1.Implementation of the Tic-Tac-Toe problem

EXPLANATION-

```
# Tic-Tac-Toe Program using
# random number in Python

# importing all necessary libraries
import numpy as np
import random
from time import sleep

# Creates an empty board
def create_board():
	return(np.array([[0, 0, 0],
					[0, 0, 0],
					[0, 0, 0]]))

# Check for empty places on board
def possibilities(board):
	l = []

	for i in range(len(board)):
		for j in range(len(board)):

			if board[i][j] == 0:
				l.append((i, j))
	return(l)

# Select a random place for the player
def random_place(board, player):
	selection = possibilities(board)
	current_loc = random.choice(selection)
	board[current_loc] = player
	return(board)

# Checks whether the player has three
# of their marks in a horizontal row
def row_win(board, player):
	for x in range(len(board)):
		win = True

		for y in range(len(board)):
			if board[x, y] != player:
```

```
                    win = False
                    continue

            if win == True:
                return(win)
        return(win)

# Checks whether the player has three
# of their marks in a vertical row
def col_win(board, player):
        for x in range(len(board)):
            win = True

            for y in range(len(board)):
                if board[y][x] != player:
                    win = False
                    continue

            if win == True:
                return(win)
        return(win)

# Checks whether the player has three
# of their marks in a diagonal row
def diag_win(board, player):
        win = True
        y = 0
        for x in range(len(board)):
            if board[x, x] != player:
                win = False
        if win:
            return win
        win = True
        if win:
            for x in range(len(board)):
                y = len(board) - 1 - x
                if board[x, y] != player:
                    win = False
        return win

# Evaluates whether there is
# a winner or a tie
def evaluate(board):
        winner = 0
```

```
    for player in [1, 2]:
        if (row_win(board, player) or
            col_win(board,player) or
            diag_win(board,player)):

            winner = player

    if np.all(board != 0) and winner == 0:
        winner = -1
    return winner

# Main function to start the game
def play_game():
    board, winner, counter = create_board(), 0, 1
    print(board)
    sleep(2)

    while winner == 0:
        for player in [1, 2]:
            board = random_place(board, player)
            print("Board after " + str(counter) + " move")
            print(board)
            sleep(2)
            counter += 1
            winner = evaluate(board)
            if winner != 0:
                break
    return(winner)

# Driver Code
print("Winner is: " + str(play_game()))
```

2. Implementation of 8 puzzle problem

ALGORITHM:

The implementation of A* Algorithm involves maintaining two lists- OPEN and CLOSED.

OPEN contains those nodes that have been evaluated by the heuristic function but have not been expanded into

successors yet.

CLOSED contains those nodes that have already been visite

The algorithm is s follows-

Step-01:

Define a list OPEN.

Initially, OPEN consists solely of a single node, the start node S.

Step-02:

If the list is empty, return failure and exit.

Step-03:

Remove node n with the smallest value of f(n) from OPEN and move it to list CLOSED. If node n is a goal state, return success and exit.

Step-04:

Expand node n.

Step-05:

If any successor to n is the goal node, return success and the solution by tracing the path from goal node to S. Otherwise, go to Step-06.

Step-06:

For each successor node,

Apply the evaluation function f to the node.

If the node has not been in either list, add it to OPEN.

Step-07:

Go back to Step-02.

SOURCE CODE 2:

```
// 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 coordinates

node->x = newX;
node->y = newY;

return node; }

// bottom, 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 coordinate 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++)

} }

// 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;





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);

solve(initial, x, y, final);

return 0; }

3.Developing agent problems for real world problems

ALGORITHM :

SOURCE CODE:

// C/C++ program to solve fractional Knapsack Problem #include <bits/stdc++.h>

using namespace std;

// Structure for an item which stores weight and

// corresponding value of Item struct Item {

```
int value, weight;

// Constructor

Item(int value, int weight) {

this->value=value; this->weight=weight;

}


};

bool cmp(struct Item a, struct Item b) {


double r1 = (double)a.value / (double)a.weight; double r2 = (double)b.value / (double)b.weight; return r1 > r2;


