INDEX

List of Experiments

S.NO.	Experiments	Page no.	Submission Date	Remarks

Experiment-1

Aim : WAP to implement Water jug problem.

Program:

```
// C++ program to count minimum number of steps
// required to measure d litres water using jugs
// of m litres and n litres capacity.

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

// Utility function to return GCD of 'a'
// and 'b'.
int gcd(int a, int b)
{
    if (b==0)
       return a;
    return gcd(b, a%b);
}

/* fromCap -- Capacity of jug from which
              water is poured
   toCap   -- Capacity of jug to which
              water is poured
   d       -- Amount to be measured */
int pour(int fromCap, int toCap, int d)
{
    // Initialize current amount of water
    // in source and destination jugs
    int from = fromCap;
    int to = 0;

    // Initialize count of steps required
    int step = 1; // Needed to fill "from" Jug

    // Break the loop when either of the two
    // jugs has d litre water
    while (from != d && to != d)
    {
        // Find the maximum amount that can be
        // poured
        int temp = min(from, toCap - to);
```

```
        // Pour "temp" litres from "from" to "to"
        to   += temp;
        from -= temp;

        // Increment count of steps
        step++;

        if (from == d || to == d)
            break;

        // If first jug becomes empty, fill it
        if (from == 0)
        {
            from = fromCap;
            step++;
        }

        // If second jug becomes full, empty it
        if (to == toCap)
        {
            to = 0;
            step++;
        }
    }
    return step;
}

// Returns count of minimum steps needed to
// measure d litre
int minSteps(int m, int n, int d)
{
    // To make sure that m is smaller than n
    if (m > n)
        swap(m, n);

    // For d > n we cant measure the water
    // using the jugs
    if (d > n)
        return -1;

    // If gcd of n and m does not divide d
    // then solution is not possible
    if ((d % gcd(n,m)) != 0)
        return -1;
```

```
    // Return minimum two cases:
    // a) Water of n litre jug is poured into
    //    m litre jug
    // b) Vice versa of "a"
    return min(pour(n,m,d),   // n to m
               pour(m,n,d));  // m to n
}

// Driver code to test above
int main()
{
    int n = 3, m = 5, d = 4;

    printf("Minimum number of steps required is %d",
           minSteps(m, n, d));

    return 0;
}
```

OUTPUT :

```
Minimum number of steps required is 6
```

Experiment-2

Aim : WAP to implement Hill climbing algorithm.

Program:

```
#include<iostream>
#include<cstdio>
using namespace std;
int calcCost(int arr[],int N){
int c=0;
for(int i=0;i<N;i++){
for(int j=i+1;j<N;j++) if(arr[j]<arr[i]) c++;
}
return c;
}

void swap(int arr[],int i,int j){
int tmp=arr[i];
arr[i]=arr[j];
arr[j]=tmp;
}

int main(){
int N;
scanf("%d",&N);
int arr[N];
for(int i=0;i<N;i++) scanf("%d",&arr[i]);
int bestCost=calcCost(arr,N),newCost,swaps=0;;
while(bestCost>0){
for(int i=0;i<N-1;i++){
swap(arr,i,i+1);
newCost=calcCost(arr,N);
if(bestCost>newCost){
printf("After swap %d: \n",++swaps);
for(int i=0;i<N;i++) printf("%d ",arr[i]);
printf("\n");
bestCost=newCost;
}
else swap(arr,i,i+1);
}
}
printf("Final Ans\n");
for(int i=0;i<N;i++) printf("%d ",arr[i]);
printf("\n");
return 0;}
```

Experiment-3

Aim : WAP to play the game of Tic-Tac-Toe.

