list of room numbers visited so far, so we don't go round in circles visiting the same rooms over and over again. What we do is to write down the room numbers visited on a piece of paper. Before entering the room, we check to see if its number is on our piece of paper. If it is, we ignore the room, since we must have been to it previously. If the room number is not on the paper, we write down the number, and enter the room. And so on until we find the telephone.

The steps required to solve the problems are:

1. Go to the door of any room.
2. If the room number is on our list, ignore the room and go to Step 1. If there are no room in sight, then "backtrack" through the room we went through previously, to see if there are any other rooms near it.
3. Otherwise, add the room number to our list.
4. Look in the room for a telephone.
5. If there is no telephone, go to Step 1. Otherwise we stop, and our list has path we took to come to the correct room.

```
d(a, b).
d(b, e).
d(b, c).
d(d, e).
d(c, d).
d(e, f).
d(g, e).
```



CODE:

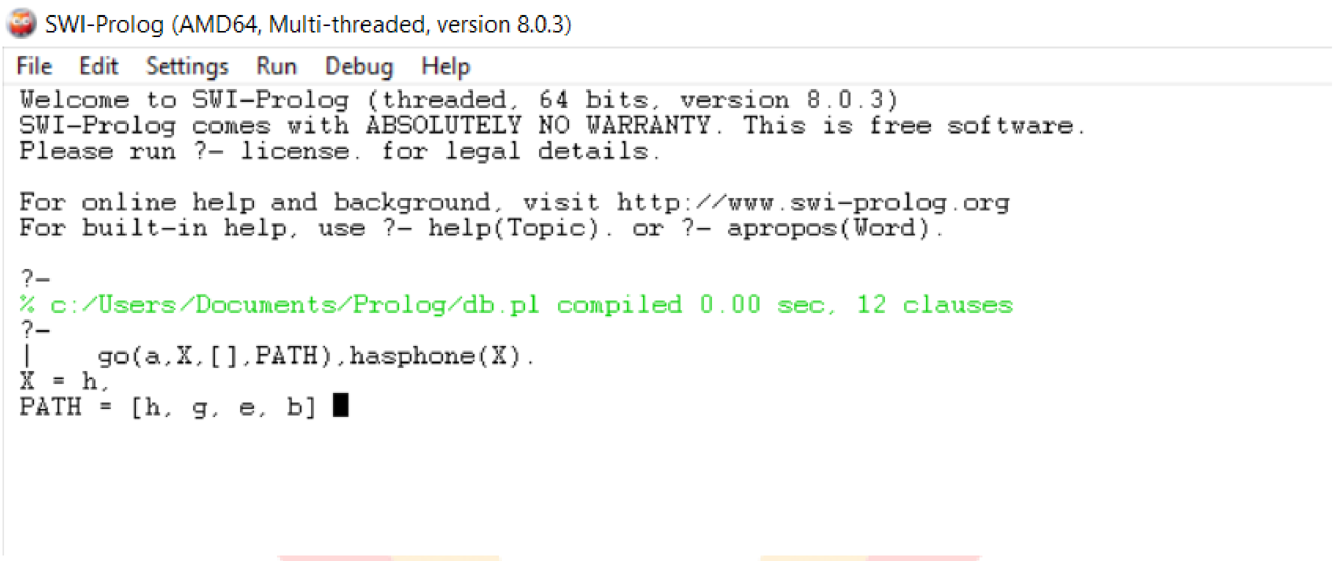
Knowledge Base:

```
door(a, b).
door(b, e).
door(b, c).
door(d, e).
door(c, d).
door(g, e).
door(g, h).
door(e, f).

go(From, To, Path):-
  go(From, To, [], Path).

go(X, X, T, T).
go(X, Y, T, NT) :-
    (door(X,Z) ; door(Z, X)),
    \+ member(Z,T),
    go(Z, Y, [Z|T], NT).

hasphone(h).
```

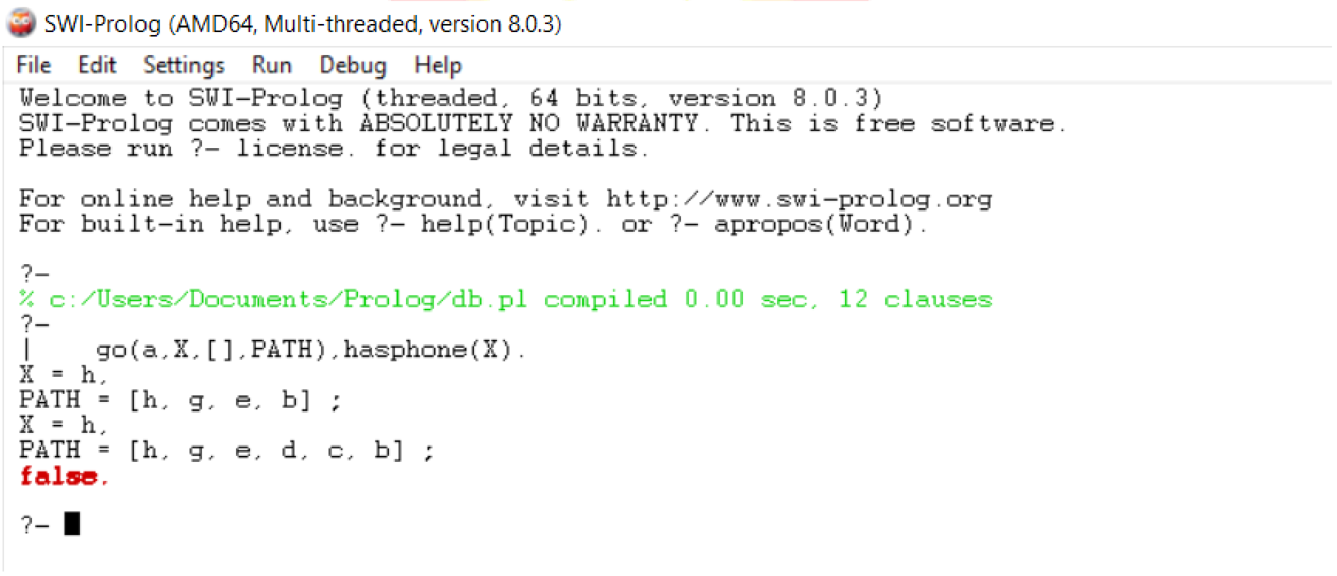
Function Call:

```
go(a,X,[],PATH),hasphone(X).
```

RESULTS:


SWI-Prolog (AMD64, Multi-threaded, version 8.0.3)
File Edit Settings Run Debug Help

```
Welcome to SWI-Prolog (threaded, 64 bits, version 8.0.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
% c:/Users/Documents/Prolog/db.pl compiled 0.00 sec, 12 clauses
?-
|    go(a,X,[],PATH),hasphone(X).
X = h,
PATH = [h, g, e, b]
```


SWI-Prolog (AMD64, Multi-threaded, version 8.0.3)
File Edit Settings Run Debug Help

```
Welcome to SWI-Prolog (threaded, 64 bits, version 8.0.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
% c:/Users/Documents/Prolog/db.pl compiled 0.00 sec, 12 clauses
?-
|    go(a,X,[],PATH),hasphone(X).
X = h,
PATH = [h, g, e, b] ;
X = h,
PATH = [h, g, e, d, c, b] ;
false.

?-
```

EXPERIMENT 5

AIM OF THE EXPERIMENT: To implement a software project in Lisp.

THEORY:

Lisp (historically LISP) is a family of computer programming languages with a long history and a distinctive, fully parenthesized prefix notation. Originally, specified in 1958, Lisp is the second-oldest high-level programming language in widespread use today. Only Fortran is older by one year. Lisp has changed since its early days, and many dialects have existed over its history. Today, the best-known general-purpose Lisp dialects are Clojure, Common Lisp and Scheme.
Lisp was originally created as a practical mathematical notation for computer programs, influenced by the notation of Alonzo Church's lambda calculus. It quickly became the favored programming language for artificial intelligence (AI) research. As one of the earliest programming languages, Lisp pioneered many ideas in computer science, including tree-data structures, automatic storage management, dynamic typing, conditionals, higher-order functions, recursion, the self-hosting compiler, and the read-eval-print-loop.
The name LISP derives from "List Processor". Linked lists are one of Lisp's major data structures, and the Lisp source code is made of lists. Thus, Lisp programs can manipulate source code as a data structure, giving rise to the macro systems that allow programmers to create new syntax or new domain-specific languages embedded in Lisp.
The interchangeability of code and data gives Lisp its instantly recognizable syntax. All program code is written as s-expressions, or parenthesized lists. A function call or syntactic form is written as a list with the function or operator's name first, and the arguments following: for instance, a function f that takes three arguments would be called as (f arg1 arg2 arg3).
Since inception, Lisp was closely connected with the artificial intelligence research community, especially on PDP-10 systems. Lisp was used as the implementation of the programming language Macro Planner, which was

used in the famous AI system SHRDLU. In the 1970s, as AI research spawned commercial offshoots, the performance of existing Lisp systems became a growing issue.
Lisp is an expression oriented language. Unlike most other languages, no distinction is made between "expressions" and "statements";[dubious – discuss] all code and data are written as expressions. When an expression is evaluated, it produces a value (in Common Lisp, possibly multiple values), which can then be embedded into other expressions. Each value can be any data type.

McCarthy's 1958 paper introduced two types of syntax: Symbolic expressions (S-expressions, sexps), which mirror the internal representation of code and data; and Meta expressions (M-expressions), which express functions of S-expressions. M-expressions never found favor, and almost all Lisps today use S-expressions to manipulate both code and data.

The use of parentheses is Lisp's most immediately obvious difference from other programming language families. As a result, students have long given Lisp nicknames such as Lost In Stupid Parentheses, or Lots of Irritating Superfluous Parentheses.[52] However, the S-expression syntax is also responsible for much of Lisp's power: the syntax is extremely regular, which facilitates manipulation by computer. However, the syntax of Lisp is not limited to traditional parentheses notation. It can be extended to include alternative notations. For example, XMLisp is a Common Lisp extension that employs the metaobject protocol to integrate S-expressions with the Extensible Markup Language (XML).

The reliance on expressions gives the language great flexibility. Because Lisp functions are written as lists, they can be processed exactly like data. This allows easy writing of programs which manipulate other programs (metaprogramming). Many Lisp dialects exploit this feature using macro systems, which enables extension of the language almost without limit.
A Lisp list is written with its elements separated by whitespace, and surrounded by parentheses. For example, (1 2 foo) is a list whose elements are the three atoms 1, 2, and foo. These values are implicitly typed: they are

respectively two integers and a Lisp-specific data type called a "symbol", and do not have to be declared as such.