}

// Main greedy function to solve problem

double fractionalKnapsack(int W, struct Item arr[], int n) {
```

Fractional knapsack problem is solved using greedy method in the following steps-

Step-01 : For each item, compute its value / weight ratio.

Step-02 : Arrange all the items in decreasing order of their value / weight ratio.

Step-03 : Start putting the items into the knapsack beginning from the item with the highest ratio.

Put as many items as you can into the knapsack.

// sorting Item on basis of ratio

sort(arr, arr + n, cmp);

int curWeight = 0; // Current weight in knapsack double finalvalue = 0.0; // Result (value in Knapsack) // Looping through all Items

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

// If adding Item won't overflow, add it completely if (curWeight + arr[i].weight <= W) {

curWeight += arr[i].weight; finalvalue += arr[i].value;

} else {

} }

// Returning final value return finalvalue;

```
}

// Driver code int main()

{

{

}

int remain = W - curWeight; finalvalue += arr[i].value

* ((double)remain

/ (double)arr[i].weight); break;
```

```
int W = 50; // Weight of knapsack

Item arr[] = { { 60, 10 }, { 100, 20 }, { 120, 30 } }; int n = sizeof(arr) / sizeof(arr[0]);

// Function call

cout << "Maximum value we can obtain = "

<< fractionalKnapsack(W, arr, n); return 0;
```

4.Developing Best First Search in real world problems

PSEUDOCODE:

```
Best-First-Search(Graph 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
```

SOURCE CODE :

```
// C++ program to implement Best First Search using priority // queue
#include <bits/stdc++.h>
using namespace std;

typedef pair<int, int> pi; vector<vector<pi> > graph;
// Function for adding edges to graph void addedge(int x, int y, int cost)
{
```

```
graph[x].push_back(make_pair(cost, y)); graph[y].push_back(make_pair(cost, x));

}

        //  Function For Implementing Best First Search

        //  Gives output path having lowest cost
         void best_first_search(int source, int target, int n) {

        vector<bool> visited(n, false); // MIN HEAP priority queue
```

```
} }

priority_queue<pi, vector<pi>, greater<pi> > pq; // sorting in pq gets done by first value of pair pq.push(make_pair(0,
source));

int s = source;

visited[s] = true; while (!pq.empty()) {

// Driver code to test above methods int main()

{
```

```
int x = pq.top().second;

// Displaying the path having lowest cost cout << x << " ";

pq.pop();

if (x == target)



break;

for (int i = 0; i < graph[x].size(); i++) {



if (!visited[graph[x][i].second]) { visited[graph[x][i].second] = true;










} }
```

```
// No. of Nodes

int v = 14;

graph.resize(v);

// The nodes shown in above example(by alphabets) are // implemented using integers addedge(x,y,cost);
addedge(0, 1, 3);
```

```
addedge(0, 2, 6); addedge(0, 3, 5); addedge(1, 4, 9); addedge(1, 5, 8); addedge(2, 6, 12); addedge(2, 7, 14);
addedge(3, 8, 7);
```

```
pq.push(make_pair(graph[x][i].first,graph[x][i].second));
```

addedge(8, 9, 5); addedge(8, 10, 6); addedge(9, 11, 1); addedge(9, 12, 10); addedge(9, 13, 2); int source = 0;

int target = 9;

// Function call best_first_search(source, target, v);

return 0; }

5.Developing Depth first search in real world problems

Algorithm

Let's say you're stuck in a corn maze. It's been hours since you've drank water or eaten anything. Fortunately, you remember your introduction of algorithms class and do a depth first search of the entire maze.

1. Place your right hand on the corn wall.
2. Walk without removing your hand

One of the benefits of depth first search is the entire graph will be traversed with each node being one adjacent to the next one, and the backtracking also following nodes next to each other. This follows the walking in a maze analogy.

6.Developing A* algorithm in real world problems

#include using namespace std; #define ROW 9 #define COL 10 typedef pair Pair; typedef pair > pPair; struct cell { int parent_i, parent_j; double f, g, h; }; bool isValid(int row, int col) { return (row >= 0) && (row < ROW) && (col >= 0) && (col < COL); } bool isUnBlocked(int grid[][COL], int row, int col) { if (grid[row][col] == 1) return (true); else return (false); } bool isDestination(int row, int col, Pair dest) { if (row == dest.first && col == dest.second) return (true); else return (false); } double calculateHValue(int row, int col, Pair dest) { return ((double)sqrt((row - dest.first) * (row - dest.first) + (col - dest.second) * (col - dest.second))); } void tracePath(cell cellDetails[][COL],