Program:

```
// A C++ Program to play tic-tac-toe
#include<bits/stdc++.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-\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",
```

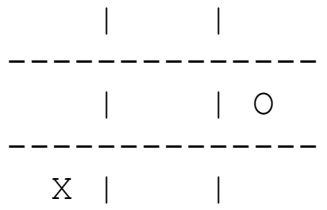
```
                        COMPUTERMOVE, moves[moveIndex]+1);
                showBoard(board);
                moveIndex ++;
                whoseTurn = HUMAN;
        }
        else if (whoseTurn == HUMAN) {
                x = moves[moveIndex] / SIDE;
                y = moves[moveIndex] % SIDE;
                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);
}
```

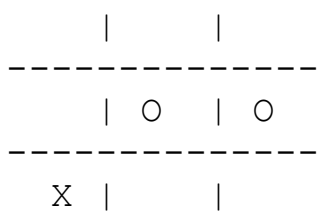
OUTPUT:

```
      Tic-Tac-Toe
Choose a cell numbered from 1 to 9 as below and play

               1 | 2  | 3
              --------------
               4 | 5  | 6
              --------------
               7 | 8  | 9

-    -    -    -    -    -    -    -    -    -

COMPUTER has put a O in cell 6


                   |    |
              --------------
                   |    | O
              --------------
                   |    |
```

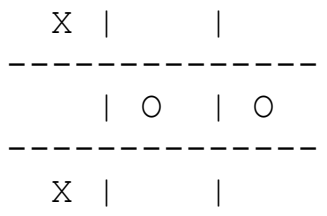
HUMAN has put a X in cell 7



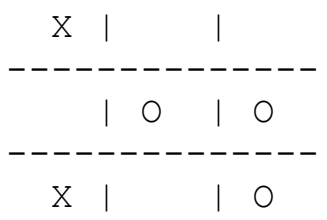
COMPUTER has put a O in cell 5



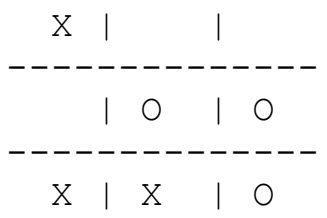
HUMAN has put a X in cell 1



COMPUTER has put a O in cell 9



HUMAN has put a X in cell 8



```
COMPUTER has put a O in cell 4

  X |    |
 ------------
  O | O  | O
 ------------
  X | X  | O

COMPUTER has won
```

Experiment-4

Aim : WAP to implement A* algorithm.

Program:

```
#include <list>
#include <algorithm>
#include <iostream>

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

class map {
public:
    map() {
        char t[8][8] = {
            {0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0,
0},
            {0, 0, 0, 0, 1, 1, 1, 0}, {0, 0, 1, 0, 0, 0, 1,
0},
            {0, 0, 1, 0, 0, 0, 1, 0}, {0, 0, 1, 1, 1, 1, 1,
0},
            {0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0,
0}
        };
        w = h = 8;
        for( int r = 0; r < h; r++ )
            for( int s = 0; s < w; s++ )
                m[s][r] = t[r][s];
    }
    int operator() ( int x, int y ) { return m[x][y]; }
    char m[8][8];
    int w, h;
};

class node {
public:
    bool operator == (const node& o ) { return pos ==
```

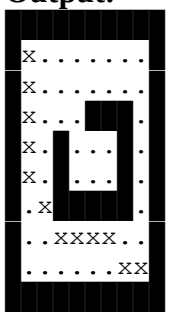
```
o.pos; }
    bool operator == (const point& o ) { return pos == o; }
    bool operator < (const node& o ) { return dist + cost <
o.dist + o.cost; }
    point pos, parent;
    int dist, cost;
};

class aStar {
public:
    aStar() {
        neighbours[0] = point( -1, -1 ); neighbours[1] =
point(  1, -1 );
        neighbours[2] = point( -1,  1 ); neighbours[3] =
point(  1,  1 );
        neighbours[4] = point(  0, -1 ); neighbours[5] =
point( -1,  0 );
        neighbours[6] = point(  0,  1 ); neighbours[7] =
point(  1,  0 );
    }

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

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

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

