**PRACTICAL 4**

**Objective:- Write a program for Breadth first Search (BFS)**

**Theory :**

[Breadth First Traversal (or Search)](http://en.wikipedia.org/wiki/Breadth-first_search) for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

BFS uses always **queue.**

There are many ways to traverse graphs. BFS is the most commonly used approach.

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

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

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

**Time complexity : O(V + E)**, where V is the number of nodes and E is the number of edges.

## BFS algorithm:

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

1. Visited
2. Not Visited

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

The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

## Pseudocode :

create a queue Q

mark v as visited and put v into Q

whilw Q is non-empty

remove the head u of Q

mark and enqueue all (unvisited) neighbours of u

**Source Code :**

#include<stdio.h>

// #include<conio.h>

int adj[30][30],queue[30],visited[30],n,i,j,f=0,r=-1;

// adjacency matrix 'adj' defined

// a visited array stores whether the vertex has been visited or not

// queue array stores the corresponding sequence of verices to be taken into account

// n is no of vertices

//a recursive bfs function is defined

void bfs(int v) {

for (i=1;i<=n;i++) //for all given n vertices

if(adj[v][i] && !visited[i]) //if they have an adjacent edge with source vertex and have not been visited

queue[++r]=i; //added into queue for further processing

if(f<=r) { //until we reach end of the queue

visited[queue[f]]=1; //mark each vertex selected from queue as visited

bfs(queue[f++]); //run bfs for that vertex

}

}

int main() {

int v; //temporary variable for getting user choice

// clrscr();

//number of vertices specified

printf("\n Enter the number of vertices:");

scanf("%d",&n);

// each vertex is initialized by setting visted and queue to 0

for (i=1;i<=n;i++) {

queue[i]=0;

visited[i]=0;

}

// graph is enterd in nxn format

printf("\n Enter graph data in matrix form:\n");

for (i=1;i<=n;i++)

for (j=1;j<=n;j++)

scanf("%d",&adj[i][j]);

//starting vertex v is specified

printf("\n Enter the starting vertex:");

scanf("%d",&v);

//bfs is called for the starting vertex

bfs(v);

//print reachable nodes after bfs is done

printf("\n The node which are reachable are:\n");

for (i=1;i<=n;i++)

if(visited[i]) //if visited is true, that means the vertex was reachable

printf("%d\t",i); else

printf("\n Bfs is not possible"); //if if false, graph is disconnected and searching fails

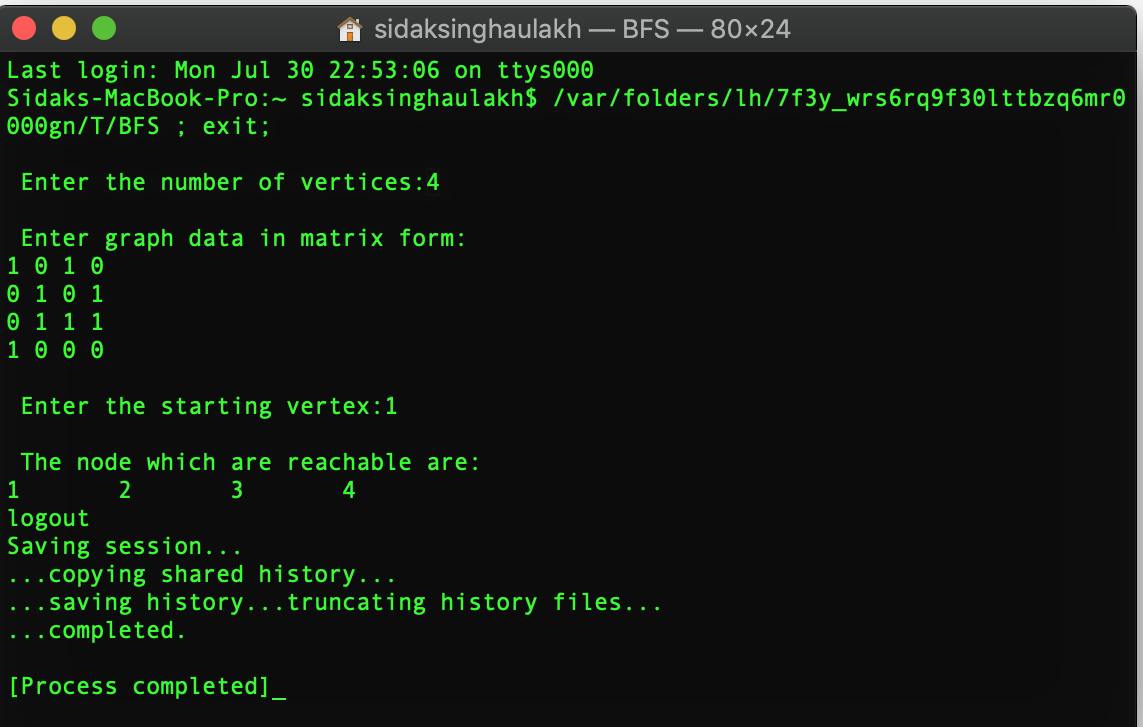
// getch();

printf("\n");

return 0;