```
Pair dest) { printf("\nThe Path is "); int row = dest.first; int col = dest.second; stack Path; while
(!(cellDetails[row][col].parent_i == row && cellDetails[row][col].parent_j == col)) {
Path.push(make_pair(row, col)); int temp_row = cellDetails[row][col].parent_i; int temp_col =
cellDetails[row][col].parent_j; row = temp_row; col = temp_col; } Path.push(make_pair(row, col));
while (!Path.empty()) { pair p = Path.top(); Path.pop(); printf("-> (%d,%d) ", p.first, p.second); }
return; } void aStarSearch(int grid[][COL], Pair src, Pair dest) { if (isValid(src.first, src.second) ==
false) { printf("Source is invalid\n"); return; } if (isValid(dest.first, dest.second) == false) {
printf("Destination is invalid\n"); return; } if (isUnBlocked(grid, src.first, src.second) == false ||
isUnBlocked(grid, dest.first, dest.second) == false) { printf("Source or the destination is
blocked\n"); return; } if (isDestination(src.first, src.second, dest) == true) { printf("We are already
at the destination\n"); return; } bool closedList[ROW][COL]; memset(closedList, false,
sizeof(closedList)); cell cellDetails[ROW][COL]; int i, j; for (i = 0; i < ROW; i++) { for (j = 0; j <
COL; j++) { cellDetails[i][j].f = FLT_MAX; cellDetails[i][j].g = FLT_MAX; cellDetails[i][j].h =
FLT_MAX; cellDetails[i][j].parent_i = -1; cellDetails[i][j].parent_j = -1; } } i = src.first, j =
src.second; cellDetails[i][j].f = 0.0; cellDetails[i][j].g = 0.0; cellDetails[i][j].h = 0.0;
cellDetails[i][j].parent_i = i; cellDetails[i][j].parent_j = j; set openList;
openList.insert(make_pair(0.0, make_pair(i, j))); bool foundDest = false; while
(!openList.empty()) { pPair p = *openList.begin(); // Remove this vertex from the open list
openList.erase(openList.begin()); // Add this vertex to the closed list i = p.second.first; j =
p.second.second; closedList[i][j] = true; double gNew, hNew, fNew; if (isValid(i - 1, j) == true) { if
(isDestination(i - 1, j, dest) == true) { cellDetails[i - 1][j].parent_i = i; cellDetails[i - 1][j].parent_j =
j; printf("The destination cell is found\n"); tracePath(cellDetails, dest); foundDest = true; return; }
else if (closedList[i - 1][j] == false && isUnBlocked(grid, i - 1, j) == true) { gNew =
cellDetails[i][j].g + 1.0; hNew = calculateHValue(i - 1, j, dest); fNew = gNew + hNew; if
(cellDetails[i - 1][j].f == FLT_MAX || cellDetails[i - 1][j].f > fNew) { openList.insert(make_pair(
fNew, make_pair(i - 1, j))); cellDetails[i - 1][j].f = fNew; cellDetails[i - 1][j].g = gNew; cellDetails[i -
1][j].h = hNew; cellDetails[i - 1][j].parent_i = i; cellDetails[i - 1][j].parent_j = j; } } } if (isValid(i + 1,
j) == true) { if (isDestination(i + 1, j, dest) == true) { cellDetails[i + 1][j].parent_i = i; cellDetails[i +
1][j].parent_j = j; printf("The destination cell is found\n"); tracePath(cellDetails, dest); foundDest
= true; return; } else if (closedList[i + 1][j] == false && isUnBlocked(grid, i + 1, j) == true) { gNew
= cellDetails[i][j].g + 1.0; hNew = calculateHValue(i + 1, j, dest); fNew = gNew + hNew; if
