

**Third Year B. Tech., Sem V 20222-23**

**Design and Analysis of Algorithm Lab**

**Assignment / Journal submission**

**PRN/ Roll No: 21520010**

**Full name: Mayur Sunil Savale**

**Batch: T8**

**Assignment: Week 10**

**Title of assignment: Back Tracking**

1. Implement the following using Back Tracking

a) 4-Queen's problem

Ans:

**a. Algorithm: (Pseudocode)**

- Start from leftmost column
- If column number is greater or equal to board size then print the solution
- For every cell in column check if we can place the queen:
  - A. If we can place the queen then mark the cell
  - B. Recursively call the function for next column
  - C. If recursion return false then unmark the cell and backtrack

## b. Code snapshots of implementation

```
#include<bits/stdc++.h>
using namespace std;
bool canPlace(vector<vector<bool>>board,int r,int c)
{
    for(int i=0;i<c;i++)
    {
        if(board[r][i])
            return false;
    }
    for(int i=r,j=c;i>=0&& j>=0;i--,j--)
    {
        if(board[i][j])
            return false;
    }
    for(int i=r,j=c;i<4&& j>=0;i++,j--)
    {
        if(board[i][j])
            return false;
    }
    return true;
}
bool solve(vector<vector<bool>>board,int col)
{
    if(col>=4)
    {
        for(int i=0;i<4;i++)
        {
            for(int j=0;j<4;j++)
                cout<<board[i][j]<<" ";
            cout<<"\n";
        }
        return true;
    }
}
```

```

    for(int i=0;i<4;i++)
    {
        if(canPlace(board,i,col))
        {
            board[i][col]=true;
            if(solve(board,col+1))
                return true;
            board[i][col]=false;
        }
    }
    return false;
}

int main()
{
    vector<vector<bool>>board(4,vector<bool>(4,false));
    if(!solve(board,0))
    {
        cout<<"Solution does not exist";
        return 0;
    }
    return 0;
}

```

### Output:

```

PS C:\Users\mayur> cd "c:\Users\mayur\OneDrive\Desktop\5 th sem\DAA Assignment\Assignment-10\" ;
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
PS C:\Users\mayur\OneDrive\Desktop\5 th sem\DAA Assignment\Assignment-10>

```

**c. Complexity of proposed algorithm (Time & Space)**

- Time Complexity:  $O(4!)$
- Space Complexity:  $O(4^2)$

**d. Your comment (How your solution is optimal?)**

- This problem can be solve using naive approach of trying every configuration. But here I used backtracking so that it will reduce time complexity from  $n^n$  to  $n!.2$ .

b) 8-Queen's problem

Ans:

**a. Algorithm: (Pseudocode)**

- Start from leftmost column
- If column number is greater or equal to board size then print the solution
- For every cell in column check if we can place the queen:
  - A. If we can place the queen then mark the cell
  - B. Recursively call the function for next column
  - C. If recursion return false then unmark the cell and backtrack