}

**OUTPUT**

****

**PRACTICAL 5**

**Objective:- Write a program for Depth first Search (DFS)**

**Theory :**

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

**Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

**Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.

**Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

**Time complexity :** O(V+E), when implemented using an adjacency list.

## Algorithm :

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

1. Visited
2. Not Visited

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

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

## Pseudocode :

The pseudocode for DFS is shown below. In the init() function, notice that we run the DFS function on every node. This is because the graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the DFS algorithm on every node.

DFS(G,v) ( v is the vertex where the search starts )

Stack S := {}; ( start with an empty stack )

for each vertex u, set visited[u] := false;

push S, v;

while (S is not empty) do

u := pop S;

if (not visited[u]) then

visited[u] := true;

for each unvisited neighbour w of u

push S, w;

end if

end while

END DFS()

**Source Code :**

#include<stdio.h>

void DFS(int);

int G[30][30],visited[30],n; //n is no of vertices and graph is sorted in array G[10][10]

int main()

{

int start;

int i,j;

//read n, the number of vertices

printf("Enter number of vertices:");

scanf("%d",&n);

//read the adjecency matrix, the matrix will be of size nxn, so will be filled accordingly

printf("\nEnter graph data in matrix form: \n");

for(i=0;i<n;i++)

for(j=0;j<n;j++)

scanf("%d",&G[i][j]);

//visited array of all is is initialized to zero

for(i=0;i<n;i++)

visited[i]=0;

printf("Enter starting index\n");

scanf("\n%d", &start);

DFS(start); //DFS is started from start vertex specified by user

return 0;

}

void DFS(int i)

{

int j;

printf("\n%d",i); //prints source vertex

visited[i]=1; //marks it as visited

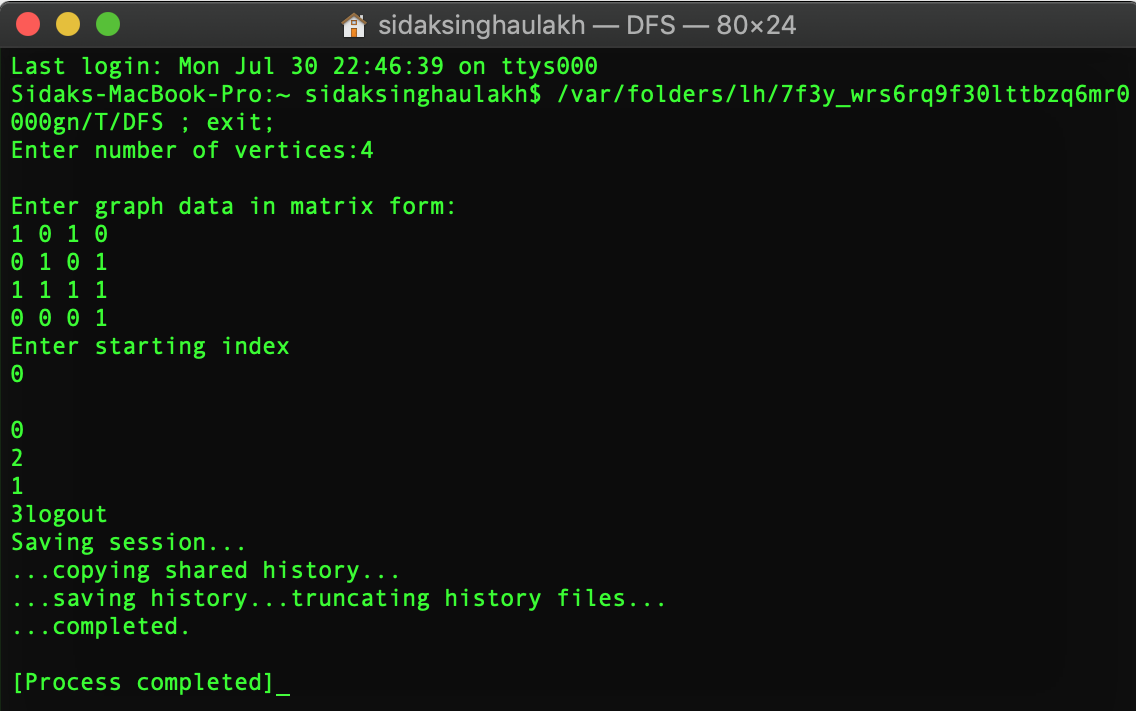
for(j=0;j<n;j++)

if(!visited[j]&&G[i][j]==1) //checks adjacent vertex if they haven't been visited and if there exists edge

