PRN: 2020BTECS00037

NAME: Khushi Nitinkumar Patel

BATCH: T5

Assignment no10: Backtracking

1) Implement the following using Back Tracking

a) 4-Queen's problem

- **Code:**

```
 #include<bits/stdc++.h>

using namespace std;

bool isSafe(int** arr, int x, int y, int n){
    for(int row=0; row<x; row++){
        if(arr[row][y]==1){
            return false;
        }
    }

    int row = x;
    int col = y;

    while(row>=0 && col>0){
        if(arr[row][col] == 1){
            return false;
        }

        row--;
        col--;
    }


     row = x;
     col = y;

    while(row>=0 && col<n){
        if(arr[row][col] == 1){
            return false;
```

```
        }

        row--;
        col++;
    }

    return true;
}

bool nQueen(int**arr, int x, int n){
    if(x>=n){
        return true;
    }

    for(int col=0; col<n; col++){
        if(isSafe(arr,x,col,n)){
            arr[x][col]=1;

            if (nQueen(arr,x+1,n)){
                return true;
            }

            arr[x][col]=0;  //backtracking step
        }
    }
    return false;

}

int main(){
    int n=4;

    int** arr=new int*[n];
    for(int i=0; i<n; i++){
        arr[i]=new int [n];
        for(int j=0; j<n; j++){
            arr[i][j]=0;
        }
    }
    if(nQueen(arr,0,n)){
        for(int i=0; i<n; i++){
            for(int j=0; j<n; j++){
                cout<<arr[i][j]<<" ";
            }
            cout<<endl;
        }
    }
}
```

Output:

```
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0
```

Time Complexity:
O(N!)

b) 8-Queen's problem

- **Code:**

```
#include<bits/stdc++.h>

#define n 8
using namespace std;

bool isSafe(int** arr, int x, int y){
    for(int row=0; row<x; row++){
        if(arr[row][y]==1){
            return false;
        }
    }

    int row = x;
    int col = y;

    while(row>=0 && col>0){
        if(arr[row][col] == 1){
            return false;
        }

        row--;
        col--;
    }


     row = x;
     col = y;

    while(row>=0 && col<n){
        if(arr[row][col] == 1){
            return false;
        }
```

```
        row--;
        col++;
    }

    return true;
}

bool nQueen(int**arr, int x){
    if(x>=n){
        return true;
    }

    for(int col=0; col<n; col++){
        if(isSafe(arr,x,col)){
            arr[x][col]=1;

            if (nQueen(arr,x+1)){
                return true;
            }

            arr[x][col]=0;  //backtracking step
        }
    }
    return false;

}


int main(){

    int** arr=new int*[n];
    for(int i=0; i<n; i++){
        arr[i]=new int [n];
        for(int j=0; j<n; j++){
            arr[i][j]=0;
        }
    }

    if(nQueen(arr,0)){
        for(int i=0; i<n; i++){
            for(int j=0; j<n; j++){
                cout<<arr[i][j]<<" ";
            }
            cout<<endl;
        }
    }

}
```

Output:

```
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0
```

Time Complexity:
O(N!)

c) Hamiltonian cycle

- Algorithm:
  1. First create an empty path array and add vertex 0 to it.
  2. Add other vertices starting from the vertex 1.
  3. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added.
  4. If we find such a vertex, we add the vertex as part of the solution and if don't find a vertex then we return false.

Code:

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

#define V 5

void printSolution(int path[]);

bool isSafe(int v, bool graph[V][V],int path[], int pos)
{
    if (graph [path[pos - 1]][ v ] == 0)
        return false;

    for (int i = 0; i < pos; i++)
        if (path[i] == v)
            return false;
```

```
    return true;
}

bool hamCycleUtil(bool graph[V][V],int path[], int pos)
{
    if (pos == V)
    {
        if (graph[path[pos - 1]][path[0]] == 1)
            return true;
        else
            return false;
    }

    for (int v = 1; v < V; v++)
    {
        if (isSafe(v, graph, path, pos))
        {
            path[pos] = v;

            if (hamCycleUtil (graph, path, pos + 1) == true)
                return true;

            path[pos] = -1;
        }
    }

    return false;
}

bool hamCycle(bool graph[V][V])
{
    int *path = new int[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;

    path[0] = 0;
    if (hamCycleUtil(graph, path, 1) == false )
    {
        cout << "\nSolution does not exist";
        return false;
    }

    printSolution(path);
    return true;
}

void printSolution(int path[])
{
```

```
    cout << "Solution Exists:"
            " Following is one Hamiltonian Cycle \n";
    for (int i = 0; i < V; i++)
        cout << path[i] << " ";

    cout << path[0] << " ";
    cout << endl;
}

int main()
{
    bool graph1[V][V] = {{0, 1, 0, 1, 0},
                        {1, 0, 1, 1, 1},
                        {0, 1, 0, 0, 1},
                        {1, 1, 0, 0, 1},
                        {0, 1, 1, 1, 0}};

    hamCycle(graph1);

    bool graph2[V][V] = {{0, 1, 0, 1, 0},
                        {1, 0, 1, 1, 1},
                        {0, 1, 0, 0, 1},
                        {1, 1, 0, 0, 0},
                        {0, 1, 1, 0, 0}};


    hamCycle(graph2);

    return 0;
}
```

Output:

```
Solution Exists: Following is one Hamiltonian Cycle
0 1 2 4 3 0
```

Time complexity: O(n^n)

d) Graph coloring Problem

- **Code:**

