Experiment no: 9

Aim: Write a program for Graph Coloring algorithm using backtracking approach.

Software required: C

Theory:

Aim: Write a program for graph coloring algorithm using backtracking approach.

Theory: The least possible value of 'm' required to color the graph successfully is known as the chromatic number of given graph.

Naive approach: In this approach using brute force, we find all permutations of color combinations that can colour the graph if any of permutations is valid for given graph colors we output the results otherwise not.

Backtracking algorithm:

The backtracking algorithm makes the process efficient by avoiding many bad decisions made in naive approach, there we color a single vertex and then move to its adjacent vertex to color it with different color.

Algorithm:

1) Different colors:
   a) Confirm whether it is valid to color the current vertex with the current color.
   b) If yes then colour it otherwise try a different color.
   c) Check if all vertices are colored or not.
   d) If not then move to next adjacent uncolored vertex.
2) If no other colour is available then backtrack.

Conclusion: Thus, graph coloring algorithm using backtracking approach has been studied and implemented.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
//Number of vertices
#define numOfVertices 4
// 0 - Green, 1 - Blue
char colors[][30] = {"Green", "Blue"};
int color_used = 2;
int colorCount;
//Graph connections
int graph[numOfVertices][numOfVertices] = {{0, 1, 0, 1},
{1, 0, 1, 0},
{0, 1, 0, 1},
{1, 0, 1, 0}};
//Vertex
typedef struct{
char name;
bool colored;
int color;
} Vertex;
//VertexList
Vertex *vertexArray[numOfVertices];
//function check if a vertex has any uncolored neighbors
int hasUncoloredNeighbours(int idx){
for(int i=0;i<numOfVertices; i++){
if(graph[idx][i] == 1 && vertexArray[i]->colored == false)
return i;
}
return -1;
}
```

```
//Function to check whether it is valid to color with color[colorIndex]
bool canColorWith(int colorIndex, int vertex) {
Vertex *neighborVertex;
for(int i=0; i<numOfVertices; i++){
//skipping if vertex are not connected
if(graph[vertex][i] == 0) continue;
neighborVertex = vertexArray[i];
if(neighborVertex->colored && neighborVertex->color == colorIndex)
return false;
}
return true;
}
//function to color the vertex
bool setColors(int idx){
int colorIndex, unColoredIdx;
//Step: 1
for (colorIndex=0; colorIndex<color_used; colorIndex++){
// Step-1.1 : checking validity
if(!canColorWith(colorIndex, idx)) continue;
//Step-1.2 : coloring
vertexArray[idx]->color = colorIndex;
vertexArray[idx]->colored = true;
colorCount++;
//Step-1.3 : Whether all vertices colored?
if(colorCount == numOfVertices ) //Base Case
return true;
//Step-1.4 : Next uncolored vertex
while((unColoredIdx = hasUncoloredNeighbours(idx)) != -1){
if(setColors(unColoredIdx))
return true;
}
```

```
}
// Step-2 : Backtracking
vertexArray[idx]->color = -1;
vertexArray[idx]->colored = false;
return false;
}
int main()
{
//define Vertex
Vertex vertexA, vertexB, vertexC, vertexD;
vertexA.name = 'A';
vertexB.name = 'B';
vertexC.name = 'C';
vertexD.name = 'D';
//add veritces to the array
vertexArray[0] = &vertexA;
vertexArray[1] = &vertexB;
vertexArray[2] = &vertexC;
vertexArray[3] = &vertexD;
//set default values (uncolor) for all vertices
for(int i=0; i<numOfVertices;i++){
vertexArray[i]->colored = false;
vertexArray[i]->color = -1;
}
//start coloring with first vertex
bool hasSolution = setColors(0);
//check if all vertices was successfully colored
if (!hasSolution)
printf("No Solution");
else {
for(int i=0; i<numOfVertices;i++){
```

```
printf("%c %s \n",vertexArray[i]->name,colors[vertexArray[i]->color]);
}
}
return 0;
}
```

Analysis:

Algorithm	Graph colouring.
Time Complexity	O(m^V)
Space complexity	O(V)

Output:

Conclusion: Graph colouring has been studied and implemented.