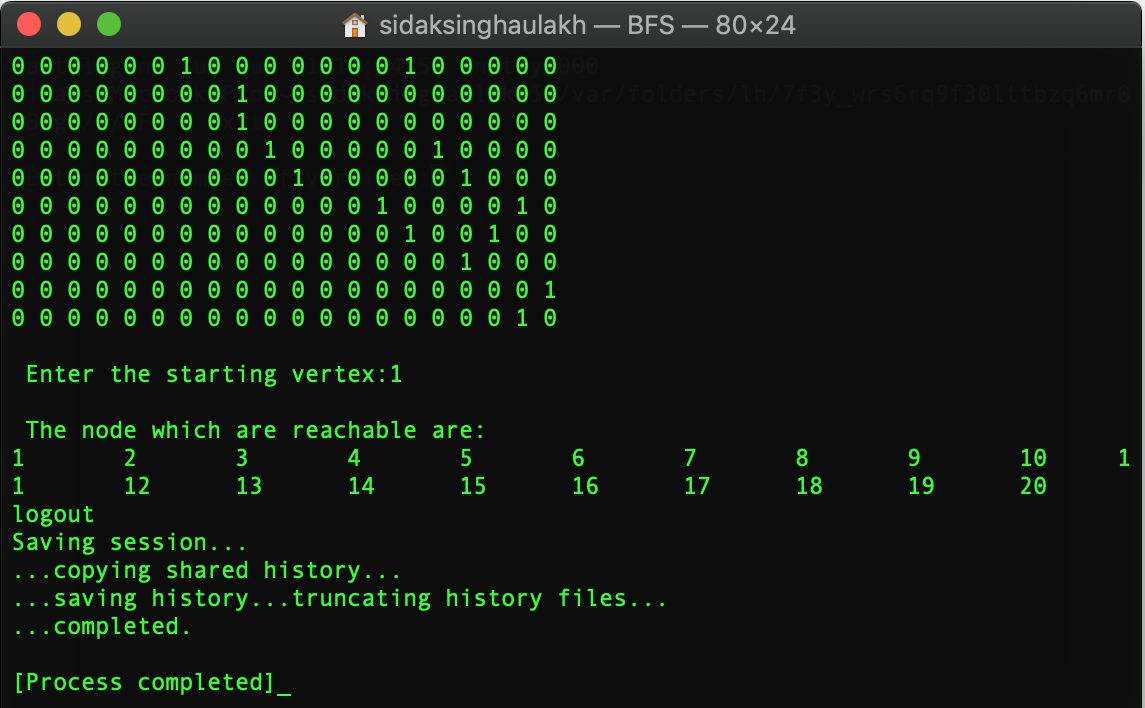
DFS(j); //recursively calls dfs for the adjacent edge.