The empty list () is also represented as the special atom nil. This is the only entity in Lisp which is both an atom and a list.

Expressions are written as lists, using prefix notation. The first element in the list is the name of a function, the name of a macro, a lambda expression or the name of a "special operator" (see below). The remainder of the list are the arguments. For example, the function list returns its arguments as a list, so the expression

```
 (list 1 2 (quote foo))
```

evaluates to the list (1 2 foo). The "quote" before the foo in the preceding example is a "special operator" which returns its argument without evaluating it. Any unquoted expressions are recursively evaluated before the enclosing expression is evaluated. For example,

```
 (list 1 2 (list 3 4))
```

evaluates to the list (1 2 (3 4)). Note that the third argument is a list; lists can be nested.
Arithmetic operators are treated similarly. The expression

```
 (+ 1 2 3 4)
```

evaluates to 10. The equivalent under infix notation would be "1 + 2 + 3 + 4".

Lisp has no notion of operators as implemented in Algol-derived languages. Arithmetic operators in Lisp are variadic functions (or n-ary), able to take any number of arguments. A C-style '++' increment operator is sometimes implemented under the name incf giving syntax

```
 (incf x)
```

equivalent to (setq x (+ x 1)), returning the new value of x.

"Special operators" (sometimes called "special forms") provide Lisp's control structure. For example, the special operator if takes three arguments. If the first argument is non-nil, it evaluates to the second argument; otherwise, it evaluates to the third argument. Thus, the expression

```
 (if nil
     (list 1 2 "foo")
     (list 3 4 "bar"))
```

evaluates to (3 4 "bar"). Of course, this would be more useful if a non-trivial expression had been substituted in place of nil.

Lisp also provides logical operators and, or and not. The and and or operators do short circuit evaluation and will return their first nil and non-nil argument respectively.

```
 (or (and "zero" nil "never") "James" 'task 'time)
```

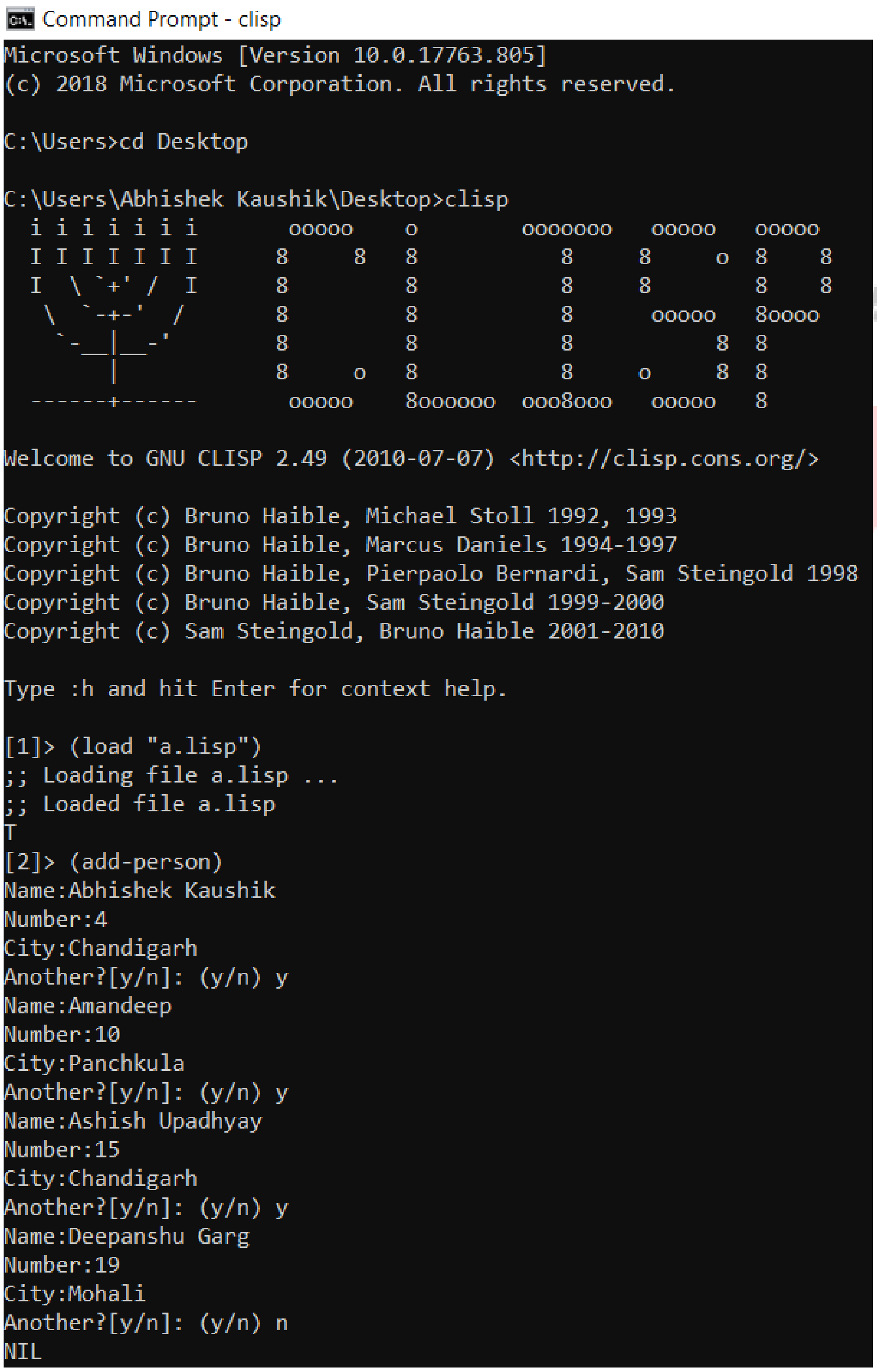
will evaluate to "James".

CODE:

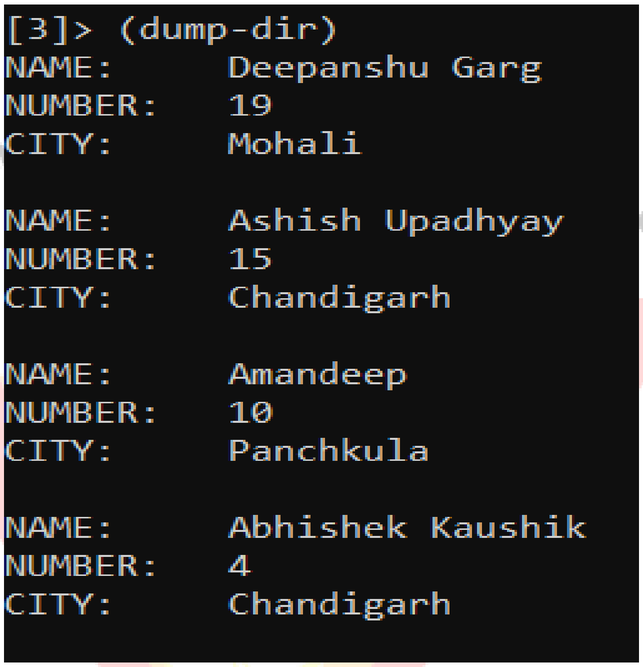
```
(defun make-record (name number city)
  (list :name name :number number :city city))

(defvar *DIR* nil)

(defun add-record (person)
  (push person *dir*))

(defun dump-dir ()
  (dolist (person *dir*)
  (format t "~{~a:~10t~a~%~}~%" person)))

(defun prompt-read (prompt)
  (format *query-io* "~a:" prompt)
  (force-output *query-io*)
  (read-line *query-io*))

(defun prompt-for-person ()
  (make-record
    (prompt-read "Name")
    (prompt-read "Number")
    (prompt-read "City")))

(defun save-db (filename)
 (with-open-file (out filename
                  :direction :output
                  :if-exists :supersede)
 (with-standard-io-syntax
   (print *dir* out))))

(defun load-db (filename)
  (with-open-file (in filename)
 (with-standard-io-syntax
 (setf *dir* (read in)))))

(defun select (selector-fn)
 (remove-if-not selector-fn *dir*))

(defun update (selector-fn &key name number city)
 (setf *dir*
  (mapcar
              #'(lambda (row)
     (when (funcall selector-fn row)
      (if name (setf (getf row :name) name))
      (if number (setf (getf row :number) number))
      (if city (setf (getf row :city) city)))row) *dir*)))

(defun delete-rows (selector-fn)
 (setf *dir* (remove-if selector-fn *dir*)))

(defun make-comparison-expr (field value)
  `(equal (getf person,field),value))

(defun make-comparisons-lists (fields)
 (loop while fields
  collecting (make-comparison-expr (pop fields) (pop fields))))

(defun add-person ()
 (loop (add-record (prompt-for-person))
  (if (not (y-or-n-p "Another?[y/n]:"))(return))))

(defun select-by-name (name)
 (remove-if-not
       #'(lambda (person)(equal (getf person :name)name)) *dir*))
```

```
(defmacro where (&rest clauses)
  `#'(lambda (person)(and,@(make-comparisons-lists clauses))))
```

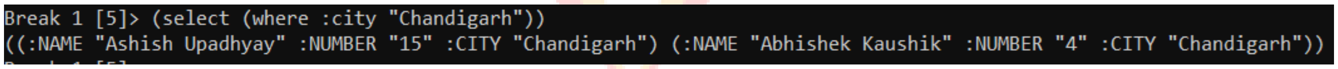
RESULT:


Command Prompt - clisp

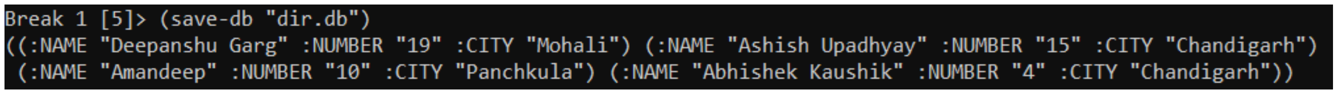
```
Microsoft Windows [Version 10.0.17763.805]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users>cd Desktop

C:\Users\Abhishek Kaushik\Desktop>clisp
  i i i i i i i       ooooo    o        ooooooo   ooooo   ooooo
  I I I I I I I      8     8   8           8     8     o  8    8
  I  \ `+' /  I      8         8           8     8        8    8
   \  `-+-'  /       8         8           8      ooooo   8oooo
    `-__|__-'        8         8           8           8  8
        |            8     o   8           8     o     8  8
  ------+------       ooooo    8oooooo  ooo8ooo   ooooo   8

Welcome to GNU CLISP 2.49 (2010-07-07) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010

Type :h and hit Enter for context help.

[1]> (load "a.lisp")
;; Loading file a.lisp ...
;; Loaded file a.lisp
T
[2]> (add-person)
Name:Abhishek Kaushik
Number:4
City:Chandigarh
Another?[y/n]: (y/n) y
Name:Amandeep
Number:10
City:Panchkula
Another?[y/n]: (y/n) y
Name:Ashish Upadhyay
Number:15
City:Chandigarh
Another?[y/n]: (y/n) y
Name:Deepanshu Garg
Number:19
City:Mohali
Another?[y/n]: (y/n) n
NIL
```



```
[3]> (dump-dir)
NAME:      Deepanshu Garg
NUMBER:    19
CITY:      Mohali

NAME:      Ashish Upadhyay
NUMBER:    15
CITY:      Chandigarh

NAME:      Amandeep
NUMBER:    10
CITY:      Panchkula

NAME:      Abhishek Kaushik
NUMBER:    4
CITY:      Chandigarh
```



```
Break 1 [5]> (select (where :city "Chandigarh"))
((:NAME "Ashish Upadhyay" :NUMBER "15" :CITY "Chandigarh") (:NAME "Abhishek Kaushik" :NUMBER "4" :CITY "Chandigarh"))
```



```
Break 1 [5]> (save-db "dir.db")
((:NAME "Deepanshu Garg" :NUMBER "19" :CITY "Mohali") (:NAME "Ashish Upadhyay" :NUMBER "15" :CITY "Chandigarh")
 (:NAME "Amandeep" :NUMBER "10" :CITY "Panchkula") (:NAME "Abhishek Kaushik" :NUMBER "4" :CITY "Chandigarh"))
```

EXPERIMENT 6

AIM OF THE EXPERIMENT: To implement unification algorithm in Prolog.