```
        }
        return false;
    }

    bool fillOpen( node& n ) {
        int stepCost, nc, dist;
        point neighbour;

        for( int x = 0; x < 8; x++ ) {
            // one can make diagonals have different cost
            stepCost = x < 4 ? 1 : 1;
            neighbour = n.pos + neighbours[x];
            if( neighbour == end ) return true;

            if( isValid( neighbour ) && m( neighbour.x,
neighbour.y ) != 1 ) {
                nc = stepCost + n.cost;
                dist = calcDist( neighbour );
                if( !existPoint( neighbour, nc + dist ) ) {
                    node m;
                    m.cost = nc; m.dist = dist;
                    m.pos = neighbour;
                    m.parent = n.pos;
                    open.push_back( m );
                }
            }
        }
        return false;
    }

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

```
    int path( std::list<point>& path ) {
        path.push_front( end );
        int cost = 1 + closed.back().cost;
        path.push_front( closed.back().pos );
        point parent = closed.back().parent;

        for( std::list<node>::reverse_iterator i =
closed.rbegin(); i != closed.rend(); i++ ) {
            if( ( *i ).pos == parent && !( ( *i ).pos ==
start ) ) {
                path.push_front( ( *i ).pos );
                parent = ( *i ).parent;
            }
        }
        path.push_front( start );
        return cost;
    }

    map m; point end, start;
    point neighbours[8];
    std::list<node> open;
    std::list<node> closed;
};

int main( int argc, char* argv[] ) {
    map m;
    point s, e( 7, 7 );
    aStar as;

    if( as.search( s, e, m ) ) {
        std::list<point> path;
        int c = as.path( path );
        for( int y = -1; y < 9; y++ ) {
            for( int x = -1; x < 9; x++ ) {
                if( x < 0 || y < 0 || x > 7 || y > 7 || m(
x, y ) == 1 )
                    std::cout << char(0xdb);
                else {
                    if( std::find( path.begin(),
path.end(), point( x, y ) )!= path.end() )
                        std::cout << "x";
                    else std::cout << ".";
                }
            }
            std::cout << "\n";
```

```
        }

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

Output:




```
Path cost 11: (0, 0) (0, 1) (0, 2) (0, 3) (0, 4) (1, 5) (2, 6) (3, 6) (4, 6) (5, 6)
(6, 7) (7, 7)
```

Experiment-5

Aim : WAP to implement 8 puzzle problem.

Program:

```
/ Program to print path from root node to destination node
// for N*N -1 puzzle algorithm using Branch and Bound
// The solution assumes that instance of puzzle is solvable
#include <bits/stdc++.h>
using namespace std;
#define N 3

// state space tree nodes
struct Node
{
    // stores the parent node of the current node
    // helps in tracing path when the answer is found
    Node* parent;

    // stores matrix
    int mat[N][N];

    // stores blank tile coordinates
    int x, y;

    // stores the number of misplaced tiles
    int cost;

    // stores the number of moves so far
    int level;
};

// Function to print N x N matrix
int printMatrix(int mat[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf("%d ", mat[i][j]);
        printf("\n");
    }
}

// Function to allocate a new node
```

```
Node* newNode(int mat[N][N], int x, int y, int newX,
              int newY, int level, Node* parent)
{
    Node* node = new Node;

    // set pointer for path to root
    node->parent = parent;

    // copy data from parent node to current node
    memcpy(node->mat, mat, sizeof node->mat);

    // move tile by 1 position
    swap(node->mat[x][y], node->mat[newX][newY]);

    // set number of misplaced tiles
    node->cost = INT_MAX;

    // set number of moves so far
    node->level = level;

    // update new blank tile cordinates
    node->x = newX;
    node->y = newY;

    return node;
}

// botton, left, top, right
int row[] = { 1, 0, -1, 0 };
int col[] = { 0, -1, 0, 1 };

// Function to calculate the number of misplaced tiles
// ie. number of non-blank tiles not in their goal position
int calculateCost(int initial[N][N], int final[N][N])
{
    int count = 0;
    for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
        if (initial[i][j] && initial[i][j] != final[i][j])
           count++;
    return count;
}