**b. Code snapshots of implementation**

```
#include<bits/stdc++.h>
using namespace std;
bool canPlace(vector<vector<bool>>board,int r,int c)
{
    for(int i=0;i<c;i++)
    {
        if(board[r][i])
            return false;
    }
    for(int i=r,j=c;i>=0&& j>=0;i--,j--)
    {
        if(board[i][j])
            return false;
    }
}
```

```

    }
    for(int i=r,j=c;i<8&& j>=0;i++,j--)
    {
        if(board[i][j])
            return false;
    }
    return true;
}
bool solve(vector<vector<bool>>board,int col)
{
    if(col>=8)
    {
        for(int i=0;i<8;i++)
        {
            for(int j=0;j<8;j++)
                cout<<board[i][j]<<" ";
            cout<<"\n";
        }
        return true;
    }
    for(int i=0;i<8;i++)
    {
        if(canPlace(board,i,col))
        {
            board[i][col]=true;
            if(solve(board,col+1))
                return true;
            board[i][col]=false;
        }
    }
    return false;
}
int main()
{
    vector<vector<bool>>board(8,vector<bool>(8,false));

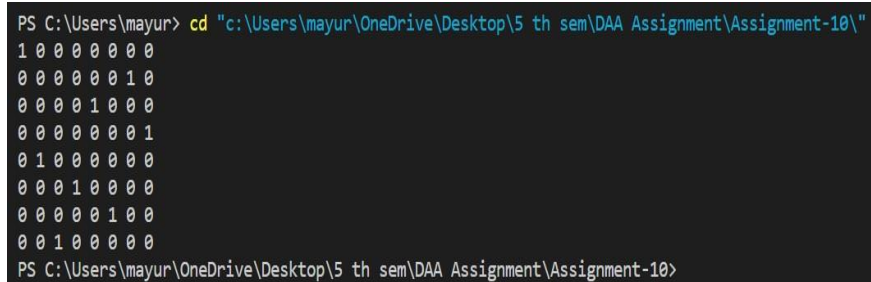
```

```

if(!solve(board,0))
{
    cout<<"Solution does not exist";
    return 0;
}
return 0;
}

```

### Output:



```

PS C:\Users\mayur> cd "c:\Users\mayur\OneDrive\Desktop\5 th sem\DAA Assignment\Assignment-10\"
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
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 1 0 0 0 0 0
PS C:\Users\mayur\OneDrive\Desktop\5 th sem\DAA Assignment\Assignment-10>

```

### c. Complexity of proposed algorithm (Time & Space)

- Time Complexity:  $O(8!)$
- Space Complexity:  $O(8^2)$

### d. Your comment (How your solution is optimal?)

- This problem can be solve using naive approach of trying every configuration. But here I used backtracking so that it will reduce time complexity from  $n^n$  to  $n!.2$ .

c) Hamiltonian Cycle

Ans:

**a. Algorithm: (Pseudocode)**

- Start from leftmost column
- If column number is greater or equal to board size then print the solution
- Traverse for adjacent nodes:
  - A. If the vertex is not visited
  - B. Then call recursively and if node doesn't lead to solution then backtrack to it and try for next node

**b. Code snapshots of implementation**

```
#include<bits/stdc++.h>
using namespace std;
class Solution
{
public:
    int n;
    bool ham(vector<vector<int>>graph,vector<bool>&vis,int
node,int&cnt)
    {
        vis[node]=true;
        cnt++;
        if(cnt==n)
            return true;
        for(auto x:graph[node])
        {
            if(!vis[x]&&ham(graph,vis,x,cnt))
                return true;
        }
        vis[node]=false;
        cnt--;
        return false;
    }
};
```

```

    }
    bool check(int N,int M,vector<vector<int>> Edges)
    {
        vector<vector<int>>graph(N+1);
        n=N;
        for(int i=0;i<Edges.size();i++)
        {
            graph[Edges[i][0]].push_back(Edges[i][1]);
            graph[Edges[i][1]].push_back(Edges[i][0]);
        }
        for(int i=1;i<=N;i++)
        {
            int cnt=0;
            vector<bool>vis(N+1,false);
            if(ham(graph,vis,i,cnt))
                return true;
        }
        return false;
    }
};

int main()
{
    int t;
    cin>>t;
    while(t--)
    {
        int N,M,X,Y;
        cin>>N>>M;
        vector<vector<int>> Edges;
        for(int i=0;i<M;i++)
        {
            cin>>X>>Y;
            Edges.push_back({X,Y});
        }
        Solution obj;
    }