(cellDetails[i + 1][j].f == FLT_MAX || cellDetails[i + 1][j].f > fNew) { openList.insert(make_pair(
fNew, make_pair(i + 1, j))); cellDetails[i + 1][j].f = fNew; cellDetails[i + 1][j].g = gNew; cellDetails[i
+ 1][j].h = hNew; cellDetails[i + 1][j].parent_i = i; cellDetails[i + 1][j].parent_j = j; } } } if (isValid(i, j
+ 1) == true) { if (isDestination(i, j + 1, dest) == true) { cellDetails[i][j + 1].parent_i = i;
cellDetails[i][j + 1].parent_j = j; printf("The destination cell is found\n"); tracePath(cellDetails,
dest); foundDest = true; return; } else if (closedList[i][j + 1] == false && isUnBlocked(grid, i, j + 1)
== true) { gNew = cellDetails[i][j].g + 1.0; hNew = calculateHValue(i, j + 1, dest); fNew = gNew +
hNew; if (cellDetails[i][j + 1].f == FLT_MAX || cellDetails[i][j + 1].f > fNew) {
openList.insert(make_pair( fNew, make_pair(i, j + 1))); cellDetails[i][j + 1].f = fNew; cellDetails[i][j
+ 1].g = gNew; cellDetails[i][j + 1].h = hNew; cellDetails[i][j + 1].parent_i = i; cellDetails[i][j +
1].parent_j = j; } } } if (isValid(i, j - 1) == true) { if (isDestination(i, j - 1, dest) == true) {
cellDetails[i][j - 1].parent_i = i; cellDetails[i][j - 1].parent_j = j; printf("The destination cell is
found\n"); tracePath(cellDetails, dest); foundDest = true; return; } else if (closedList[i][j - 1] ==
false && isUnBlocked(grid, i, j - 1) == true) { gNew = cellDetails[i][j].g + 1.0; hNew =
calculateHValue(i, j - 1, dest); fNew = gNew + hNew; if (cellDetails[i][j - 1].f == FLT_MAX ||
cellDetails[i][j - 1].f > fNew) { openList.insert(make_pair( fNew, make_pair(i, j - 1))); cellDetails[i][j
- 1].f = fNew; cellDetails[i][j - 1].g = gNew; cellDetails[i][j - 1].h = hNew; cellDetails[i][j -
1].parent_i = i; cellDetails[i][j - 1].parent_j = j; } } } if (isValid(i - 1, j + 1) == true) { if
(isDestination(i - 1, j + 1, dest) == true) { cellDetails[i - 1][j + 1].parent_i = i; cellDetails[i - 1][j +
1].parent_j = j; printf("The destination cell is found\n"); tracePath(cellDetails, dest); foundDest =
true; return; } else if (closedList[i - 1][j + 1] == false && isUnBlocked(grid, i - 1, j + 1) == true) {
gNew = cellDetails[i][j].g + 1.414; hNew = calculateHValue(i - 1, j + 1, dest); fNew = gNew +
hNew; if (cellDetails[i - 1][j + 1].f == FLT_MAX || cellDetails[i - 1][j + 1].f > fNew) {
openList.insert(make_pair( fNew, make_pair(i - 1, j + 1))); cellDetails[i - 1][j + 1].f = fNew;
cellDetails[i - 1][j + 1].g = gNew; cellDetails[i - 1][j + 1].h = hNew; cellDetails[i - 1][j + 1].parent_i
= i; cellDetails[i - 1][j + 1].parent_j = j; } } } if (isValid(i - 1, j - 1) == true) { if (isDestination(i - 1, j -
1, dest) == true) { cellDetails[i - 1][j - 1].parent_i = i; cellDetails[i - 1][j - 1].parent_j = j; printf("The
destination cell is found\n"); tracePath(cellDetails, dest); foundDest = true; return; } else if
(closedList[i - 1][j - 1] == false && isUnBlocked(grid, i - 1, j - 1) == true) { gNew =
cellDetails[i][j].g + 1.414; hNew = calculateHValue(i - 1, j - 1, dest); fNew = gNew + hNew; if