THEORY:

In logic and computer science, unification is an algorithmic process of solving equations between symbolic expressions.

Depending on which expressions (also called terms) are allowed to occur in an equation set (also called unification problem), and which expressions are considered equal, several frameworks of unification are distinguished. If higher-order variables, that is, variables representing functions, are allowed in an expression, the process is called higher-order unification, otherwise first-order unification. If a solution is required to make both sides of each equation literally equal, the process is called syntactic or free unification, otherwise semantic or equational unification, or E-unification, or unification modulo theory.

A solution of a unification problem is denoted as a substitution, that is, a mapping assigning a symbolic value to each variable of the problem's expressions. A unification algorithm should compute for a given problem a complete, and minimal substitution set, that is, a set covering all its solutions, and containing no redundant members. Depending on the framework, a complete and minimal substitution set may have at most one, at most finitely many, or possibly infinitely many members, or may not exist at all. In some frameworks it is generally impossible to decide whether any solution exists. For first-order syntactical unification, Martelli and Montanari[gave an algorithm that reports unsolvability or computes a complete and minimal singleton substitution set containing the so-called most general unifier.

For example, using x,y,z as variables, the singleton equation set { cons(x,cons(x,nil)) = cons(2,y) } is a syntactic first-order unification problem that has the substitution { $x \mapsto 2$, $y \mapsto$ cons(2,nil) } as its only solution. The syntactic first-order unification problem { y = cons(2,y) } has no solution over the set of finite terms; however, it has the single solution { $y \mapsto$ cons(2,cons(2,cons(2,...))) } over the set of infinite trees. The semantic first-order unification problem { a·x = x·a } has each substitution of the form { $x \mapsto$ a·...·a } as a solution in a semigroup, i.e. if (·) is considered associative; the same problem, viewed in an abelian group, where (·) is considered also commutative, has any substitution at all as a solution. The singleton set { a = y(x) } is a syntactic second-order unification problem, since y is a function variable. One solution is { $x \mapsto a$, $y \mapsto$ (identity function) }; another one is { $y \mapsto$ (constant function mapping each value to a), $x \mapsto$ (any value) }.

A unification algorithm was first discovered by Jacques Herbrand, while a first formal investigation can be attributed to John Alan Robinson, who used first-order syntactical unification as a basic building block of his resolution procedure for first-order logic, a great step forward in automated reasoning technology, as it eliminated one source of combinatorial explosion: searching for instantiation of terms. Today, automated reasoning is still the main application area of unification. Syntactical first-order unification is used in logic programming and programming language type system implementation, especially in Hindley–Milner based type inference algorithms. Semantic unification is used in SMT solvers, term rewriting algorithms and cryptographic protocol analysis. Higher-order unification is used in proof assistants, for example Isabelle and Twelf, and restricted forms of higher-order unification (higher-order pattern unification) are used in some programming language implementations, such as lambdaProlog, as higher-order patterns are expressive, yet their associated unification procedure retains theoretical properties closer to first-order unification.

ALGORITHM:

$G \cup \{t \doteq t\} \Rightarrow G$		delete
$G \cup \{f(s_0,\ldots,s_k) \doteq f(t_0,\ldots,t_k)\} \Rightarrow G \cup \{s_0 \doteq t_0,\ldots,s_k \doteq t_k\}$		decompose
$G \cup \{f(s_0,\ldots,s_k) \doteq g(t_0,\ldots,t_m)\} \Rightarrow \bot$	if $f \neq g$ or $k \neq m$	conflict
$G \cup \{f(s_0,\ldots,s_k) \doteq x\} \Rightarrow G \cup \{x \doteq f(s_0,\ldots,s_k)\}$		swap
$G \cup \{x \doteq t\} \Rightarrow G\{x \mapsto t\} \cup \{x \doteq t\}$	if $x \notin \mathrm{vars}(t)$ and $x \in \mathrm{vars}(G)$	eliminate[note 8]
$G \cup \{x \doteq f(s_0,\ldots,s_k)\} \Rightarrow \bot$	if $x \in \mathrm{vars}(f(s_0,\ldots,s_k))$	check

Given a finite set G = {$s_1 = t_1$,…, $s_n = t_n$} of potential equations, the algorithm applies rules to transform it to an equivalent set of equations of the form { x1 ≐ u1, ..., xm ≐ um } where x1, ..., xm are distinct variables and u1, ..., um are terms containing none of the xi. A set of this form can be read as a substitution. If there is no solution the algorithm terminates with ⊥; other authors use "Ω", "{}", or "fail" in that case. The operation of substituting all occurrences of variable x in problem G with term t is denoted G {x ↦ t}. For simplicity, constant symbols are regarded as function symbols having zero arguments.