```

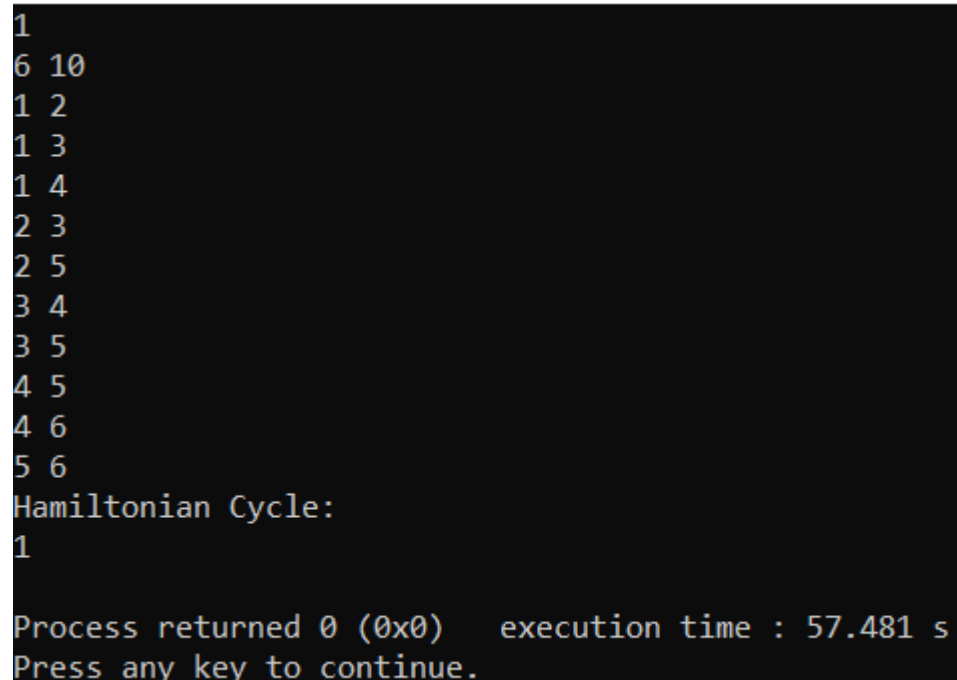


```

        cout<<"Hamiltonian Cycle:\n";
        if(obj.check(N,M,Edges))
        {
            cout<<"1"<<endl;
        }
        else
            cout<<"0"<<endl;
    }
}

```

#### Output:



```

1
6 10
1 2
1 3
1 4
2 3
2 5
3 4
3 5
4 5
4 6
5 6
Hamiltonian Cycle:
1
Process returned 0 (0x0)   execution time : 57.481 s
Press any key to continue.

```

#### c. Complexity of proposed algorithm (Time & Space)

- Time Complexity:  $O(V!)$
- Space Complexity:  $O(V)$

#### d. Your comment (How your solution is optimal?)

- Here, If any path doesn't lead to solution for that I have used backtracking so that I can check every possible path that can contain Hamiltonian cycle and for that the time complexity required is  $V$ .

#### d) Graph Coloring Problem

Ans:

##### a. Algorithm: (Pseudocode)

- By using the backtracking method, the main idea is to assign colours one by one to different vertices right from the first vertex (vertex 0).
- Before colour assignment, check if the adjacent vertices have same or different colour. By considering already assigned colours to the adjacent vertices.
- If the colour assignment does not violate any constraints, then we mark that colour as part of the result. If colour assignment is not possible then backtrack and return false.

##### b. Code snapshots of implementation

```
#include<bits/stdc++.h>
using namespace std;
class Solution
{
    public:
        int n;
        bool ham(vector<vector<int>>graph,vector<bool>&vis,int
node,int&cnt)
        {
            vis[node]=true;
            cnt++;
            if(cnt==n)
                return true;
            for(auto x:graph[node])
            {
                if(!vis[x]&&ham(graph,vis,x,cnt))
                    return true;
            }
            vis[node]=false;
            cnt--;
```

```

        return false;
    }
    bool check(int N,int M,vector<vector<int>> Edges)
    {
        vector<vector<int>>graph(N+1);
        n=N;
        for(int i=0;i<Edges.size();i++)
        {
            graph[Edges[i][0]].push_back(Edges[i][1]);
            graph[Edges[i][1]].push_back(Edges[i][0]);
        }
        for(int i=1;i<=N;i++)
        {
            int cnt=0;
            vector<bool>vis(N+1,false);
            if(ham(graph,vis,i,cnt))
                return true;
        }
        return false;
    }
};
int main()
{
    int t;
    cin>>t;
    while(t--)
    {
        int N,M,X,Y;
        cin>>N>>M;
        vector<vector<int>> Edges;
        for(int i=0;i<M;i++)
        {
            cin>>X>>Y;
            Edges.push_back({X,Y});
        }
    }
}