(cellDetails[i - 1][j - 1].f == FLT_MAX || cellDetails[i - 1][j - 1].f > fNew) {
openList.insert(make_pair( fNew, make_pair(i - 1, j - 1))); cellDetails[i - 1][j - 1].f = fNew;
cellDetails[i - 1][j - 1].g = gNew; cellDetails[i - 1][j - 1].h = hNew; cellDetails[i - 1][j - 1].parent_i =
i; cellDetails[i - 1][j - 1].parent_j = j; } } } if (isValid(i + 1, j + 1) == true) { if (isDestination(i + 1, j +
1, dest) == true) { cellDetails[i + 1][j + 1].parent_i = i; cellDetails[i + 1][j + 1].parent_j = j;
printf("The destination cell is found\n"); tracePath(cellDetails, dest); foundDest = true; return; }
else if (closedList[i + 1][j + 1] == false && isUnBlocked(grid, i + 1, j + 1) == true) { gNew =
cellDetails[i][j].g + 1.414; hNew = calculateHValue(i + 1, j + 1, dest); fNew = gNew + hNew; if
(cellDetails[i + 1][j + 1].f == FLT_MAX || cellDetails[i + 1][j + 1].f > fNew) {
openList.insert(make_pair( fNew, make_pair(i + 1, j + 1))); cellDetails[i + 1][j + 1].f = fNew;
cellDetails[i + 1][j + 1].g = gNew; cellDetails[i + 1][j + 1].h = hNew; cellDetails[i + 1][j +
1].parent_i = i; cellDetails[i + 1][j + 1].parent_j = j; } } } if (isValid(i + 1, j - 1) == true) { if
(isDestination(i + 1, j - 1, dest) == true) { cellDetails[i + 1][j - 1].parent_i = i; cellDetails[i + 1][j -
1].parent_j = j; printf("The destination cell is found\n"); tracePath(cellDetails, dest); foundDest =
true; return; } else if (closedList[i + 1][j - 1] == false && isUnBlocked(grid, i + 1, j - 1) == true) {
gNew = cellDetails[i][j].g + 1.414; hNew = calculateHValue(i + 1, j - 1, dest); fNew = gNew +
hNew; if (cellDetails[i + 1][j - 1].f == FLT_MAX || cellDetails[i + 1][j - 1].f > fNew) {
openList.insert(make_pair( fNew, make_pair(i + 1, j - 1))); cellDetails[i + 1][j - 1].f = fNew;
cellDetails[i + 1][j - 1].g = gNew; cellDetails[i + 1][j - 1].h = hNew; cellDetails[i + 1][j - 1].parent_i
= i; cellDetails[i + 1][j - 1].parent_j = j; } } } } } if (foundDest == false) printf("Failed to find the
Destination Cell\n"); return; } int main() { int grid[ROW][COL] = { { 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 }, { 1,
1, 1, 0, 1, 1, 1, 0, 1, 1 }, { 1, 1, 1, 0, 1, 1, 0, 1, 0, 1 }, { 0, 0, 1, 0, 1, 0, 0, 0, 0, 1 }, { 1, 1, 1, 0, 1, 1,
1, 0, 1, 0 }, { 1, 0, 1, 1, 1, 1, 0, 1, 0, 0 }, { 1, 0, 0, 0, 0, 1, 0, 0, 0, 1 }, { 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 }, {
1, 1, 1, 0, 0, 0, 1, 0, 0, 1 } }; Pair src = make_pair(8, 0); Pair dest = make_pair(0, 0);
aStarSearch(grid, src, dest); return (0); }
```

7.Implementation of constraint satiation problems

ALGORITHM:

1. Graph / map colouring problems are those where the nodes are assigned colors such that the adjacent connected nodes / regions don’t have the same color assigned.

2. At the same time, it is required to use the minimum number of colors possible – called the chromatic number.

3. Start by coloring the first node with a color.

4. Color the subsequent connected nodes with a different color. 5. Check at every step that it satisfies the condition.

SOURCE CODE:

```
class Graph:
```

```
def __init__(self, edges, N):

self.adj = [[] for _ in range(N)] for (src, dest) in edges:

self.adj[src].append(dest)

self.adj[dest].append(src) def colorGraph(graph):

result = {}
for u in range(N):

assigned = set([result.get(i) for i in graph.adj[u] if i in result]) color = 1
for c in assigned:

if color != c: break

color = color + 1 result[u] = color

for v in range(N):
print("Color assigned to vertex", v, "is", colors[result[v]])

if __name__ == '__main__':
colors = ["","BLUE", "GREEN", "RED", "YELLOW", "ORANGE"]
```

edges = [(0,1), (0,3), (0,4), (1,0), (1,4), (1,2), (2,1), (2,3), (3,0), (3,4), (3,5), (4,0), (4,1), (4,2), (4,3), (4,5), (4,6), (5,3), (

5,6), (5,4), (6,4), (6,5)] N=7

graph = Graph(edges, N) colorGraph(graph)

8.AIM: To find the best value using Minimax algorithm from the given input.

ALGORITHM:

1. Minimax algorithm, also known as Maximin, is a technique in AI to minimize the maximum loss or maximize the minimum gain while decision making.

2. A recursive algorithm, used in two player games (predominantly) where a tree structure is used to denote the position of the player.

3. The goal is to obtain the maximum possible score without knowing the decision of the other players in the game.

4. The levels of the tree are alternatively names as max and min, where the root level takes max.

SOURCE CODE:

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

int minimax(int depth, int nodeIndex, bool isMax, int scores[], int h)

{

}

if (depth == h)
return scores[nodeIndex];

if (isMax)
return max(minimax(depth+1, nodeIndex*2, false, scores, h),
minimax(depth+1, nodeIndex*2 + 1, false, scores, h));

else
return min(minimax(depth+1, nodeIndex*2, true, scores, h),

minimax(depth+1, nodeIndex*2 + 1, true, scores, h));

int log2(int n)
{
return (n==1)? 0 : 1 + log2(n/2); }

int main() {

}
```

```
int scores[] = {3, 5, 2, 9, 12, 5, 23, 23};

int n = sizeof(scores)/sizeof(scores[0]);

int h = log2(n);

int res = minimax(0, 0, true, scores, h);

cout << "The optimal value is : " << res << endl; return 0;
```

9.AIM : To implement unification for real-world problems. i) UNFICATION

Algorithm:-

Step. 1: If Ψ 1 or Ψ 2 is a variable or constant, then: a) If Ψ 1 or Ψ 2 are identical, then return NIL.

b) Else if Ψ 1 is a variable,

a. then if Ψ 1 occurs in Ψ 2 , then return FAILURE b. Else return { (Ψ 2 / Ψ 1)}.

c) Else if Ψ 2 is a variable,

a. If Ψ 2 occurs in Ψ 1 then return FAILURE,

b. Else return {(Ψ 1 / Ψ 2)}.

d) Else return FAILURE.

Step.2: If the initial Predicate symbol in Ψ 1 and Ψ 2 are not same, then return FAILURE. Step. 3: IF Ψ 1 and Ψ 2 have a different number of arguments, then return FAILURE. Step. 4: Set Substitution set(SUBST) to NIL.

Step. 5: For i=1 to the number of elements in Ψ 1 .

a) Call Unify function with the ith element of Ψ 1 and ith element of Ψ 2 , and put the result into S.

b) If S = failure then returns Failure

c) If S ≠ NIL then do,

a. Apply S to the remainder of both L1 and L2.

b. SUBST= APPEND(S, SUBST).