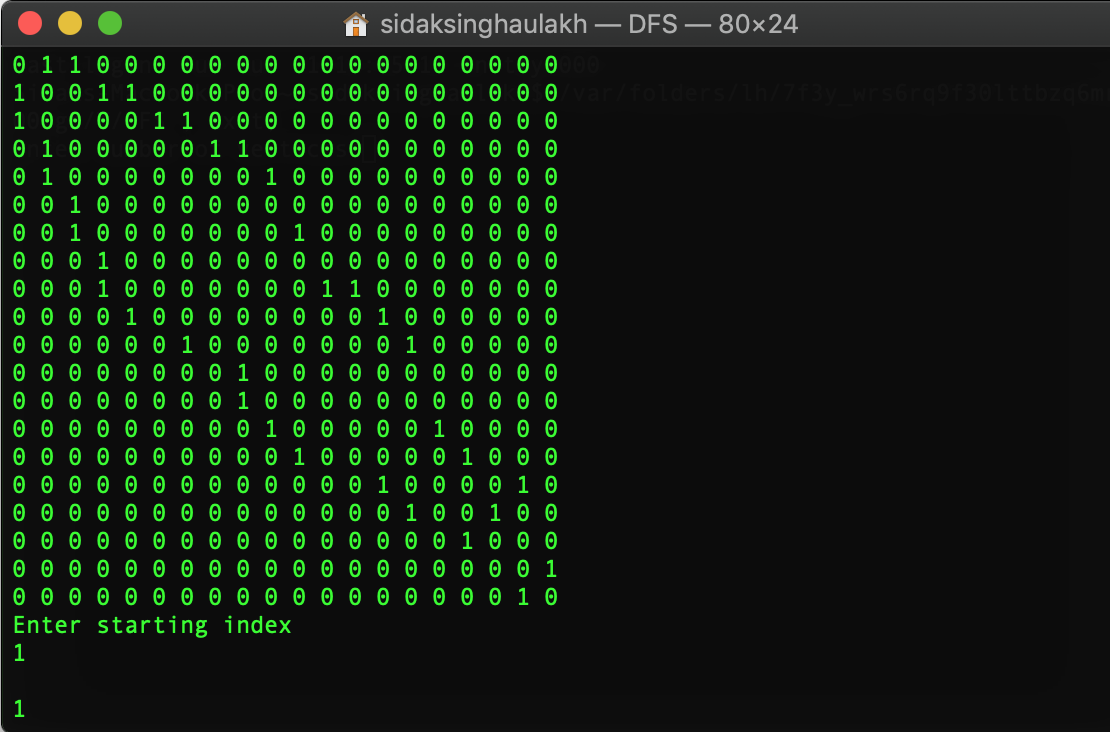
}

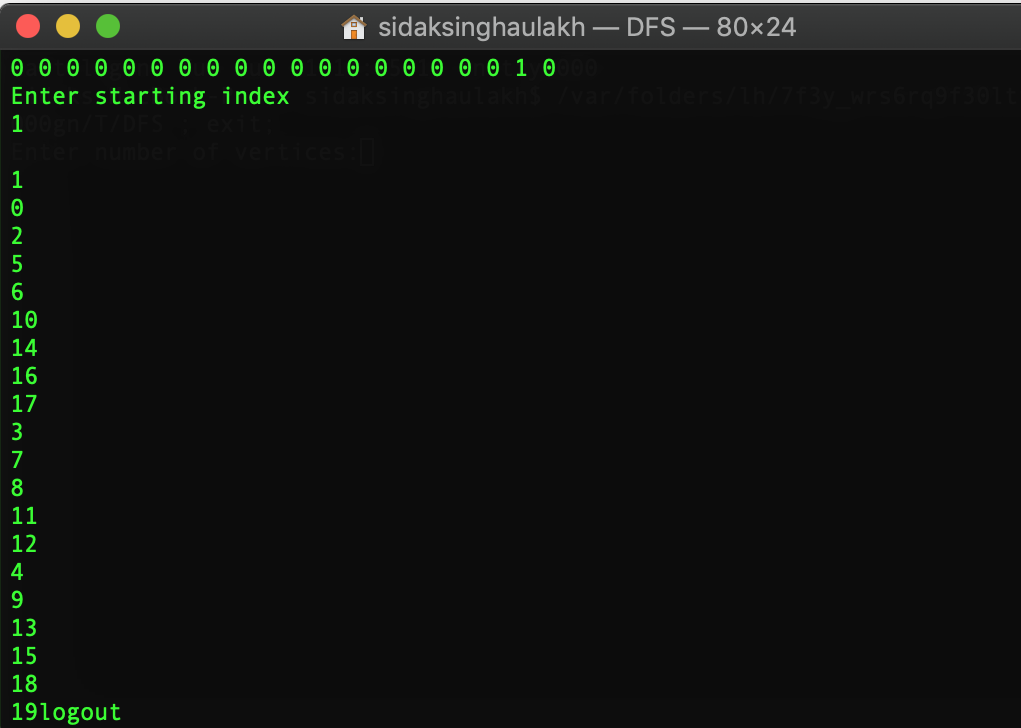
**OUTPUT**

****

**EXAMPLE OF DFS:-**

****

****

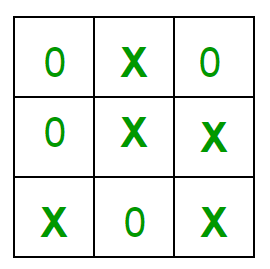
****

**PRACTICAL-6**

**Objective:- Implementation of Tic-Tac-Toe game (COMPUTER VS HUMAN)**

**Rules of the Game**

* The game is to be played between two people (in this program between HUMAN and COMPUTER).
* One of the player chooses ‘O’ and the other ‘X’ to mark their respective cells.
* The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character (‘O’ or ‘X’).
* If no one wins, then the game is said to be draw.



**Implementation**

In our program the moves taken by the computer and the human are chosen randomly. We use rand() function for this.

**What more can be done in the program?**

The program is in not played optimally by both sides because the moves are chosen randomly. The program

can be easily modified so that both players play optimally (which will fall under the category of Artificial Intelligence). Also the program can be modified such that the user himself gives the input (using scanf() or cin).  
The above changes are left as an exercise to the readers.

**Winning Strategy – An Interesting Fact**

If both the players play optimally then it is destined that you will never lose (“although the match can still be drawn”). It doesn’t matter whether you play first or second.In another ways –

“ Two expert players will always draw ”. Isn’t this interesting ?