```

```

        Solution obj;
        cout<<"Hamiltonian Cycle:\n";
        if(obj.check(N,M,Edges))
        {
            cout<<"1"<<endl;
        }
        else
            cout<<"0"<<endl;
    }
}

```

**Output:**

```

Solution Exists: Following are the assigned colors
1 2 3 2
Process returned 0 (0x0)   execution time : 0.171 s
Press any key to continue.
_

```

**c. Complexity of proposed algorithm (Time & Space)**

- Time Complexity:  $O(V^m)$
- Space Complexity:  $O(V)$

**d. Your comment (How your solution is optimal?)**

- In this solution I have checked all possible configuration for a particular node with all colours by using backtracking. As I have checked all the configuration the time complexity is  $O(V^m)$ .

## 2. Implement following problem using graph traversal Technique

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

Ans:

### a. Algorithm: (Pseudocode)

- Make a queue for BFS traversal.
- For every iteration assign every child opposite colour.
- If any children are already coloured and have same colour as parent then return false.
- Traverse until queue become empty and return true.
- Traverse this for every uncoloured vertex as graph can be disconnected

### b. Code snapshots of implementation

```
#include<bits/stdc++.h>
using namespace std;
class Solution
{
public:
    bool solve(queue<int>q,vector<int>&color,int
    flg,vector<int>adj[])
    {
        while(!q.empty())
        {
            int k=q.size();
            for(int i=0;i<k;i++)
            {
                int x=q.front();
                q.pop();
                for(auto m:adj[x])
                {
                    if(color[m]==-1)
                    {
```

```

        color[m]=flg;
        q.push(m);
    }
    else if(color[m]!=flg)
        return false;
    }
}
flg= (flg==1)? 0:1;
}
return true;
}
bool isBipartite(int V, vector<int>adj[])
{
    // Code here
    vector<int>color(V,-1);
    for(int i=0;i<V;i++)
    {
        if(color[i]==-1)
        {
            queue<int>q;
            q.push(i);
            color[i]=0;
            int flg=1;
            if(!solve(q,color,flg,adj))
                return false;
        }
    }
    return true;
}
};
int main()
{
    int t;
    cin>>t;
    while(t--)

```

```

{
    int N,M,X,Y;
    cin>>N>>M;
    vector<int> Edges[N+1];
    for(int i=0;i<M;i++)
    {
        cin>>X>>Y;
        Edges[X].push_back(Y);
        Edges[Y].push_back(X);
    }
    Solution obj;
    if(obj.isBipartite(N,Edges))
    {
        cout<<"1"<<endl;
    }
    else
        cout<<"0"<<endl;
}
}

```

### Output:

```

1
6 10
1 2
1 3
1 4
2 3
2 5
3 4
3 5
4 5
4 6
5 6
It is not Bipartite Graph

Process returned 0 (0x0)   execution time : 24.022 s
Press any key to continue.

```

**c. Complexity of proposed algorithm (Time & Space)**

- Time Complexity:  $O(V+E)$
- Space Complexity:  $O(V)$

**d. Your comment (How your solution is optimal?)**

- If I have used DFS for the traversal then iteration would have increased which results in increasing time complexity. So, using BFS and graph colouring concept I have reduced the time complexity to  $O(V+E)$ .