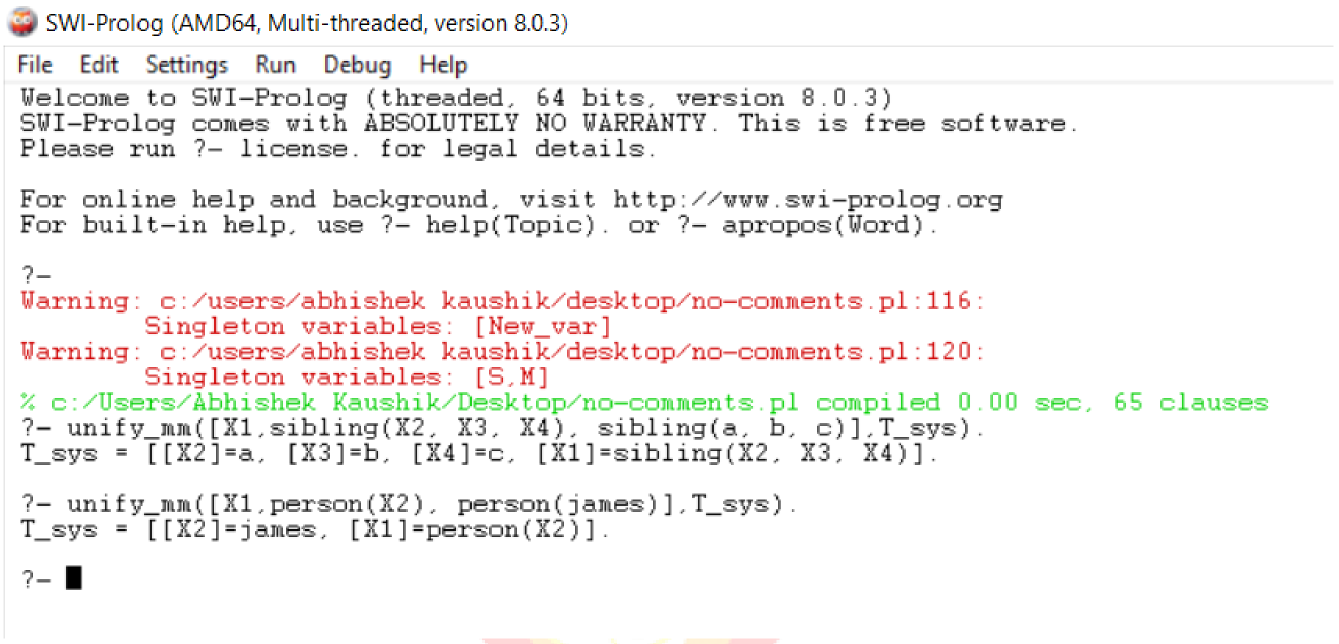
CODE:

```
unify_mm(L,T)          :- prepara(L,Tree,Z,Counter),
unify_sys(Tree,[],T,Z,Counter).


unify_sys(_,T,T,[],0)          :-      !.
unify_sys(_,_,_,[],_)          :-    write('Error: cycle'),
                                    fail, !.
unify_sys(U,T,Ts,[{0,S=[]}|Z],Co)      :-      !, Co_n is Co-1,
unify_sys(U,[S=nil|T],Ts,Z,Co_n).
unify_sys(U,T,Ts,[{0,S=M}|Z],Co)       :-      cpf(M,C,F-[]),
                                    compact(F,U,Uo,Z,Zo,0,El),
                                    Co_n is Co - El -1,
                                    unify_sys(Uo,[S=C|T],Ts,Zo,Co_n).

cpf(T,Fu,Nil-Nil)              :-      functor(T,Fu,0),!.
cpf([],[],[]-[])                :-      !.
cpf([{[]=Mi}|T],[Ci|Ct],F)      :-      !, append_d_l(Fi,Ft,F), cpf(Mi,Ci,Fi),
cpf(T,Ct,Ft).

cpf([{[S|St]=Mi}|T],[S|Ct],F)   :-      !, append_d_l([{[S|St]=Mi}|Nil]-
Nil,Ft,F), cpf(T,Ct,Ft).

cpf(T,C,F)              :-      functor(T,Fu,N), functor(C,Fu,N),
                         T=..[Fu|A1], C=..[Fu|A2],
                         cpf(A1,A2,F).

compact([],U,U,Z,Z,El,El)              :-      !.
compact([{[V|Se]=M}|T],Ui,Uo,Zi,Zo,El_t,El)      :-
                                    m_tree(Ui,V,{Cv,Sv=Mv},Mef,Ut),
                                    Cv1 is Cv - 1,
compact_iter(Se,Ut,{Cv1,Sv=Mv},Mef,Ut1,M,El_t,El_1),
                                    update_zerome(Mef,Zi,Zt),
                                    compact(T,Ut1,Uo,Zt,Zo,El_1,El).

compact_iter([],U,{Ct,St=Mt},{Ct,St=Mf},U,M,El,El)     :-
                                          merge_mt(Mt,M,Mf).
compact_iter([H|T],Ui,{Ct,St=Mt},Mef,Uo,M,El_t,El)     :-
                                          mbchk(H,St), !,
                                          Ct1 is Ct -1,
                                          m_tree(Ui,H,_,Mef,Ut),

compact_iter(T,Ut,{Ct1,St=Mt},Mef,Uo,M,El_t,El).
compact_iter([H|T],Ui,{Ct,St=Mt},Mef,Uo,M,El_t,El)     :-
                                          m_tree(Ui,H,{Ch,Sh=Mh},Mef,Ut),
                                          Ch1 is Ch -1,

merge_me({Ct,St=Mt},{Ch1,Sh=Mh},MeT),
                                          El_tt is El_t + 1,

compact_iter(T,Ut,MeT,Mef,Uo,M,El_tt,El).

update_zerome({0,S=M},Zi,[{0,S=M}|Zi])  :- !.
update_zerome(_,Zi,Zi).

merge_mt([],M2,M2)              :-      !.
merge_mt(M1,[],M1)              :-      !.
merge_mt([{S1i=M1i}|T1],[{S2i=M2i}|T2],[{S3i=M3i}|T3])  :- !,
                                          union_se(S1i,S2i,S3i),
                                          merge_mt(M1i,M2i,M3i),
                                          merge_mt(T1,T2,T3).
merge_mt(M1,M2,M3)                :-      M1=..[F|A1], M2=..[F|A2],
functor(M1,F,N), functor(M2,F,N),
                              functor(M3,F,N), M3=..[F|A3],
                                        merge_mt(A1,A2,A3).

merge_me(M1,M2,M1)                            :-     M1==M2,!.
merge_me({C1,S1=T1},{C2,S2=T2},M3)                :-     length(S1,N1),
                                                  length(S2,N2),
                                                  N1 < N2,

merge_me({C2,S2=T2},{C1,S1=T1},M3).
merge_me({C1,S1=T1},{C2,S2=T2},{C3,S3=T3})     :-     C3 is C1 + C2,
                                                  append(S2,S1,S3), %NON EFFICIENTE
                                                  merge_mt(T1,T2,T3).



trasf_begin(L,{S=R},V)         :-     separa(L,S,M,_), trasf(M,R,V).

trasf([],[],X-X)              :-     !.
trasf([[S,M]|T],[{S=Rm}|Rt],V)  :-     !, append_d_l(Vm,Vt,V),
trasf(M,Rm,Vm), trasf(T,Rt,Vt).

trasf(L,R,V)                  :-     functor_list(L,F,N), args(L,N,[],A,X-X,Vl),
                                        append_d_l(Vl,Vt,V), functor(R,F,N), R=..[F|R1],
                                        trasf(A,R1,Vt).


functor_list([],nil,nil)      :- !.
functor_list(L,F,N)      :- functor_list_1(L,F,N).

functor_list_1([],_,_).
functor_list_1([T|L],F,N)       :- functor(T,F,N), functor_list_1(L,F,N).

arg_list(_,[],[])               :-     !.
arg_list(I,[H|T],[A|At])        :-     arg(I,H,A), arg_list(I,T,At).
```

```
args(_,0,A,A,V,V)       :-      !.
args(L,I,At,A,Vt,V)     :-      arg_list(I,L,Ai), separa(Ai,S,M,Vi),
append_d_l(Vi,Vt,Vtt), I1 is I-1,
                                args(L,I1,[[S,M]|At],A,Vtt,V).

separa([],[],[],Nil-Nil).
separa([H|T],[H|St],M,V)        :-      var(H), !, append_d_l(Vt, [H|L] - L, V),
separa(T,St,M,Vt).
separa([H|T],S,[H|Mt],V)        :-      separa(T,S,Mt,V).



prepara(L,Tree,Z,N)             :-
                                trasf_begin(L,Me,V-[]), msort(V,Vars),
                                build_sys(Vars,nil,0,X-X,Sy-[],0,N1), N is N1+1,
                                sys_tree(Sy,Me,Tree,Z).

build_sys([],nil,0,Sy,Sy,Num,Num)       :- !.
build_sys([],P,C,St,Sy,Num_t,Num)       :-
                                        !, append_d_l(St,[{C,[P]=[]}|L]-L,Sy), Num is
Num_t+1.
build_sys([V|T],nil,0,Syt,Sy,Num_t,Num) :-
                                        !, build_sys(T,V,1,Syt,Sy,Num_t,Num).
build_sys([V|T],P,C,Syt,Sy,Num_t,Num)   :-
                                        V==P,!, C1 is C+1,
build_sys(T,P,C1,Syt,Sy,Num_t,Num).
build_sys([V|T],P,C,Syt,Sy,Num_t,Num)   :-
                                        append_d_l(Syt,[{C,[P]=[]}|L]-L,Sytt), Num_t1 is
Num_t+1,
                                        build_sys(T,V,1,Sytt,Sy,Num_t1,Num).

sys_tree(Sys,{[]=M},Tree,Z)     :-      !, crea_albero(Sys,Tree),
update_zerome({0,[New_var]=M},[],Z).
sys_tree(Sys,{S=M},Tree,Z)      :-      crea_albero(Sys,Tree_t),
counter_me(Tree_t,0,S,{C,S=M},Tree),
                                update_zerome({C,S=M},[],Z).

counter_me(Tree,C,[],{C,S=M},Tree)              :- !.
counter_me(Tree1,Ct,[H|T],{C,S=M},Tree)         :-
                                   m_tree(Tree1,H,{C1,_=_},{C,S=M},Tree2),
Ctt is Ct+C1,
                                   counter_me(Tree2,Ctt,T,{C,S=M},Tree).



crea_albero([],nil)     :- !.
crea_albero(Sy,[V-{C,[V]=M},L,R])       :-
                                dividi(Sy,{C,[V]=M},L1,R1), crea_albero(L1,L),
crea_albero(R1,R).

dividi(Sy,X,L,R)                :-      length(Sy,N), N1 is (N // 2) +1,
dividi_2(Sy,N1,X,D-D,L,R).

dividi_2([H|T],1,H,L-[],L,T)    :-      !.
dividi_2([H|T],N,X,Lt,L,R)      :-      append_d_l(Lt,[H|Y]-Y,Ltt), N1 is N-1,
dividi_2(T,N1,X,Ltt,L,R).


m_tree(nil,_,{0,[]=[]},_,nil)                   :-      !.
m_tree([N-Me,L,R],E,Me,Ms,[N-Ms,L,R])   :-      E==N, !.
m_tree([N-M,L,R],E,Me,Ms,[N-M,L1,R])    :-      E@<N, !,
m_tree(L,E,Me,Ms,L1).
m_tree([N-M,L,R],E,Me,Ms,[N-M,L,R1])    :-      m_tree(R,E,Me,Ms,R1).



mbchk(_,[]) :- fail.
mbchk(E,[H|_]) :- E==H , ! .
mbchk(E,[_|T]) :- mbchk(E,T).
```

```
union_se([],L,L)        :- !.
union_se([H|T], L, R) :-
        mbchk(H, L), !,
        union_se(T, L, R).
union_se([H|T], L, [H|R]) :-
        union_se(T, L, R).

append_d_l(X1 - X2, X2 - Y2, X1 - Y2).
```

RESULTS:


SWI-Prolog (AMD64, Multi-threaded, version 8.0.3)
File Edit Settings Run Debug Help

```
Welcome to SWI-Prolog (threaded, 64 bits, version 8.0.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
Warning: c:/users/abhishek kaushik/desktop/no-comments.pl:116:
        Singleton variables: [New_var]
Warning: c:/users/abhishek kaushik/desktop/no-comments.pl:120:
        Singleton variables: [S,M]
% c:/Users/Abhishek Kaushik/Desktop/no-comments.pl compiled 0.00 sec, 65 clauses
?- unify_mm([X1,sibling(X2, X3, X4), sibling(a, b, c)],T_sys).
T_sys = [[X2]=a, [X3]=b, [X4]=c, [X1]=sibling(X2, X3, X4)].

?- unify_mm([X1,person(X2), person(james)],T_sys).
T_sys = [[X2]=james, [X1]=person(X2)].

?- 
```

EXPERIMENT 7

AIM OF THE EXPERIMENT: To implement best-first search.

THEORY:

Best-first search is a search algorithm which explores a graph by expanding the most promising node chosen according to a specified rule.

Judea Pearl described best-first search as estimating the promise of node n by a "heuristic evaluation function {\displaystyle f(n)} f(n) which, in general, may depend on the description of n, the description of the goal, the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domain."[1][2]

Some authors have used "best-first search" to refer specifically to a search with a heuristic that attempts to predict how close the end of a path is to a solution, so that paths which are judged to be closer to a solution are extended first. This specific type of search is called greedy best-first search[2] or pure heuristic search.[3]

Efficient selection of the current best candidate for extension is typically implemented using a priority queue.

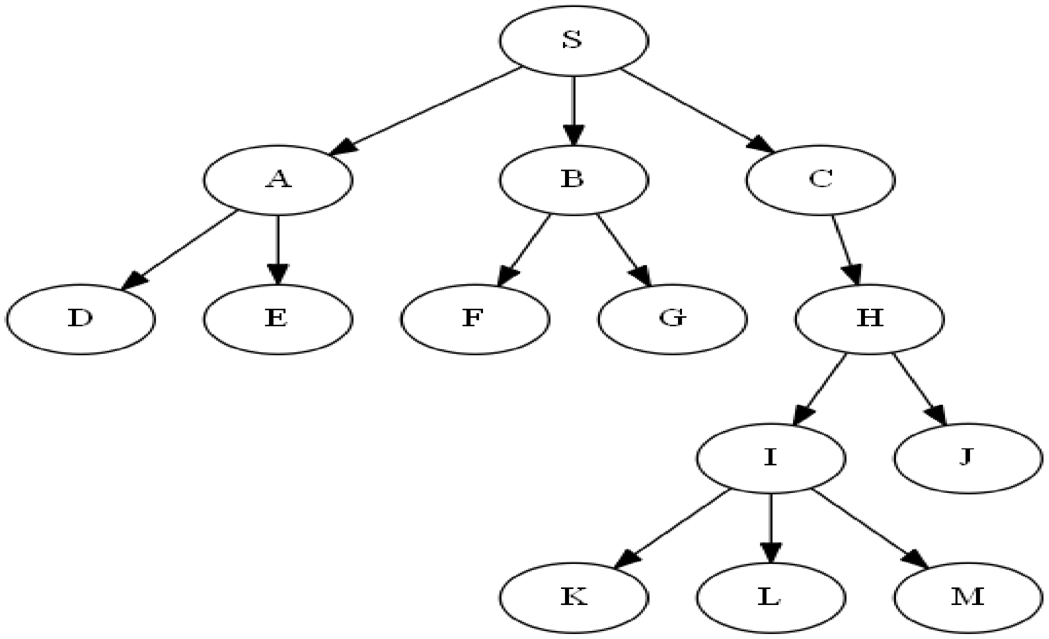
The A* search algorithm is an example of a best-first search algorithm, as is B*. Best-first algorithms are often used for path finding in combinatorial search. Neither A* nor B* is a greedy best-first search, as they incorporate the distance from the start in addition to estimated distances to the goal.