**Source Code :**

// A C++ Program to play tic-tac-toe

#include<iostream>

// #include<bits.h>

using namespace std;

#define COMPUTER 1

#define HUMAN 2

#define SIDE 3 // Length of the board

// Computer will move with 'O'

// and human with 'X'

#define COMPUTERMOVE 'O'

#define HUMANMOVE 'X'

// A function to show the current board status

void showBoard(char board[][SIDE])

{

printf("\n\n");

printf("\t\t\t %c | %c | %c \n", board[0][0],

board[0][1], board[0][2]);

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

printf("\t\t\t %c | %c | %c \n", board[1][0],

board[1][1], board[1][2]);

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

printf("\t\t\t %c | %c | %c \n\n", board[2][0],

board[2][1], board[2][2]);

return;

}

// A function to show the instructions

void showInstructions()

{

printf("\t\t\t Tic-Tac-Toe\n\n");

printf("Choose a cell numbered from 1 to 9 as below"

" and play\n\n");

printf("\t\t\t 1 | 2 | 3 \n");

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

printf("\t\t\t 4 | 5 | 6 \n");

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

printf("\t\t\t 7 | 8 | 9 \n\n");

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

return;

}

// A function to initialise the game

void initialise(char board[][SIDE], int moves[])

{

// Initiate the random number generator so that

// the same configuration doesn't arises

srand(time(NULL));

// Initially the board is empty

for (int i=0; i<SIDE; i++)

{

for (int j=0; j<SIDE; j++)

board[i][j] = ' ';

}

// Fill the moves with numbers

for (int i=0; i<SIDE\*SIDE; i++)

moves[i] = i;

// randomise the moves

random\_shuffle(moves, moves + SIDE\*SIDE);

return;

}

// A function to declare the winner of the game

void declareWinner(int whoseTurn)

{

if (whoseTurn == COMPUTER)

printf("COMPUTER has won\n");

else

printf("HUMAN has won\n");

return;

}

// A function that returns true if any of the row

// is crossed with the same player's move

bool rowCrossed(char board[][SIDE])

{

for (int i=0; i<SIDE; i++)

{

if (board[i][0] == board[i][1] &&

board[i][1] == board[i][2] &&

board[i][0] != ' ')

return (true);

}

return(false);

}

// A function that returns true if any of the column

// is crossed with the same player's move

bool columnCrossed(char board[][SIDE])

{

for (int i=0; i<SIDE; i++)

{

if (board[0][i] == board[1][i] &&

board[1][i] == board[2][i] &&

board[0][i] != ' ')

return (true);

}

return(false);

}

// A function that returns true if any of the diagonal

// is crossed with the same player's move

bool diagonalCrossed(char board[][SIDE])

{

if (board[0][0] == board[1][1] &&

board[1][1] == board[2][2] &&

board[0][0] != ' ')

return(true);

if (board[0][2] == board[1][1] &&

board[1][1] == board[2][0] &&

board[0][2] != ' ')

return(true);

return(false);

}

// A function that returns true if the game is over

// else it returns a false

bool gameOver(char board[][SIDE])

{

return(rowCrossed(board) || columnCrossed(board)

|| diagonalCrossed(board) );

}

// A function to play Tic-Tac-Toe

void playTicTacToe(int whoseTurn)

{

// A 3\*3 Tic-Tac-Toe board for playing

char board[SIDE][SIDE];

int moves[SIDE\*SIDE];

// Initialise the game

initialise(board, moves);

// Show the instructions before playing

showInstructions();

int moveIndex = 0, x, y;

// Keep playing till the game is over or it is a draw

while (gameOver(board) == false &&

moveIndex != SIDE\*SIDE)

{

if (whoseTurn == COMPUTER)

{

x = moves[moveIndex] / SIDE;

y = moves[moveIndex] % SIDE;

board[x][y] = COMPUTERMOVE;

printf("COMPUTER has put a %c in cell %d\n",

COMPUTERMOVE, moves[moveIndex]+1);

showBoard(board);

moveIndex ++;

whoseTurn = HUMAN;

}

else if (whoseTurn == HUMAN)

{

x = moves[moveIndex] / SIDE;

y = moves[moveIndex] % SIDE;

board[x][y] = HUMANMOVE;

printf ("HUMAN has put a %c in cell %d\n",

HUMANMOVE, moves[moveIndex]+1);

showBoard(board);

moveIndex ++;

whoseTurn = COMPUTER;

}

}

// If the game has drawn

if (gameOver(board) == false &&

moveIndex == SIDE \* SIDE)

printf("It's a draw\n");

else

{

// Toggling the user to declare the actual

// winner

if (whoseTurn == COMPUTER)

whoseTurn = HUMAN;

else if (whoseTurn == HUMAN)

whoseTurn = COMPUTER;

// Declare the winner

declareWinner(whoseTurn);