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

Ans:

**a. Algorithm: (Pseudocode)**

- Pick an arbitrary vertex of the graph root and run DFS from it. We have these points to notice:
- Let's say we are in the DFS, looking through the edges starting from vertex  $v \neq \text{root}$ . If the current edge  $(v, to)$  is such that none of the vertices  $to$  or its descendants in the DFS traversal tree has a back-edge to any of ancestors of  $v$ , then  $v$  is an articulation point. Otherwise,  $v$  is not an articulation point. Let's consider the remaining case of  $v = \text{root}$ .
- This vertex will be the point of articulation if and only if this vertex has more than one child in the DFS tree. Now we have to learn to check this fact for each vertex efficiently. We'll use "time of entry into node" computed by the depth first search. So, let  $tin[v]$  denote entry time for node  $v$ .
- We introduce an array  $low[v]$  which will let us check the fact for each vertex  $v$ .  $low[v]$  is the minimum of  $tin[v]$ , the entry times  $tin[p]$  for each node  $p$  that is connected to node  $v$  via a back- edge  $(v, p)$  and the values of  $low[to]$  for each vertex  $to$  which is a direct descendant of  $v$  in the DFS tree:
- $low[v] = \min(tin[v], tin[p], low[to])$  for all  $p$  for which  $(v, p)$  is a back edge for all  $to$  for which  $(v, to)$  is a tree edge



- Now, there is a back edge from vertex  $v$  or one of its descendants to one of its ancestors if and only if vertex  $v$  has a child  $to$  for which  $low[to] < tin[v]$ .
- If  $low[to] = tin[v]$  the back edge comes directly to  $v$ , otherwise it comes to one of the ancestors of  $v$ . Thus, the vertex  $v$  in the DFS tree is an articulation point if and only if  $low[to] \geq tin[v]$ .

## b. Code snapshots of implementation

```
#include<bits/stdc++.h>
using namespace std;
void solve( vector<int>adj[],int node,int
prt,vector<int>disc,vector<int>low,vector<bool>&vis,vector<int>&
ans,int &t)
{
    vis[node]=true;
    int child=0;
    low[node]=disc[node]=++t;
    for(auto x:adj[node])
    {
        if(!vis[x])
        {
            child++;
            solve(adj,x,node,disc,low,vis,ans,t);
            low[node]=min(low[node],low[x]);
            if(prt!=-1&&low[x]>=disc[node])
            {
                ans.push_back(node);
                //cout<<node<<" ";
            }
        }
        else if(x!=prt)
            low[node]=min(low[node],disc[x]);
    }
}
```

```

    }
    if(prt==-1&&child>1)
        ans.push_back(node);
}
vector<int> articulationPoints(int V, vector<int>adj[])
{
    // Code here
    vector<int>disc(V,0);
    vector<int>low(V,0);
    vector<bool>vis(V,false);
    vector<int>ans;
    int t=0;
    for(int i=0;i<V;i++)
    {
        if(!vis[i])
        {
            solve(adj,i,-1,disc,low,vis,ans,t);
        }
    }
    sort(ans.begin(),ans.end());
    return ans;
}
int main()
{
    int t;
    cin>>t;
    while(t--)
    {
        int N,M,X,Y;
        cin>>N>>M;
        vector<int> Edges[N+1];
        for(int i=0;i<M;i++)
        {
            cin>>X>>Y;
            Edges[X].push_back(Y);

```

```

        Edges[Y].push_back(X);
    }
    vector<int>ans=articulationPoints(N,Edges);
    cout<<"Articulation points: ";
    for(int i=0;i<ans.size();i++)
        cout<<ans[i]<<" ";
    cout<<"\n";
    if(ans.size()==0)
    {
        cout<<"Graph is not biconnected!!"<<endl;
    }
    else
        cout<<"Graph is biconnected!!"<<endl;
    }
}

```

### Output:

```
1
6 10
1 2
1 3
1 4
2 3
2 5
3 4
3 5
4 5
4 6
5 6
Articulation points: 5
Graph is biconnected!!

Process returned 0 (0x0)   execution time : 25.752 s
Press any key to continue.
```

#### c. Complexity of proposed algorithm (Time & Space)

- Time Complexity:  $O(V+E)$
- Space Complexity:  $O(V)$

#### d. Your comment (How your solution is optimal?)

- To find articulation point I have to check the back edges which can be possible using DFS traversal and that is the time complexity is  $O(V+E)$ .