In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node. So both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search.

We use a priority queue to store costs of nodes. So the implementation is a variation of BFS, we just need to change Queue to Priority Queue.

Analysis:

- The worst case time complexity for Best First Search is O(n * Log n) where n is number of nodes. In worst case, we may have to visit all nodes before we reach goal. Note that priority queue is implemented using Min(or Max) Heap, and insert and remove operations take O(log n) time.
- Performance of the algorithm depends on how well the cost or evaluation function is designed.



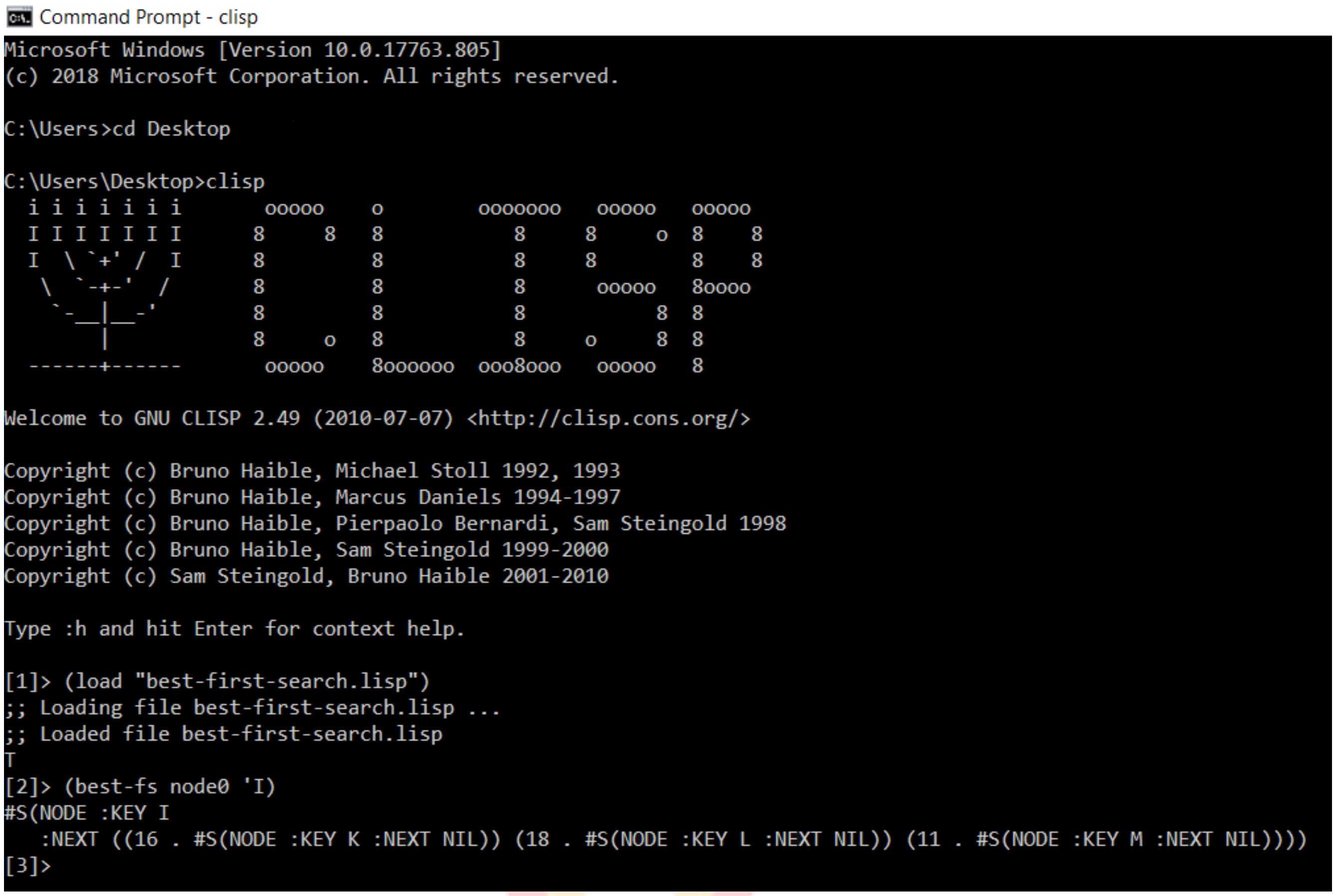
ALGORITHM:

```
Best-First-Search(Grah g, Node start)
    1) Create an empty PriorityQueue
       PriorityQueue pq;
    2) Insert "start" in pq.
       pq.insert(start)
    3) Until PriorityQueue is empty
          u = PriorityQueue.DeleteMin
          If u is the goal
             Exit
          Else
             Foreach neighbor v of u
                If v "Unvisited"
                    Mark v "Visited"
                    pq.insert(v)
             Mark u "Examined"
End procedure
```

CODE:

```
(defstruct node key next)

(defun make-edge-node-pair (edge-wt next-edge)
  (cons edge-wt next-edge))

(defvar node0 (make-node :key 'S :next nil))
(defvar node1 (make-node :key 'A :next nil))
(defvar node2 (make-node :key 'B :next nil))
(defvar node3 (make-node :key 'C :next nil))
(defvar node4 (make-node :key 'D :next nil))
(defvar node5 (make-node :key 'E :next nil))
(defvar node6 (make-node :key 'F :next nil))
(defvar node7 (make-node :key 'G :next nil))
(defvar node8 (make-node :key 'H :next nil))
(defvar node9 (make-node :key 'I :next nil))
(defvar node10 (make-node :key 'J :next nil))
(defvar node11 (make-node :key 'K :next nil))
(defvar node12 (make-node :key 'L :next nil))
(defvar node13 (make-node :key 'M :next nil))

(defun heuristic (n)
      (random n))

(setf (node-next node0) (list (make-edge-node-pair (heuristic 20) node1) (make-
edge-node-pair (heuristic 20) node2) (make-edge-node-pair (heuristic 20) node3)))
(setf (node-next node1) (list (make-edge-node-pair (heuristic 20) node4) (make-
edge-node-pair (heuristic 20) node5)))
(setf (node-next node2) (list (make-edge-node-pair (heuristic 20) node6) (make-
edge-node-pair (heuristic 20) node7)))
(setf (node-next node3) (list (make-edge-node-pair (heuristic 20) node8)))
(setf (node-next node8) (list (make-edge-node-pair (heuristic 20) node9) (make-
edge-node-pair (heuristic 20) node10)))
(setf (node-next node9) (list (make-edge-node-pair (heuristic 20) node11) (make-
edge-node-pair (heuristic 20) node12) (make-edge-node-pair (heuristic 20)
node13)))

(defun find-min (priority-q)
      (let ((edge-wt (make-array 1 :adjustable t :fill-pointer 0)))
       (loop for i in priority-q
          do (vector-push-extend (car i) edge-wt))
       (let ((min-index 0))
        (loop for curr-index from 0 to (- (length edge-wt) 1)
           do (if (< (elt edge-wt curr-index) (elt edge-wt min-index))
               (setf min-index curr-index)))
        min-index)))

(defun best-fs (root goal-symbol)
      (let ((priority-q '()))
       (push (make-edge-node-pair 0 root) priority-q)
       (loop while priority-q
          do (progn (let (index u)
                (setf index (find-min priority-q))
                (setf u (cdr (elt priority-q index)))
                (setf priority-q (delete (elt priority-q index) priority-q))
                (if (eq (node-key u) goal-symbol)
                  (return-from best-fs u)
                  (loop for next-nodes in (node-next u)
                     do (push next-nodes priority-q))))))))
```

RESULTS:


Command Prompt - clisp

```
Microsoft Windows [Version 10.0.17763.805]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users>cd Desktop

C:\Users\Desktop>clisp
  i i i i i i i       ooooo    o        ooooooo   ooooo   ooooo
  I I I I I I I      8     8   8           8     8     o  8    8
  I  \ `+' /  I      8         8           8     8        8    8
   \  `-+-'  /       8         8           8      ooooo   8oooo
    `-__|__-'        8         8           8           8  8
        |            8     o   8           8     o     8  8
  ------+------       ooooo    8oooooo  ooo8ooo   ooooo   8

Welcome to GNU CLISP 2.49 (2010-07-07) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010

Type :h and hit Enter for context help.

[1]> (load "best-first-search.lisp")
;; Loading file best-first-search.lisp ...
;; Loaded file best-first-search.lisp
T
[2]> (best-fs node0 'I)
#S(NODE :KEY I
   :NEXT ((16 . #S(NODE :KEY K :NEXT NIL)) (18 . #S(NODE :KEY L :NEXT NIL)) (11 . #S(NODE :KEY M :NEXT NIL))))
[3]>
```

EXPERIMENT 8

AIM OF THE EXPERIMENT: To implement Knight tour’s problem's

THEORY:

A knight's tour is a sequence of moves of a knight on a chessboard such that the knight visits every square only once. If the knight ends on a square that is one knight's move from the beginning square (so that it could tour the board again immediately, following the same path), the tour is closed; otherwise, it is open.

The knight's tour problem is the mathematical problem of finding a knight's tour. Creating a program to find a knight's tour is a common problem given to computer science students. Variations of the knight's tour problem involve chessboards of different sizes than the usual 8 × 8, as well as irregular (non-rectangular) boards.

On an 8 × 8 board, there are exactly 26,534,728,821,064 directed closed tours (i.e. two tours along the same path that travel in opposite directions are counted separately, as are rotations and reflections). The number of undirected closed tours is half this number, since every tour can be traced in reverse. There are 9,862 undirected closed tours on a 6 × 6 board.

The numbers of all directed tours (open and closed) on an n × n board for n = 1, 2, … are:

1; 0; 0; 0; 1,728; 6,637,920; 165,575,218,320; 19,591,828,170,979,904. (sequence A165134 in the OEIS).