}

return;

}

// Driver program

int main()

{

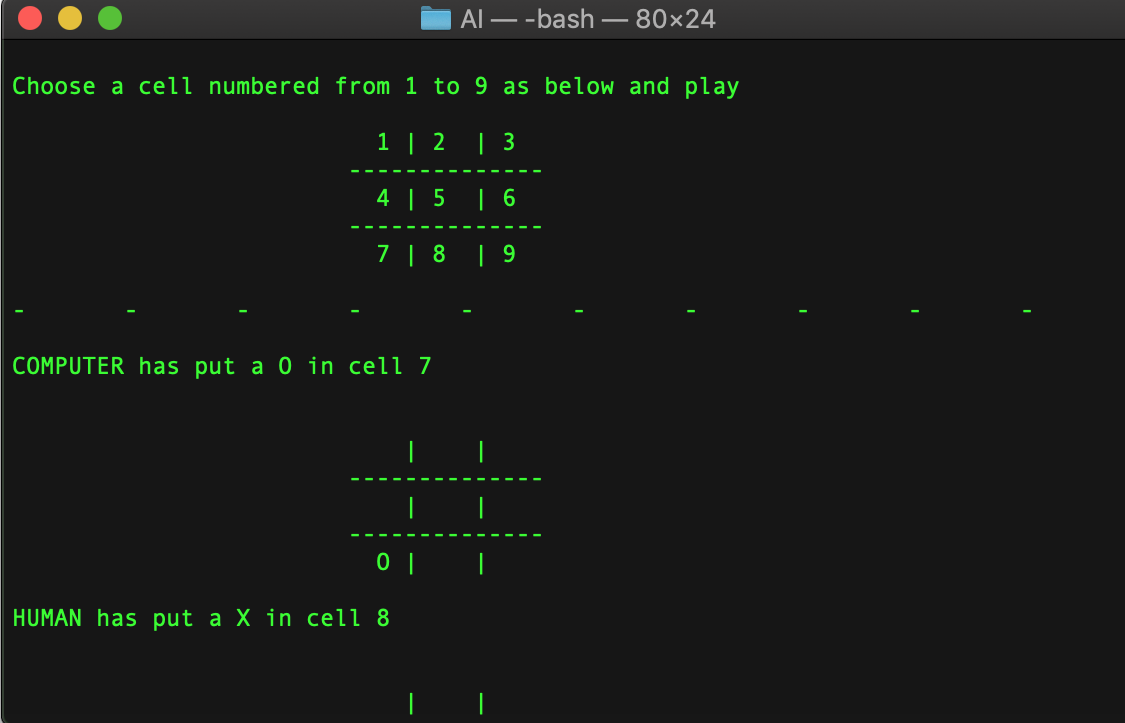
// Let us play the game with COMPUTER starting first

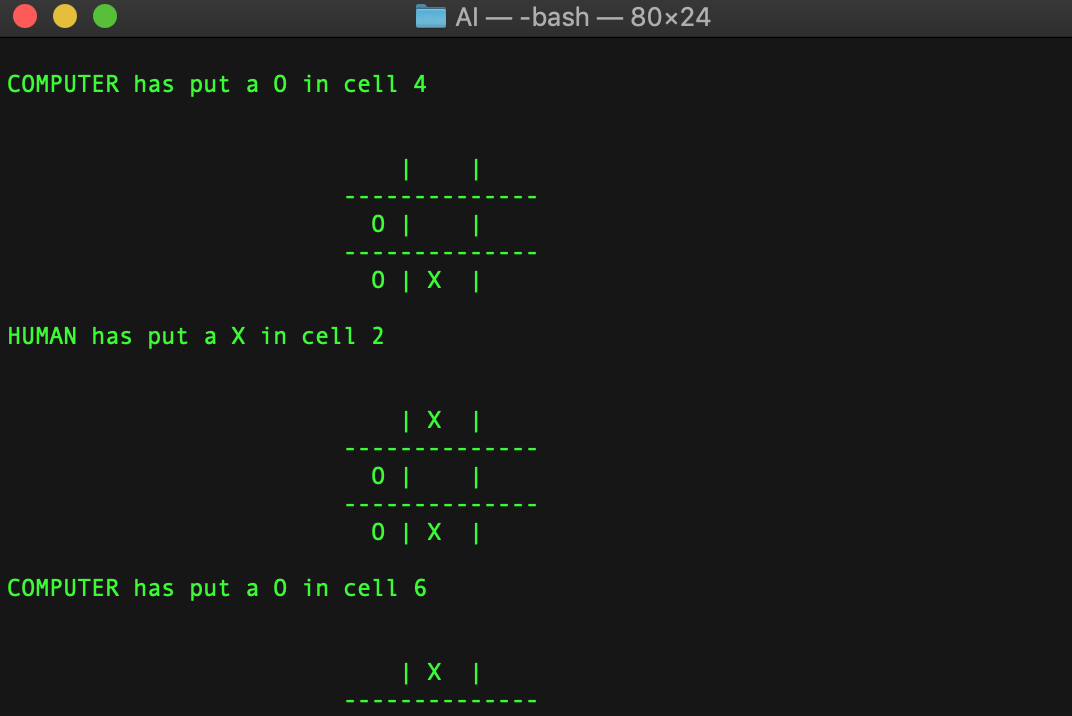
playTicTacToe(COMPUTER);

