Graph Concepts that I Forget Frequently

md.mottakin.chowdhury

September 2018

1 Definitions

Chromatic Number: Minimum number of colors required to color graph G such that no two adjacent vertex gets same color.

Brooks' Therorem: For any connected undirected graph G with maximum degree x, the chromatic number of G is at most x unless G is a complete graph or an odd cycle, in which case the chromatic number is $x + 1$.

Independent Vertex Set: Set of vertices in a graph G , no two of which are adjacent that means no two vertices in this set is connected by an edge.
Maximum Independence Set is the such set with maximum size.

Vertex Cover: Set of Vertices in a graph G, such that each edge in G incident on at least one vertex in the Set.
Minimum Vertex Cover is such a cover of minimum size.

Note that,

- A set of vertices is a vertex cover if and only if its complement is an independent set.
- The number of vertices of graph G is equal to sum of minimum vertex cover and maximum independent set.

Dominating Set: A dominating set for a graph $G = (V, E)$ is a subset D of V such that every vertex not in D is adjacent to at least one member of D. The domination number is the number of vertices in a smallest dominating set for G.

Any *maximal independent set* in a graph is necessarily also a minimal dominating set. Note that, *maximal independent set* is any independent set such that adding any other vertex to this set is not possible as an edge is created. But *maximum independent set* is such set of maximum size.

Clique: A clique of a graph G is a complete subgraph of G, and the clique of largest possible size is referred to as a maximum clique.

Suppose we are asked to find the *maximum clique* of a bipartite graph. We take the compliment graph. Then find the *maximum independent set* by finding the *maximum matching*. We know *maximum independent set cardinality* is equal to the difference between the number of vertices of the graph and the maximum bipartite matching. This value is the *maximum clique* of our given bipartite graph.

Existence of Euler Path: The existence of Eulerian paths and circuits depends on the degrees of the nodes. First, an undirected graph has an Eulerian path exactly when all the edges belong to the same connected component and

- the degree of each node is even or
- the degree of exactly two nodes is odd, and the degree of all other nodes is even.

In the first case, each Eulerian path is also an Eulerian circuit. In the second case, the odd-degree nodes are the starting and ending nodes of an Eulerian path which is not an Eulerian circuit.

In a directed graph, we focus on indegrees and outdegrees of the nodes. A directed graph contains an Eulerian path exactly when all the edges belong to the same strongly connected component and

- in each node, the indegree equals the outdegree or
- in one node, the indegree is one larger than the outdegree, in another node, the outdegree is one larger than the indegree, and in all other nodes, the indegree equals the outdegree.

In the first case, each Eulerian path is also an Eulerian circuit, and in the second case, the graph contains an Eulerian path that begins at the node whose outdegree is larger and ends at the node whose indegree is larger.

Existence of Hamiltonian Path

No efficient method is known for testing if a graph contains a Hamiltonian path, and the problem is NP-hard. Still, in some special cases, we can be certain that a graph contains a Hamiltonian path.

A simple observation is that if the graph is complete, i.e., there is an edge between all pairs of nodes, it also contains a Hamiltonian path. Also stronger results have been achieved:

- **Dirac's theorem:** If the degree of each node is at least $n/2$, the graph contains a Hamiltonian path.
- **Ore's theorem:** If the sum of degrees of each non-adjacent pair of nodes is at least n, the graph contains a Hamiltonian path.

2 Hierholzer's Algorithm

Hierholzer's algorithm is an efficient method for constructing an Eulerian circuit. The algorithm consists of several rounds, each of which adds new edges to the circuit. Of course, we assume that the graph contains an Eulerian circuit; otherwise Hierholzer's algorithm cannot find it.

First, the algorithm constructs a circuit that contains some (not necessarily all) of the edges of the graph. After this, the algorithm extends the circuit step by step by adding subcircuits to it. The process continues until all edges have been added to the circuit.

The algorithm extends the circuit by always finding a node x that belongs to the circuit but has an outgoing edge that is not included in the circuit. The algorithm constructs a new path from node x that only contains edges that are not yet in the circuit. Sooner or later, the path will return to node x , which creates a subcircuit.

If the graph only contains an Eulerian path, we can still use Hierholzer's algorithm to find it by adding an extra edge to the graph and removing the edge after the circuit has been constructed. For example, in an undirected graph, we add the extra edge between the two odd-degree nodes.

```
// Caution: Code from GforG.
// A C++ program to print Eulerian circuit in given
// directed graph using Hierholzer algorithm
#include <bits/stdc++.h>
using namespace std;

void printCircuit(vector< vector<int> > adj)
{
    // adj represents the adjacency list of
    // the directed graph
    // edge_count represents the number of edges
    // emerging from a vertex
    unordered_map<int,int> edge_count;

    for (int i=0; i<adj.size(); i++)
    {
        //find the count of edges to keep track
        //of unused edges
```

```
        edge_count[i] = adj[i].size();
    }

    if (!adj.size())
        return; //empty graph

    // Maintain a stack to keep vertices
    stack<int> curr_path;

    // vector to store final circuit
    vector<int> circuit;

    // start from any vertex
    curr_path.push(0);
    int curr_v = 0; // Current vertex

    while (!curr_path.empty())
    {
        // If there's remaining edge
        if (edge_count[curr_v])
        {
            // Push the vertex
            curr_path.push(curr_v);

            // Find the next vertex using an edge
            int next_v = adj[curr_v].back();

            // and remove that edge
            edge_count[curr_v]--;
            adj[curr_v].pop_back();

            // Move to next vertex
            curr_v = next_v;
        }

        // back-track to find remaining circuit
        else
        {
            circuit.push_back(curr_v);

            // Back-tracking
            curr_v = curr_path.top();
            curr_path.pop();
        }
    }

    // we've got the circuit, now print it in reverse
    for (int i=circuit.size()-1; i>=0; i--)
    {
        cout << circuit[i];
        if (i)
           cout<<" -> ";
    }
}
```