There are several ways to find a knight's tour on a given board with a computer. Some of these methods are algorithms while others are heuristics.

Brute-force algorithms
A brute-force search for a knight's tour is impractical on all but the smallest boards. For example, there are approximately 4×1051 possible move sequences on an 8 × 8 board, and it is well beyond the capacity of modern computers (or networks of computers) to perform operations on such a large set. However, the size of this number is not indicative of the difficulty of the problem, which can be solved "by using human insight and ingenuity ... without much difficulty."

Divide and conquer algorithms
By dividing the board into smaller pieces, constructing tours on each piece, and patching the pieces together, one can construct tours on most rectangular boards in linear time – that is, in a time proportional to the number of squares on the board.

Warnsdorff's rule
Warnsdorff's rule is a heuristic for finding a single knight's tour. The knight is moved so that it always proceeds to the square from which the knight will have the fewest onward moves. When calculating the number of onward moves for each candidate square, we do not count moves that revisit any square already visited. It is possible to have two or more choices for which the number of onward moves is equal; there are various methods for breaking such ties, including one devised by Pohl and another by Squirrel and Cull.

This rule may also more generally be applied to any graph. In graph-theoretic terms, each move is made to the adjacent vertex with the least degree. Although the Hamiltonian path problem is NP-hard in general, on many graphs that occur in practice this heuristic is able to successfully locate a solution in linear time. The knight's tour is such a special case.

The heuristic was first described in "Des Rösselsprungs einfachste und allgemeinste Lösung" by H. C. von Warnsdorff in 1823. A computer program that finds a knight's tour for any starting position using Warnsdorff's rule was written by Gordon Horsington and published in 1984 in the book Century/Acorn User Book of Computer Puzzles

ALGORITHM:

Naive Algorithm for Knight's tour
The Naive Algorithm is to generate all tours one by one and check if the generated tour satisfies the constraints.

```
while there are untried tours
{
   generate the next tour
   if this tour covers all squares
   {
      print this path;
   }
}
```

Backtracking works in an incremental way to attack problems. Typically, we start from an empty solution vector and one by one add items (Meaning of item varies from problem to problem. In context of Knight's tour problem, an item is a Knight's move). When we add an item, we check if adding the current item violates the problem constraint, if it does then we remove the item and try other alternatives. If none of the alternatives work out then we go to previous stage and remove the item added in the previous stage. If we reach the initial stage back then we say that no solution exists. If adding an item doesn't violate constraints then we recursively add items one by one. If the solution vector becomes complete then we print the solution.

Backtracking Algorithm for Knight's tour
Following is the Backtracking algorithm for Knight's tour problem.

```
If all squares are visited
    print the solution
Else
   a) Add one of the next moves to solution vector and recursively
   check if this move leads to a solution. (A Knight can make maximum
   eight moves. We choose one of the 8 moves in this step).
```

b) If the move chosen in the above step doesn't lead to a solution then remove this move from the solution vector and try other alternative moves.
c) If none of the alternatives work then return false (Returning false will remove the previously added item in recursion and if false is returned by the initial call of recursion then "no solution exists")

CODE:

```
// C program for Knight Tour problem
#include<stdio.h>
#define N 8

int solveKTUtil(int x, int y, int movei, int sol[N][N],
                                int xMove[], int yMove[]);

/* A utility function to check if i,j are valid indexes
for N*N chessboard */
int isSafe(int x, int y, int sol[N][N])
{
        return ( x >= 0 && x < N && y >= 0 &&
                        y < N && sol[x][y] == -1);
}

/* A utility function to print solution matrix sol[N][N] */
void printSolution(int sol[N][N])
{
        for (int x = 0; x < N; x++)
        {
                for (int y = 0; y < N; y++)
                        printf(" %2d ", sol[x][y]);
                printf("\n");
        }
}

/* This function solves the Knight Tour problem using
Backtracking. This function mainly uses solveKTUtil()
to solve the problem. It returns false if no complete
tour is possible, otherwise return true and prints the
tour.
Please note that there may be more than one solutions,
this function prints one of the feasible solutions. */
int solveKT()
{
	int sol[N][N];

	/* Initialization of solution matrix */
	for (int x = 0; x < N; x++)
		for (int y = 0; y < N; y++)
			sol[x][y] = -1;

	/* xMove[] and yMove[] define next move of Knight.
	xMove[] is for next value of x coordinate
	yMove[] is for next value of y coordinate */
	int xMove[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };
	int yMove[8] = { 1, 2, 2, 1, -1, -2, -2, -1 };

	// Since the Knight is initially at the first block
	sol[0][0] = 0;

	/* Start from 0,0 and explore all tours using
	solveKTUtil() */
	if (solveKTUtil(0, 0, 1, sol, xMove, yMove) == 0)
	{
		printf("Solution does not exist");
		return 0;
	}
	else
		printSolution(sol);

	return 1;
}

/* A recursive utility function to solve Knight Tour
problem */
int solveKTUtil(int x, int y, int movei, int sol[N][N],
                        int xMove[N], int yMove[N])
{
int k, next_x, next_y;
if (movei == N*N)
	return 1;

/* Try all next moves from the current coordinate x, y */
for (k = 0; k < 8; k++)
{
	next_x = x + xMove[k];
	next_y = y + yMove[k];
	if (isSafe(next_x, next_y, sol))
	{
		sol[next_x][next_y] = movei;
		if (solveKTUtil(next_x, next_y, movei+1, sol,
						xMove, yMove) == 1)
			return 1;
		else
			sol[next_x][next_y] = -1;// backtracking
	}
}

return 0;
}

/* Driver program to test above functions */
int main()
{
	solveKT();
	return 0;
}
```

EXPERIMENT 9

AIM OF THE EXPERIMENT: To implement alpha-beta pruning.

THEORY:

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. It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, Go, etc.). It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Allen Newell and Herbert A. Simon who used what John McCarthy calls an "approximation" in 1958 wrote that alpha–beta "appears to have been reinvented a number of times". Arthur Samuel had an early version for a checkers simulation. Richards, Timothy Hart, Michael Levin and/or Daniel Edwards also invented alpha–beta independently in the United States. McCarthy proposed similar ideas during the Dartmouth workshop in 1956 and suggested it to a group of his students including Alan Kotok at MIT in 1961. Alexander Brudno independently conceived the alpha–beta algorithm, publishing his results in 1963. Donald Knuth and Ronald W. Moore refined the algorithm in 1975. Judea Pearl proved its optimality for trees with randomly assigned leaf values in terms of the expected running time in two papers. The optimality of the randomized version of alpha-beta was shown by Michael Saks and Avi Wigderson in 1986.

The algorithm maintains two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of respectively. Initially, alpha is negative infinity and beta is positive infinity, i.e. both players start with their worst possible score. Whenever the maximum score that the minimizing player (i.e. the "beta" player) is assured of becomes less than the minimum

score that the maximizing player (i.e., the "alpha" player) is assured of (i.e. beta < alpha), the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play.
Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called pruning. This involves two threshold parameter Alpha and beta for future expansion, so it is called alpha-beta pruning. It is also called as Alpha-Beta Algorithm.
Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
The two-parameter can be defined as:
Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞.
Beta: The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.
The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

ALGORITHM:

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cut-off *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if α ≥ β then
                break (* α cut-off *)
        return value
(* Initial call *)
alphabeta(origin, depth, −∞, +∞, TRUE)
```

CODE:

```
#include<iostream>
using namespace std;
int index1;
char board[9] = {'*','*','*','*','*','*','*','*','*'};// Single array represents the
board '*' means empty box in board

int isFull()// Board is full
{
    for(int i =0;i<9;i++)
    {
        if(board[i]!='X')
        {
            if(board[i]!='O')
              {
                return 0;
              }
        }
    }
return 1;
}

int user_won()//Checks whether user has won
{
    for(int i=0;i<9;i+=3)
    {

if((board[i]==board[i+1])&&(board[i+1]==board[i+2])&&(board[i]=='O'))
            return 1;
    }
    for(int i=0;i<3;i++)
    {
```

```
if((board[i]==board[i+3])&&(board[i+3]==board[i+6])&&(board[i]=='O'))
            return 1;
    }
    if((board[0]==board[4])&&(board[4]==board[8])&&(board[0]=='O'))
    {
        return 1;
    }
    if((board[2]==board[4])&&(board[4]==board[6])&&(board[2]=='O'))
    {
        return 1;
    }
    return 0;
}

int cpu_won()// Checks whether CPU has won
{
    for(int i=0;i<9;i+=3)
    {

if((board[i]==board[i+1])&&(board[i+1]==board[i+2])&&(board[i]=='X'))
            return 1;
    }
    for(int i=0;i<3;i++)
    {

if((board[i]==board[i+3])&&(board[i+3]==board[i+6])&&(board[i]=='X'))
            return 1;
    }
    if((board[0]==board[4])&&(board[4]==board[8])&&(board[0]=='X'))
    {
        return 1;
    }
    if((board[2]==board[4])&&(board[4]==board[6])&&(board[2]=='X'))
    {
        return 1;
    }
    return 0;
}

void draw_board() //display tic-tac-toe board
{
    cout<<endl;
    cout<<board[0]<<"|"<<board[1]<<"|"<<board[2]<<endl;
    cout<<"-----"<<endl;
    cout<<board[3]<<"|"<<board[4]<<"|"<<board[5]<<endl;
    cout<<"-----"<<endl;
    cout<<board[6]<<"|"<<board[7]<<"|"<<board[8]<<endl;
}

int minimax(bool flag)// The minimax function
{

    int max_val=-1000,min_val=1000;
    int i,j,value = 1;
    if(cpu_won() == 1)
        {return 10;}
    else if(user_won() == 1)
        {return -10;}
    else if(isFull()== 1)
        {return 0;}
    int score[9] = {1,1,1,1,1,1,1,1,1};//if score[i]=1 then it is empty

        for(i=0;i<9;i++)
            {
                if(board[i] == '*')
                {
                    if(min_val>max_val) // reverse of pruning condition.....
                    {
                        if(flag == true)
            {
             board[i] = 'X';
             value = minimax(false);
            }
             else
             {
              board[i] = 'O';
              value = minimax(true);
             }
           board[i] = '*';
           score[i] = value;
          }
         }
      }

   if(flag == true)
   {
           max_val = -1000;
           for(j=0;j<9;j++)
           {
             if(score[j] > max_val && score[j] != 1)
             {
                 max_val = score[j];
                 index1 = j;
             }
           }
          return max_val;
   }
   if(flag == false)
   {
          min_val = 1000;
          for(j=0;j<9;j++)
          {
             if(score[j] < min_val && score[j] != 1)
             {
                    min_val = score[j];
                    index1 = j;
                }
            }
        return min_val;
    }
}

int main() //The main function
{
  int move,choice;
  cout<<"-------------------------TIC TAC TOE-----------------------------------------------------";
  cout<<endl<<"USER--->(O)      CPU------>(X)";
  cout<<endl<<"SELECT : 1-> Player first 2-> CPU first:";
  cin>>choice;
  if(choice == 1)
  {
      draw_board();
    up:cout<<endl<<"Enter the move:";
      cin>>move;
      if(board[move-1]=='*')
      {
        board[move-1] = 'O';
        draw_board();
      }
      else
      {
         cout<<endl<<"Invalid Move......Try different move";
         goto up;
      }
  }

   while(true)
   {
    cout<<endl<<"CPU MOVE....";
    minimax(true);
    board[index1] = 'X';
    draw_board();
    if(cpu_won()==1)
    {
        cout<<endl<<"CPU WON.....";
        break;
    }
    if(isFull()==1)
    {
        cout<<endl<<"Draw....";
        break;
    }
again:  cout<<endl<<"Enter the move:";
    cin>>move;
    if(board[move-1]=='*')
      {
        board[move-1] = 'O';
        draw_board();
      }
      else
      {
         cout<<endl<<"Invalid Move......Try different move";
         goto again;
      }
      if(user_won()==1)
      {
         cout<<endl<<"You Won......";
         break;
      }
      if(isFull() == 1)
    {
        cout<<endl<<"Draw....";
        break;
    }
}