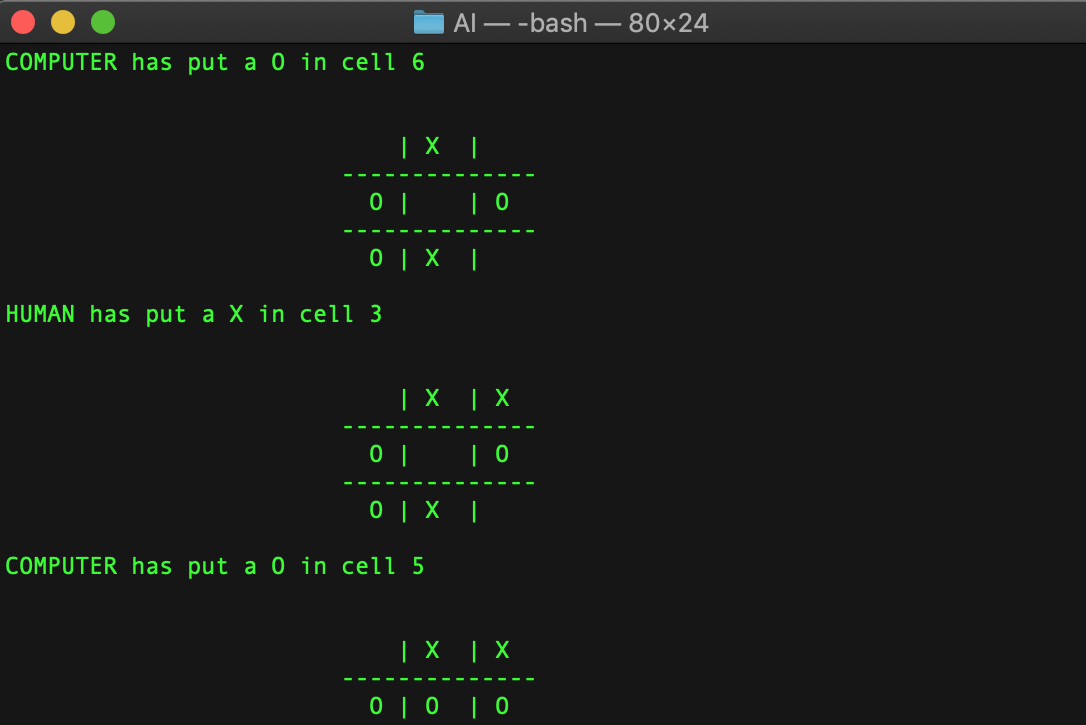
return (0);