```
#include <bits/stdc++.h>
```

```cpp
using namespace std;
#define V 4
void printSolution(int color[]);
bool isSafe(bool graph[V][V], int color[])
{
    for (int i = 0; i < V; i++)
        for (int j = i + 1; j < V; j++)
            if (graph[i][j] && color[j] == color[i])
                return false;
    return true;
}
bool graphColoring(bool graph[V][V], int m, int i,
                   int color[V])
{
    if (i == V)
    {
        if (isSafe(graph, color))
        {
            printSolution(color);
            return true;
        }
        return false;
    }
    for (int j = 1; j <= m; j++)
    {
        color[i] = j;
        if (graphColoring(graph, m, i + 1, color))
            return true;
        color[i] = 0;
    }
    return false;
}
void printSolution(int color[])
{
    cout << "For the solution, following are the assigned colors \n";
    for (int i = 0; i < V; i++)
        cout << " " << color[i];
    cout << "\n";
}
int main()
{
    bool graph[V][V] = {
        {0, 1, 1, 1},
        {1, 0, 1, 0},
        {1, 1, 0, 1},
        {1, 0, 1, 0},
    };
    int m = 3;
```

```
    int color[V];
    for (int i = 0; i < V; i++)
        color[i] = 0;
    if (!graphColoring(graph, m, 0, color))
        cout << "Solution does not exist";
    return 0;
}
```

Output:

```
For the solution, following are the assigned colors
 1 2 3 2
```

Time complexity: O(m^v)

2) Implement following problem using graph traversal Technique

a) Check whether a graph is Bipartite or not using Breadth First Search (BFS)

- Algorithm:
  1. Assign colour Red to the source vertex (putting into set U).
  2. Colour all the neighbours with colour Blue (putting into set V).
  3. Colour all neighbour's neighbour with colour Red (putting into set U).
  4. This way, assign colour to all vertices such that it satisfies all the constraints of m way colouring problem where m = 2.
  5. While assigning colours, if we find a neighbour which is coloured with same colour as current vertex, then the graph cannot be coloured with 2 vertices (or graph is not Bipartite)

Code:

```
#include <iostream>
#include <queue>
#define V 4
using namespace std;
bool isBipartite(int G[][V], int src)
{
    int colorArr[V];
    for (int i = 0; i < V; ++i)
        colorArr[i] = -1;
```

```cpp
    colorArr[src] = 1;
    queue<int> q;
    q.push(src);
    while (!q.empty())
    {
        int u = q.front();
        q.pop();
        if (G[u][u] == 1)
            return false;
        for (int v = 0; v < V; ++v)
        {
            if (G[u][v] && colorArr[v] == -1)
            {
                colorArr[v] = 1 - colorArr[u];
                q.push(v);
            }
            else if (G[u][v] && colorArr[v] == colorArr[u])
                return false;
        }
    }
    return true;
}
int main()
{
    int G[][V] = {{0, 1, 0, 1},
                  {1, 0, 1, 0},
                  {0, 1, 0, 1},
                  {1, 0, 1, 0}};
    isBipartite(G, 0) ? cout << "The given graph is Bipartite" : cout << "The
given graph is not Bipartite";
    return 0;
}
```

Output:

```
tempCodeRunnerFile } ; if ($?) { .\tempC
The given graph is Bipartite
```

```cpp
    int G[][V] = {{0, 1, 0, 1},
                  {1, 0, 1, 1},
                  {1, 1, 0, 1},
                  {1, 0, 1, 0}};
```

```
The given graph is not Bipartite
```

b) Find Articulation Point in Graph using Depth First Search (DFS) and mention whether Graph is Biconnected or not

- **Code:**

```
 #include <bits/stdc++.h>

using namespace std;

void APUtil(vector<int> adj[], int u, bool visited[], int disc[], int low[], int
&time, int parent, bool isAP[])
{
    int children = 0;
    visited[u]  =  true;
    disc[u] = low[u] = ++time;
    for (auto v : adj[u])
    {
        if (!visited[v])
        {
            children++;
            APUtil(adj, v, visited, disc, low, time, u, isAP);
            low[u] = min(low[u], low[v]);
            if (parent != -1 && low[v] >= disc[u])
                isAP[u] = true;
        }
        else if (v != parent)
            low[u] = min(low[u], disc[v]);
    }
    if (parent == -1 && children > 1)
        isAP[u] = true;
}
void AP(vector<int> adj[], int V)
{
    int disc[V] = {0};
    int low[V];
    bool visited[V] = {false};
    bool isAP[V] = {false};
    int time = 0, par = -1;
    for (int u = 0; u < V; u++)
        if (!visited[u])
            APUtil(adj, u, visited, disc, low,
                   time, par, isAP);
    for (int u = 0; u < V; u++)
```

```
        if (isAP[u] == true)
            cout << u << " ";
}
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}
int main()
{
    cout << "Articulation points in first graph \n";
    int V = 5;
    vector<int> adj1[V];
    addEdge(adj1, 1, 0);
    addEdge(adj1, 0, 2);
    addEdge(adj1, 2, 1);
    addEdge(adj1, 1, 3);
    addEdge(adj1, 3, 4);
    AP(adj1, V);
    cout << "\nArticulation points in second graph \n";
    V = 4;
    vector<int> adj2[V];
    addEdge(adj2, 0, 1);
    addEdge(adj2, 1, 2);
    addEdge(adj2, 2, 3);
    AP(adj2, V);
    cout << "\nArticulation points in third graph \n";
    V = 7;
    vector<int> adj3[V];
    addEdge(adj3, 0, 1);
    addEdge(adj3, 1, 2);
    addEdge(adj3, 2, 0);
    addEdge(adj3, 2, 3);
    addEdge(adj3, 1, 4);
    addEdge(adj3, 4, 6);
    addEdge(adj3, 3, 5);
    addEdge(adj3, 4, 5);
    AP(adj3, V);
    return 0;
}
```

Output:

```
Articulation points in first graph
1 3
Articulation points in second graph
1 2
Articulation points in third graph
4
```