}
```

EXPERIMENT 10

AIM OF THE EXPERIMENT: Brief reports on semantic web/Swarm Intelligence/Genetic Algorithm/ Artificial Neural Network

THEORY:

1. **Semantic Web**

The ultimate goal of Semantic Web is to make the machine to understand the Internet data. To enable the encoding of semantics with the data, well-known technologies are RDF(Resource Description Framework) and OWL(Web Ontology Language). These technologies formally represent the meaning involved in information. For example, ontology can describe concepts, relationships between things, and categories of things. These embedded semantics with the data offer significant advantages such as reasoning over data and dealing with heterogeneous data sources. The Semantic Web is an extension of the World Wide Web through standards by the World Wide Web Consortium (W3C). The standards promote common data formats and exchange protocols on the Web, most fundamentally the Resource Description Framework (RDF). According to the W3C, "The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries". The Semantic Web is therefore regarded as an integrator across different content, information applications and systems.The term was coined by Tim Berners-Lee for a web of data (or data web) that can be processed by machines—that is, one in which much of the meaning is machine-readable. While its critics have questioned its feasibility, proponents argue that applications in library and information science, industry, biology and human sciences research have already proven the validity of the original concept.Berners-Lee originally expressed his vision of the Semantic Web as follows: I have a dream for the Web [in which computers] become capable of analyzing all the data on the Web – the content, links, and transactions

between people and computers. A "Semantic Web", which makes this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines. The "intelligent agents" people have touted for ages will finally materialize.The 2001 Scientific American article by Berners-Lee, Hendler, and Lassila described an expected evolution of the existing Web to a Semantic Web. In 2006, Berners-Lee and colleagues stated that: "This simple idea…remains largely unrealized".
In 2013, more than four million Web domains contained Semantic Web markup.
The concept of the semantic network model was formed in the early 1960s by researchers such as the cognitive scientist Allan M. Collins, linguist M. Ross Quillian and psychologist Elizabeth F. Loftus as a form to represent semantically structured knowledge. When applied in the context of the modern internet, it extends the network of hyperlinked human-readable web pages by inserting machine-readable metadata about pages and how they are related to each other. This enables automated agents to access the Web more intelligently and perform more tasks on behalf of users. The term "Semantic Web" was coined by Tim Berners-Lee,[7] the inventor of the World Wide Web and director of the World Wide Web Consortium ("W3C"), which oversees the development of proposed Semantic Web standards. He defines the Semantic Web as "a web of data that can be processed directly and indirectly by machines". Many of the technologies proposed by the W3C already existed before they were positioned under the W3C umbrella. These are used in various contexts, particularly those dealing with information that encompasses a limited and defined domain, and where sharing data is a common necessity, such as scientific research or data exchange among businesses. In addition, other technologies with similar goals have emerged, such as microformats.
Many files on a typical computer can also be loosely divided into human-readable documents and machine-readable data. Documents like mail messages, reports, and brochures are read by humans. Data, such as calendars, addressbooks, playlists, and spreadsheets are presented using an application program that lets them be viewed, searched and combined.

Currently, the World Wide Web is based mainly on documents written in Hypertext Markup Language (HTML), a markup convention that is used for coding a body of text interspersed with multimedia objects such as images and interactive forms. Metadata tags provide a method by which computers can categorize the content of web pages. In the examples below, the field names "keywords", "description" and "author" are assigned values such as "computing", and "cheap widgets for sale" and "John Doe".