// Function to check if (x, y) is a valid matrix cordinate
```

```
int isSafe(int x, int y)
{
    return (x >= 0 && x < N && y >= 0 && y < N);
}

// print path from root node to destination node
void printPath(Node* root)
{
    if (root == NULL)
        return;
    printPath(root->parent);
    printMatrix(root->mat);

    printf("\n");
}

// Comparison object to be used to order the heap
struct comp
{
    bool operator()(const Node* lhs, const Node* rhs) const
    {
        return (lhs->cost + lhs->level) > (rhs->cost + rhs-
>level);
    }
};

// Function to solve N*N - 1 puzzle algorithm using
// Branch and Bound. x and y are blank tile coordinates
// in initial state
void solve(int initial[N][N], int x, int y,
           int final[N][N])
{
    // Create a priority queue to store live nodes of
    // search tree;
    priority_queue<Node*, std::vector<Node*>, comp> pq;

    // create a root node and calculate its cost
    Node* root = newNode(initial, x, y, x, y, 0, NULL);
    root->cost = calculateCost(initial, final);

    // Add root to list of live nodes;
    pq.push(root);

    // Finds a live node with least cost,
    // add its childrens to list of live nodes and
```

```
    // finally deletes it from the list.
    while (!pq.empty())
    {
        // Find a live node with least estimated cost
        Node* min = pq.top();

        // The found node is deleted from the list of
        // live nodes
        pq.pop();

        // if min is an answer node
        if (min->cost == 0)
        {
            // print the path from root to destination;
            printPath(min);
            return;
        }

        // do for each child of min
        // max 4 children for a node
        for (int i = 0; i < 4; i++)
        {
            if (isSafe(min->x + row[i], min->y + col[i]))
            {
                // create a child node and calculate
                // its cost
                Node* child = newNode(min->mat, min->x,
                              min->y, min->x + row[i],
                              min->y + col[i],
                              min->level + 1, min);
                child->cost = calculateCost(child->mat,
final);

                // Add child to list of live nodes
                pq.push(child);
            }
        }
    }
}

// Driver code
int main()
{
    // Initial configuration
    // Value 0 is used for empty space
```

```
    int initial[N][N] =
    {
        {1, 2, 3},
        {5, 6, 0},
        {7, 8, 4}
    };

    // Solvable Final configuration
    // Value 0 is used for empty space
    int final[N][N] =
    {
        {1, 2, 3},
        {5, 8, 6},
        {0, 7, 4}
    };

    // Blank tile coordinates in initial
    // configuration
    int x = 1, y = 2;

    solve(initial, x, y, final);

    return 0;
}
```

OUTPUT :

```
1 2 3
5 6 0
7 8 4

1 2 3
5 0 6
7 8 4

1 2 3
5 8 6
7 0 4

1 2 3
5 8 6
0 7 4
```

Experiment-6

Aim : WAP to implement Tower of Hanoi.

Program :

```
// C++ recursive function to
// solve tower of hanoi puzzle
#include <bits/stdc++.h>
using namespace std;

void towerOfHanoi(int n, char from_rod,
                  char to_rod, char aux_rod)
{
    if (n == 1)
    {
        cout << "Move disk 1 from rod " << from_rod <<
                            " to rod " << to_rod<<endl;
        return;
    }
    towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
    cout << "Move disk " << n << " from rod " << from_rod <<
                                  " to rod " << to_rod <<
endl;
    towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}

// Driver code
int main()
{
    int n = 4; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names
of rods
    return 0;
}
```

OUTPUT :

```
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
```

```
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 2 from rod B to rod A
Move disk 1 from rod C to rod A
Move disk 3 from rod B to rod C
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
```

Experiment-7

Aim : WAP to implement DFS.

Program :

```
// C++ program to print DFS traversal from
// a given vertex in a given graph
#include<iostream>
#include<list>
using namespace std;