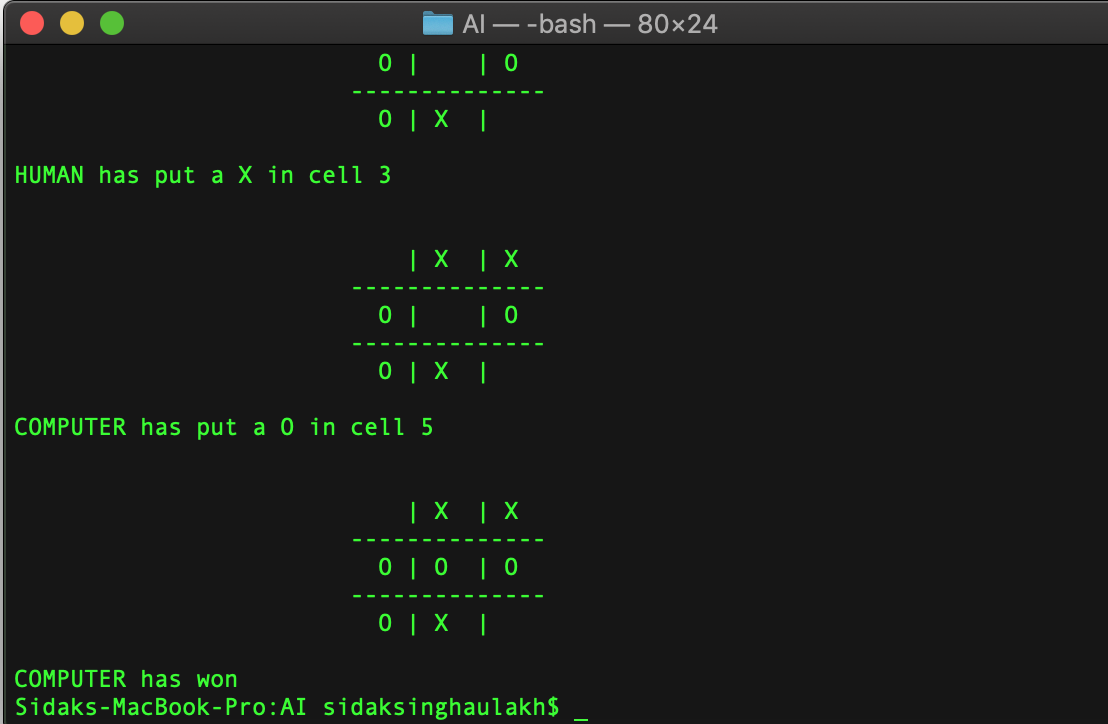
}

**OUTPUT**









**PRACTICAL-7**

**Objective:- Implementation of Tic-Tac-Toe game (COMPUTER VS HUMAN)**

**UNBEATABLE TIC-TAC-TOE**

**Source Code :**

#include <iostream>

using namespace std;

// function to draw the board after every move.

void draw\_board(int \*b) {

int k = 0;

cout << "----++---++----" << endl;

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

cout << "| ";

if (b[k] == 0) { // cross-checks the number entered by user or computer and then plots it on the board

cout << k + 1 << " |";

} else {

if (b[k] == -1) {

cout << "X |";

} else {

cout << "O |";

}

}

k++;

}

cout << "\n----++---++----" << endl;

}

}

int win(const int \*board) {

// determines if a player has won, returns 0 otherwise.

unsigned wins[8][3] = {{0, 1, 2}, {3, 4, 5}, {6, 7, 8}, {0, 3, 6}, //win combination array

{1, 4, 7}, {2, 5, 8}, {0, 4, 8}, {2, 4, 6}};

int i;

for (i = 0; i < 8; ++i) {

if (board[wins[i][0]] != 0 && board[wins[i][0]] == board[wins[i][1]] && //checks if combination occurs in board matrix

board[wins[i][1]] == board[wins[i][2]])

return board[wins[i][2]];

}

return 0;

}

int minimax(int \*board, int player) {

int winner = win(board);

if (winner != 0)

return winner \* player;

int move = -1;

int score = -2;

for (int i = 0; i < 9; i++) {

if (board[i] == 0) {

board[i] = player;

int thisScore = -minimax(board, player \* -1);

if (thisScore > score) {

score = thisScore;

move = i;

}

board[i] = 0; // Reset board after try

}

}

if (move == -1)

return 0;

return score;

}

int computerMove(int \*board) {

int move = -1;

int score = -2;

for (int i = 0; i < 9; ++i) {

if (board[i] == 0) {

board[i] = 1;

int tempScore = -minimax(board, -1);

board[i] = 0;

if (tempScore > score) {

score = tempScore;

move = i;

}

}

}

// returns a score based on minimax tree at a given node.

return move;

}

int main() {

cout << "\n~~~~~~~~~~~~~Tic Tac Toe~~~~~~~~~~~~~\n";

cout << "\n\n BOARD:\n";

cout << "----++---++----" << endl;

cout << "| 1 || 2 || 3 |" << endl;

cout << "----++---++----" << endl;

cout << "| 4 || 5 || 6 |" << endl;

cout << "----++---++----" << endl;

cout << "| 7 || 8 || 9 |" << endl;

cout << "----++---++----" << endl << endl << endl;

cout << "Only legal moves are the numbers you see on the board\n\n";

int board[9] = {0};

int moves = 0, k;

// Player = -1 ; Computer = 1

while (moves < 9) {

int mv;

cout << "Enter Player 1's Move\n";

cin >> mv;

if (board[mv - 1] == 0) {

board[mv - 1] = -1;

moves++;

cout << "\n\nBoard after your move:\n";

draw\_board(board);

if (win(board) == 0) {

k = computerMove(board);

board[k] = 1;

cout << "\n\nBoard after computer's move:\n";

draw\_board(board);

moves++;

if (win(board) != 0) {

break;

}

} else

break;

} else {

cout << "Illegal Move, Try again !! \n\n";

}

}

switch (win(board)) {

case 0:

cout << "Its a draw. Better Smart next time.\n";

break;

case 1:

cout << "You lose.\n";

break;

case -1:

cout << "This will never Happen. But if it does(It never will), "

"Congratulations, You have beaten the unbeatable\n";

break;

}

return 0;

}

**OUTPUT**