Step.6: Return SUBST

SOURCE CODE :

```
#include<stdio.h> int no_of_pred; int no_of_arg[10]; int i,j;

char nouse;

char predicate[10]; char argument[10][10]; void unify();
```

```
void display();

void chk_arg_pred();



void main() {

char ch; do{



printf("\t=========PROGRAM FOR UNIFICATION=========\n"); printf("\nEnter Number of Predicates:- [ ]\b\b");
scanf("%d",&no_of_pred);

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



{

scanf("%c",&nouse); //to accept "enter" as a character

printf("\nEnter Predicate %d:-[ ]\b\b",i+1);

scanf("%c",&predicate[i]);

printf("\n\tEnter No.of Arguments for Predicate %c:-[ ]\b\b",predicate[i]); scanf("%d",&no_of_arg[i]);



for(j=0;j<no_of_arg[i];j++) {



scanf("%c",&nouse);

printf("\n\tEnter argument %d:( )\b\b",j+1); scanf("%c",&argument[i][j]);
```

```
} }

display();

chk_arg_pred();

printf("Do you want to continue(y/n): "); scanf("%c",&ch);

}while(ch == 'y'); }

void display() {

printf("\n\t=======PREDICATES ARE======"); for(i=0;i<no_of_pred;i++)

{

printf("\n\t%c(",predicate[i]); for(j=0;j<no_of_arg[i];j++)

{ printf("%c",argument[i][j]); if(j!=no_of_arg[i]-1)
```

```
printf(","); }

printf(")"); }

}
void chk_arg_pred() {
int pred_flag=0;
int arg_flag=0;

/*======Checking Prediactes========*/ for(i=0;i<no_of_pred-1;i++)
{

if(predicate[i]!=predicate[i+1])
{
printf("\nPredicates not same.."); printf("\nUnification cannot progress!"); pred_flag=1;

break;
```

```
} }

/*=====Chking No of Arguments====*/ if(pred_flag!=1)

{

for(i=0;i<no_of_arg[i]-1;i++) {

if(no_of_arg[i]!=no_of_arg[i+1])

{

printf("\nArguments Not Same..!"); arg_flag=1;

break;

} }
```

```
}

if(arg_flag==0&&pred_flag!=1)

unify(); }

/*==========UNIFY FUNCTION=========*/ void unify()

{

int flag=0; for(i=0;i<no_of_pred-1;i++) {

for(j=0;j<no_of_arg[i];j++) {

if(argument[i][j]!=argument[i+1][j]) {

if(flag==0)
printf("\n\t======SUBSTITUTION IS======"); printf("\n\t%c/%c",argument[i+1][j],argument[i][j]);

flag++; }

} }

if(flag==0)
{ printf("\nArguments are Identical...");

printf("\nNo need of Substitution\n"); }

}
```

OUTPUT :

RESULT : Implementation of unification for real-world problems.

ii) RESOLUTION

PSEUDOCODE :

SOURCE CODE :

```
class A:

def rk(self):

print(" In class A") class B(A):

def rk(self):

print(" In class B")

class C(A): def rk(self):

print("In class C")

# classes ordering class D(B, C):

pass r = D()

r.rk()
```

10.JOB SCHEDULING PRIBLEM

Algorithm :-

```
JOB_SCHEDULING( J, D, P )

// Description : Schedule the jobs using the greedy approach which maximizes the profit

// Input :

J: Array of N jobs

D: Array of the deadline for each job

P: Array of profit associated with each job

// Output : Set of scheduled job which gives maximum profit

Sort all jobs in J in decreasing order of profit

S ← Φ // S is set of scheduled jobs, initially it is empty SP ← 0 // Sum is the profit earned

for i ← 1 to N do

if Job J[i] is feasible then

Schedule the job in the latest possible free slot meeting its deadline. S ← S ∪ J[i]

SP ← SP + P[i]

end

end
```

Source Code ;

```
def printJobScheduling(arr, t):

n = len(arr)

for i in range(n):
for j in range(n - 1 - i):

if arr[j][2] < arr[j + 1][2]:
arr[j], arr[j + 1] = arr[j + 1], arr[j]

result = [False] * t
job = ['-1'] * t
for i in range(len(arr)):

for j in range(min(t - 1, arr[i][1] - 1), -1, -1):

if result[j] is False: result[j] = True job[j] = arr[i][0] break

print(job)
arr = [['a', 2, 100],
['b', 1, 19], ['c', 2, 27], ['d', 1, 25], ['e', 3, 15]]

print("Following is maximum profit sequence of jobs") printJobScheduling(arr, 3)
```