// Graph class represents a directed graph
// using adjacency list representation
class Graph
{
	int V; // No. of vertices

	// Pointer to an array containing
	// adjacency lists
	list<int> *adj;

	// A recursive function used by DFS
	void DFSUtil(int v, bool visited[]);
public:
	Graph(int V); // Constructor

	// function to add an edge to graph
	void addEdge(int v, int w);
```

```
        // DFS traversal of the vertices
        // reachable from v
        void DFS(int v);
};

Graph::Graph(int V)
{
        this->V = V;
        adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
        adj[v].push_back(w); // Add w to v's list.
}

void Graph::DFSUtil(int v, bool visited[])
{
        // Mark the current node as visited and
        // print it
        visited[v] = true;
        cout << v << " ";

        // Recur for all the vertices adjacent
        // to this vertex
        list<int>::iterator i;
```

```
        for (i = adj[v].begin(); i != adj[v].end(); ++i)
                if (!visited[*i])
                        DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
        // Mark all the vertices as not visited
        bool *visited = new bool[V];
        for (int i = 0; i < V; i++)
                visited[i] = false;

        // Call the recursive helper function
        // to print DFS traversal
        DFSUtil(v, visited);
}

// Driver code
int main()
{
        // Create a graph given in the above diagram
        Graph g(4);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
```

```
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);
        cout << "Following is Depth First Traversal"
                        " (starting from vertex 2) \n";
        g.DFS(2);

        return 0;
}
```

OUTPUT :

```
Following is Depth First Traversal (starting from vertex 2)
2 0 1 3
```

Experiment-8

Aim : WAP to implement BFS.

Program :

```
// Program to print BFS traversal from a given
// source vertex. BFS(int s) traverses vertices
// reachable from s.
#include<iostream>
#include <list>
using namespace std;
// This class represents a directed graph using
// adjacency list representation
class Graph
{       int V; // No. of vertices
        // Pointer to an array containing adjacency
        // lists
        list<int> *adj;
public:
        Graph(int V); // Constructor
        // function to add an edge to graph
        void addEdge(int v, int w);
        // prints BFS traversal from a given source s
        void BFS(int s);
};
Graph::Graph(int V)
{       this->V = V;
        adj = new list<int>[V];
```

```
}

void Graph::addEdge(int v, int w)
{
        adj[v].push_back(w); // Add w to v's list.
}

void Graph::BFS(int s)
{
        // Mark all the vertices as not visited
        bool *visited = new bool[V];
        for(int i = 0; i < V; i++)
                visited[i] = false;

        // Create a queue for BFS
        list<int> queue;
        // Mark the current node as visited and enqueue it
        visited[s] = true;
        queue.push_back(s);
        // 'i' will be used to get all adjacent
        // vertices of a vertex
        list<int>::iterator i;
        while(!queue.empty())
        {
                // Dequeue a vertex from queue and print it
                s = queue.front();
                cout << s << " ";
```

```
            queue.pop_front();
            // Get all adjacent vertices of the dequeued
            // vertex s. If a adjacent has not been visited,
            // then mark it visited and enqueue it
            for (i = adj[s].begin(); i != adj[s].end(); ++i)
            {       if (!visited[*i])
                    {
                            visited[*i] = true;
                            queue.push_back(*i);
                    }
            }
        }
}

// Driver program to test methods of graph class
int main()
{
        // Create a graph given in the above diagram
        Graph g(4);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        cout << "Following is Breadth First Traversal "
```

```
            << "(starting from vertex 2) \n";

        g.BFS(2);


        return 0;

}
```

OUTPUT :