```
<meta name="keywords" content="computing, computer studies, computer"
/>
<meta name="description" content="Cheap widgets for sale" />
<meta name="author" content="John Doe" />
```

Because of this metadata tagging and categorization, other computer systems that want to access and share this data can easily identify the relevant values. With HTML and a tool to render it (perhaps web browser software, perhaps another user agent), one can create and present a page that lists items for sale. The HTML of this catalog page can make simple, document-level assertions such as "this document's title is 'Widget Superstore'", but there is no capability within the HTML itself to assert unambiguously that, for example, item number X586172 is an Acme Gizmo with a retail price of €199, or that it is a consumer product. Rather, HTML can only say that the span of text "X586172" is something that should be positioned near "Acme Gizmo" and "€199", etc. There is no way to say "this is a catalog" or even to establish that "Acme Gizmo" is a kind of title or that "€199" is a price. There is also no way to express that these pieces of information are bound together in describing a discrete item, distinct from other items perhaps listed on the page.

Semantic HTML refers to the traditional HTML practice of markup following intention, rather than specifying layout details directly. For example, the use of <em> denoting "emphasis" rather than <i>, which specifies italics. Layout details are left up to the browser, in combination with Cascading Style Sheets. But this practice falls short of specifying the semantics of objects such as items for sale or prices.

Microformats extend HTML syntax to create machine-readable semantic markup about objects including people, organisations, events and products. Similar initiatives include RDFa, Microdata and Schema.org

2. Swarm Intelligence

Swarm intelligence (SI) is the collective behavior of decentralized, self-organized systems, natural or artificial. The concept is employed in work on artificial intelligence. The expression was introduced by Gerardo Beni and Jing Wang in 1989, in the context of cellular robotic systems.

SI systems consist typically of a population of simple agents or boids interacting locally with one another and with their environment. The inspiration often comes from nature, especially biological systems. The agents follow very simple rules, and although there is no centralized control structure dictating how individual agents should behave, local, and to a certain degree random, interactions between such agents lead to the emergence of "intelligent" global behavior, unknown to the individual agents. Examples of swarm intelligence in natural systems include ant colonies, bird flocking, hawks hunting, animal herding, bacterial growth, fish schooling and microbial intelligence.

The application of swarm principles to robots is called swarm robotics, while 'swarm intelligence' refers to the more general set of algorithms. 'Swarm prediction' has been used in the context of forecasting problems. Similar approaches to those proposed for swarm robotics are considered for genetically modified organisms in synthetic collective intelligence.

Models of swarm behavior

Boids

Boids is an artificial life program, developed by Craig Reynolds in 1986, which simulates the flocking behaviour of birds. His paper on this topic was published in 1987 in the proceedings of the ACM SIGGRAPH conference.[3]
The name "boid" corresponds to a shortened version of "bird-oid object", which refers to a bird-like object.[4]

As with most artificial life simulations, Boids is an example of emergent behavior; that is, the complexity of Boids arises from the interaction of individual agents (the boids, in this case) adhering to a set of simple rules. The rules applied in the simplest Boids world are as follows:
separation: steer to avoid crowding local flockmates

alignment: steer towards the average heading of local flockmates
cohesion: steer to move toward the average position (center of mass) of local flockmates
More complex rules can be added, such as obstacle avoidance and goal seeking.

Self-propelled particles

Self-propelled particles (SPP), also referred to as the Vicsek model, was introduced in 1995 by Vicsek et al.[5] as a special case of the boids model introduced in 1986 by Reynolds.[6] A swarm is modelled in SPP by a collection of particles that move with a constant speed but respond to a random perturbation by adopting at each time increment the average direction of motion of the other particles in their local neighbourhood.[7] SPP models predict that swarming animals share certain properties at the group level, regardless of the type of animals in the swarm.[8] Swarming systems give rise to emergent behaviours which occur at many different scales, some of which are turning out to be both universal and robust. It has become a challenge in theoretical physics to find minimal statistical models that capture these behaviours.

3. **Genetic Algorithm**
   In computer science and operations research, a genetic algorithm(GA) is a meta heuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection. John Holland introduced genetic algorithms in 1960 based on the concept of Darwin's theory of evolution; afterwards, his student David E. Goldberg extended GA in 1989.
   In a genetic algorithm, a population of candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem is evolved toward better solutions. Each candidate solution has a set of properties (its chromosomes or genotype) which can be mutated and altered; traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible.

The evolution usually starts from a population of randomly generated individuals, and is an iterative process, with the population in each iteration called a generation. In each generation, the fitness of every individual in the population is evaluated; the fitness is usually the value of the objective function in the optimization problem being solved. The more fit individuals are stochastically selected from the current population, and each individual's genome is modified (recombined and possibly randomly mutated) to form a new generation. The new generation of candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

A typical genetic algorithm requires:

a genetic representation of the solution domain,
a fitness function to evaluate the solution domain.
A standard representation of each candidate solution is as an array of bits. Arrays of other types and structures can be used in essentially the same way. The main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size, which facilitates simple

crossover operations. Variable length representations may also be used, but crossover implementation is more complex in this case. Tree-like representations are explored in genetic programming and graph-form representations are explored in evolutionary programming; a mix of both linear chromosomes and trees is explored in gene expression programming. Once the genetic representation and the fitness function are defined, a GA proceeds to initialize a population of solutions and then to improve it through repetitive application of the mutation, crossover, inversion and selection operators.

4. Artificial Neural Network

Artificial neural networks (ANN) or connectionist systems are computing systems that are inspired by, but not identical to, biological neural networks that constitute animal brains. Such systems "learn" to perform tasks by considering examples, generally without being programmed with task-specific rules. For example, in image recognition, they might learn to identify images that contain cats by analyzing example images that have been manually labeled as "cat" or "no cat" and using the results to identify cats in other images. They do this without any prior knowledge of cats, for example, that they have fur, tails, whiskers and cat-like faces. Instead, they automatically generate identifying characteristics from the examples that they process.

An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron that receives a signal then processes it and can signal neurons connected to it.

In ANN implementations, the "signal" at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called edges. Neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold. Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the input layer), to the last layer (the output layer), possibly after traversing the layers multiple times.

The original goal of the ANN approach was to solve problems in the same way that a human brain would. However, over time, attention moved to performing specific tasks, leading to deviations from biology. ANNs have been used on a variety of tasks, including computer vision, speech recognition, machine translation, social network filtering, playing board and video games, medical diagnosis and even in activities that have traditionally been considered as reserved to humans, like painting.

Warren McCulloch and Walter Pitts (1943) opened the subject by creating a computational model for neural networks. In the late 1940s, D. O. Hebb created a learning hypothesis based on the mechanism of neural plasticity that became known as Hebbian learning. Farley and Wesley A. Clark (1954) first used computational machines, then called "calculators", to simulate a Hebbian network. Rosenblatt[6] (1958) created the perceptron.The first functional networks with many layers were published by Ivakhnenko and Lapa in 1965, as the Group Method of Data Handling. The basics of continuous backpropagation were derived in the context of control theory by Kelley in 1960 and by Bryson in 1961, using principles of dynamic programming.
In 1970, Seppo Linnainmaa published the general method for automatic differentiation (AD) of discrete connected networks of nested differentiable functions. In 1973, Dreyfus used backpropagation to adapt parameters of controllers in proportion to error gradients. Werbos's (1975) backpropagation algorithm enabled practical training of multi-layer networks. In 1982, he applied Linnainmaa's AD method to neural networks in the way that became widely used. Thereafter research stagnated following Minsky and Papert (1969), who discovered that basic perceptrons were incapable of processing the exclusive-or circuit and that computers lacked sufficient power to process useful neural networks. In 1992, max-pooling was introduced to help with least shift invariance and tolerance to deformation to aid in 3D object recognition. Schmidhuber adopted a multi-level hierarchy of networks (1992) pre-trained one level at a time by unsupervised learning and fine-tuned by backpropagation.
Geoffrey Hinton et al. (2006) proposed learning a high-level representation using successive layers of binary or real-valued latent variables with a restricted Boltzmann machine to model each layer. In 2012, Ng and Dean created a network that learned to recognize higher-level concepts, such as cats, only from watching unlabeled images. Unsupervised pre-training and increased computing power from GPUs and distributed computing allowed the use of larger networks, particularly in image and visual recognition problems, which became known as "deep learning".[citation needed] Ciresan and colleagues (2010) showed that despite the vanishing gradient problem, GPUs make backpropagation feasible for many-layered

feedforward neural networks.Between 2009 and 2012, ANNs began winning prizes in ANN contests, approaching human level performance on various tasks, initially in pattern recognition and machine learning. For example, the bi-directional and multi-dimensional long short-term memory (LSTM) of Graves et al. won three competitions in connected handwriting recognition in 2009 without any prior knowledge about the three languages to be learned. Ciresan and colleagues built the first pattern recognizers to achieve human-competitive/superhuman performance on benchmarks such as traffic sign recognition (IJCNN 2012).
ANNs have evolved into a broad family of techniques that have advanced the state of the art across multiple domains. The simplest types have one or more static components, including number of units, number of layers, unit weights and topology. Dynamic types allow one or more of these to evolve via learning. The latter are much more complicated, but can shorten learning periods and produce better results. Some types allow/require learning to be "supervised" by the operator, while others operate independently. Some types operate purely in hardware, while others are purely software and run on general purpose computers.
Some of the main breakthroughs include: convolutional neural networks that have proven particularly successful in processing visual and other two-dimensional data; long short-term memory avoid the vanishing gradient problem and can handle signals that have a mix of low and high frequency components aiding large-vocabulary speech recognition, text-to-speech synthesis, and photo-real talking heads; competitive networks such as generative adversarial networks in which multiple networks (of varying structure) compete with each other, on tasks such as winning a game or on deceiving the opponent about the authenticity of an input.
Neural architecture search (NAS) uses machine learning to automate ANN design. Various approaches to NAS have designed networks that compare well with hand-designed systems. The basic search algorithm is to propose a candidate model, evaluate it against a dataset and use the results as feedback to teach the NAS network.[77] Available systems include AutoML and AutoKeras.[78]

Design issues include deciding the number, type and connectedness of network layers, as well as the size of each and the connection type (full, pooling, ...).
Hyperparameters must also be defined as part of the design (they are not learned), governing matters such as how many neurons are in each layer, learning rate, step, stride, depth, receptive field and padding (for CNNs), etc.
Because of their ability to reproduce and model nonlinear processes, Artificial neural networks have found applications in many disciplines. Application areas include system identification and control (vehicle control, trajectory prediction,[80] process control, natural resource management), quantum chemistry,[81] general game playing,[82] pattern recognition (radar systems, face identification, signal classification,[83] 3D reconstruction,[84] object recognition and more), sequence recognition (gesture, speech, handwritten and printed text recognition), medical diagnosis, finance[85] (e.g. automated trading systems), data mining, visualization, machine translation, social network filtering[86] and e-mail spam filtering. ANNs have been used to diagnose cancers, including lung cancer,[87] prostate cancer, colorectal cancer[88] and to distinguish highly invasive cancer cell lines from less invasive lines using only cell shape information.[89][90]
ANNs have been used to accelerate reliability analysis of infrastructures subject to natural disasters[91][92] and to predict foundation settlements.[93]
ANNs have also been used for building black-box models in geoscience: hydrology, ocean modelling and coastal engineering, and geomorphology.
ANNs have been employed in cybersecurity, with the objective to discriminate between legitimate activities and malicious ones. For example, machine learning has been used for classifying Android malware,[99] for identifying domains belonging to threat actors and for detecting URLs posing a security risk. Research is underway on ANN systems designed for penetration testing, for detecting botnets, credit cards frauds and network intrusions.
ANNs have been proposed as a tool to simulate the properties of many-body open quantum systems. In brain research ANNs have studied short-term behavior of individual neurons, the dynamics of neural circuitry arise from interactions between individual neurons and how behavior can arise from abstract neural modules that represent complete subsystems. Studies

considered long-and short-term plasticity of neural systems and their relation to learning and memory from the individual neuron to the system level.

EXPERIMENT 11

AIM OF THE EXPERIMENT: Brief report on latest AI technologies

THEORY:

1. **Natural language generation**

   Natural language generation is an AI sub-discipline that converts data into text, enabling computers to communicate ideas with perfect accuracy.

It is used in customer service to generate reports and market summaries and is offered by companies like Attivio, Automated Insights, Cambridge Semantics, Digital Reasoning, Lucidworks, Narrative Science, SAS, and Yseop.

2. **Speech recognition**

   Siri is just one of the systems that can understand you.

Every day, more and more systems are created that can transcribe human language, reaching hundreds of thousands through voice-response interactive systems and mobile apps.
Companies offering speech recognition services include NICE, Nuance Communications, OpenText and Verint Systems.

3. **Virtual Agents**

   A virtual agent is nothing more than a computer agent or program capable of interacting with humans. The most common example of this kind of technology are chatbots.

Virtual agents are currently being used for customer service and support and as smart home managers.

Some of the companies that provide virtual agents include Amazon, Apple, Artificial Solutions, Assist AI, Creative Virtual, Google, IBM, IPsoft, Microsoft and Satisfi.

4. **Machine Learning Platforms**

   These days, computers can also easily learn, and they can be incredibly intelligent!

Machine learning (ML) is a subdiscipline of computer science and a branch of AI. Its goal is to develop techniques that allow computers to learn.
By providing algorithms, APIs (application programming interface), development and training tools, big data, applications and other machines, ML platforms are gaining more and more traction every day.
They are currently mainly being used for prediction and classification.
Some of the companies selling ML platforms include Amazon, Fractal Analytics, Google, H2O.ai, Microsoft, SAS, Skytree and Adext. The last one is actually the first and only audience management tool in the world that applies real AI and machine learning to digital advertising to find the most profitable audience or demographic group for any ad.

5. **AI-optimised hardware**

   AI technology makes hardware much friendlier.

How? Through new graphics and central processing units and processing devices specifically designed and structured to execute AI-oriented tasks.
And if you haven't seen them already, expect the imminent appearance and wide acceptance of AI-optimized silicon chips that can be inserted right into your portable devices and elsewhere.You can get access to this technology through Alluviate, Cray, Google, IBM, Intel, and Nvidia.

EXPERIMENT 12

AIM OF THE EXPERIMENT: Brief report on Rule ML

THEORY:

RuleML is a global initiative, led by a non-profit organization RuleML Inc., that is devoted to advancing research and industry standards design activities in the technical area of rules that are semantic and highly inter-operable. The standards design takes the form primarily of a markup language, also known as RuleML. The research activities include an annual research conference, the RuleML Symposium, also known as RuleML for short. Founded in fall 2000 by Harold Boley, Benjamin Grosof, and Said Tabet, RuleML was originally devoted purely to standards design, but then quickly branched out into the related activities of coordinating research and organizing an annual research conference starting in 2002. The M in RuleML is sometimes interpreted as standing for Markup and Modeling. The markup language was developed to express both forward (bottom-up) and backward (top-down) rules in XML for deduction, rewriting, and further inferential-transformational tasks. It is defined by the Rule Markup Initiative, an open network of individuals and groups from both industry and academia that was formed to develop a canonical Web language for rules using XML markup and transformations from and to other rule standards/systems.
Markup standards and initiatives related to RuleML include:

Rule Interchange Format (RIF): The design and overall purpose of W3C's Rule Interchange Format (RIF) industry standard is based primarily on the RuleML industry standards design. Like RuleML, RIF embraces a multiplicity of potentially useful rule dialects that nevertheless share common characteristics.
RuleML Technical Committee from Oasis-Open: An industry standards effort devoted to legal automation utilizing RuleML.

Semantic Web Rule Language (SWRL): An industry standards design, based primarily on an early version of RuleML, whose development was funded in part by the DARPA Agent Markup Language (DAML) research program.
Semantic Web Services Framework], particularly its Semantic Web Services Language: An industry standards design, based primarily on a medium-mature version of RuleML, whose development was funded in part by the DARPA Agent Markup Language (DAML) research program and the WSMO research effort of the EU.
Mathematical Markup Language (MathML): However, MathML's Content Markup is better suited for defining functions rather than relations or general rules
Predictive Model Markup Language (PMML): With this XML-based language one can define and share various models for data-mining results, including association rules
Attribute Grammars in XML (AG-markup): For AG's semantic rules, there are various possible XML markups that are similar to Horn-rule markup
Extensible Stylesheet Language Transformations (XSLT): This is a restricted term-rewriting system of rules, written in XML, for transforming XML documents into other text documents