```
Following is Breadth First Traversal (starting from vertex
2)
2 0 3 1
```

Experiment-9

Aim: Study of various knowledge representation techniques.

There are mainly four ways of knowledge representation which are given as follows:

1. Logical Representation
2. Semantic Network Representation
3. Frame Representation
4. Production Rules

1. Logical Representation

Logical representation is a language with some concrete rules which deals with propositions and has no ambiguity in representation. Logical representation means drawing a conclusion based on various conditions. This representation lays down some important communication rules. It consists of precisely defined syntax and semantics which supports the sound inference. Each sentence can be translated into logics using syntax and semantics.

Syntax:

5. Syntaxes are the rules which decide how we can construct legal sentences in the logic.
6. It determines which symbol we can use in knowledge representation.
7. How to write those symbols.

2. Semantic Network Representation

Semantic networks are alternative of predicate logic for knowledge representation. In Semantic networks, we can represent our knowledge in the form of graphical networks. This network consists of nodes representing objects and arcs which describe the relationship between those objects. Semantic networks can categorize the object in different forms and can also link those objects. Semantic networks are easy to understand and can be easily extended.

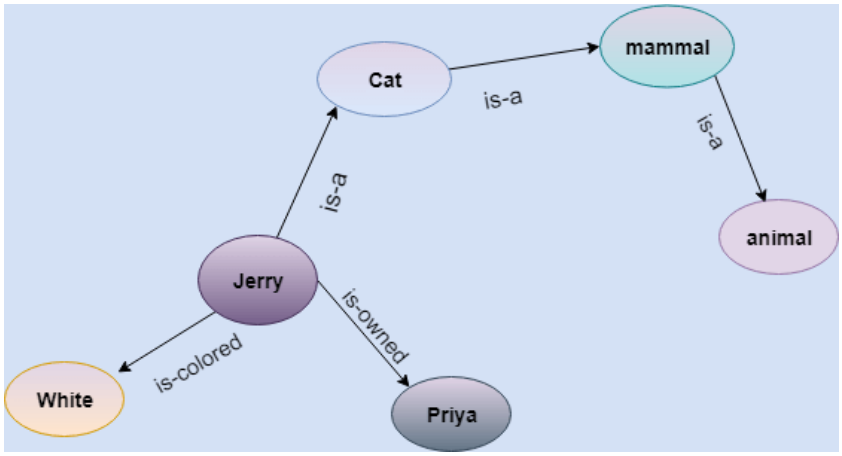
This representation consist of mainly two types of relations:

1. IS-A relation (Inheritance)
2. Kind-of-relation

Example: Following are some statements which we need to represent in the form of nodes and arcs.

Statements:

1. Jerry is a cat.
2. Jerry is a mammal
3. Jerry is owned by Priya.
4. Jerry is brown colored.
5. All Mammals are animal.



3. Frame Representation

A frame is a record like structure which consists of a collection of attributes and its values to describe an entity in the world. Frames are the AI data structure which divides knowledge into substructures by representing stereotypes situations. It consists of a collection of slots and slot values. These slots may be of any type and sizes. Slots have names and values which are called facets.

Facets: The various aspects of a slot is known as **Facets**. Facets are features of frames which enable us to put constraints on the frames. Example: IF-NEEDED facts are called when data of any particular slot is needed. A frame may consist of any number of slots, and a slot may include any number of facets and facets may have any number of values. A frame is also known as **slot-filter knowledge representation** in artificial intelligence.

4. Production Rules

Production rules system consist of (**condition, action**) pairs which mean, "If condition then action". It has mainly three parts:

- The set of production rules
- Working Memory
- The recognize-act-cycle

In production rules agent checks for the condition and if the condition exists then production rule fires and corresponding action is carried out. The condition part of the rule determines which rule may be applied to a problem. And the action part carries out the associated problem-solving steps. This complete process is called a recognize-act cycle.

The working memory contains the description of the current state of problems-solving and rule can write knowledge to the working memory. This knowledge match and may fire other rules.

Example:

- **IF (at bus stop AND bus arrives) THEN action (get into the bus)**
- **IF (on the bus AND paid AND empty seat) THEN action (sit down).**
- **IF (on bus AND unpaid) THEN action (pay charges).**