ESO207 Programming Assignment-3

Mohan Krishna (200590) and Suyash Srivastava (201031)

Hackerrank ID: sumokrisri

10^{th} November, 2021

1 Pseudo Code for Bipartite(G):

Algorithm 1: dfs(G, i, flag, assign)

Data :

1. Graph $G(V, E)$ in adjacency list representation: for all j A[j] represents the adjacency list of vertex j.
2. $flag$: if it becomes zero at any point then graph is not bipartite.
3. vertex i where we currently are in the depth first search
4. '*assign*' which represents the partite set to which i should be added.

Result:

- If i has not yet been assigned a Partite Set, adds i to the set represented by *assign*.
- If i is already in some Partite Set P, checks if P is the same as *assign*.
- Traverses the graph using Depth First Search.
- Assigns sets to all the (previously unassigned) vertices which have a path to vertex i.
- If at any time bipartite property is violated, it sets $flag$ to 0 and returns $flag$.

```
Function dfs(G, flag, i, assign):
    if set[i] == 0 then
        /* i hasn't been visited yet                                      */
        set[i] ← assign
    end
    if set[i] ≠ assign then
        /* has been visited earlier and bipartite property (elaborated upon in
           section 2) is being violated                                   */
        flag ← 0; return flag
    end
    if set[i] == 1 then
        neighbour ← 2
    end
    else
        neighbour ← 1
    end
    for all vertices u in adjacency list of i do
        if set[u] == 0 then
            /* u hasn't been visited yet                                  */
            dfs (G, flag, u, neigbour)
        end
        if set[u] ≠ neighbour then
            /* u has been visited earlier, but is in same set as i        */
            flag ← 0; return flag
        end
    end
```

Algorithm 2: Bipartite(G)

Data :

Graph $G(V,E)$ in adjacency list representation: $A[1..n]$ represents the array of vertices

Result:

If graph is bipartite Returns (V_1, V_2) where (V_1, V_2) is a partition of V such that all edges of G are between V_1 and V_2.

If graph is not bipartite Returns -1.

```
Function Bipartite(G):
    flag ← 1
    /* A graph with less than two vertices cannot be bipartite */
    if number of vertices is less than 2 then
        return -1
    end
    /* Set is an array which stores the set of each vertex.  i corresponds to
       the vertex in V. */
    for i=0 and i<n do
        /* assign 0 to all vertices */
        set[i] ← 0; i ← i + 1
    end
    for i=0 and i<n do
        /* Assign sets such that all the neighbours of a vertex are in
           opposite set of the vertex i.e.  no two adjacent vertices are of
           same set if we arrive at a contradiction at any point, we stop.
           Assume the two sets are 1 and 2 */
        if set[i]==0 then
            /* i hasn't been assigned set yet */
            dfs(G,flag,i,1)
        end
        if flag == 0 then
            return -1
        end
    end
    V1 = V2 = {}
    for i = 0 and i<n do
        if set[i] == 1 then
            V1 ← V1 ∪ {i}
        end
        else
            V2 ← V2 ∪ {i}
        end
    end
```

2 Uniqueness of partition of vertices in a Bipartite Graph

In a Bipartite Graph, the vertices can be partitioned into 2 non-empty set V_1 and V_2 such that all the edges cross the cut (V_1, V_2) (That is each edge of G has one endpoint in V_1 and other endpoint in V_2).
Let u be in V_1, and let v be a neighbour of u
$\Rightarrow e = (u, v)$ is an edge in G
$\Rightarrow e$ crosses the cut (V_1, V_2)
$\Rightarrow v \in V_2$ Thus all neighbours of a vertex in V_1 are in V_2. Similarly, all neighbours of a vertex in V_2 are in V_1.

We will use this observation to prove the following theorem:

Theorem: If a bipartite graph is **connected**, then the partition of vertices is unique (apart from interchanging the partite sets).

Let G be a connected bipartite graph. Let P be a partition (V_1,V_2) and let P' be the partition (V_2,V_1). Then for all partitions $Q = (U_1, U_2)$ of G, $Q = P$ or $Q = P'$.

Proof:
Let $v_0 \in V_1$, and suppose without loss of generality that $v_0 \in U_1$.
A similar proof follows if $v_0 \in U_2$, with P' in place of P in the proof below.**Observation 1**
$v_0 \in V_1 \Rightarrow v_0 \notin V_2 \Rightarrow (v_0 \notin V_2 \cap U_1)$
$v_0 \in U_1 \Rightarrow v_0 \notin U_2 \Rightarrow (v_0 \notin V_1 \cap U_2)$
$(v_0 \notin V_2 \cap U_1) \wedge (v_0 \notin V_1 \cap U_2)$

To prove the theorem, it suffices to show that $Q = P$.

We prove the theorem by contradiction. Suppose, to the contrary, that there exists a vertex $v_n \in G$ such that $v_n \in V_i$ and $e \in U_{3-i}$.
Because the component is connected, there exists a path from vertex v_0 to v_n; let us fix one such path: $v_0 - v_1 - v_2 - ... - v_{n-1} - v_n$. Because alternate vertices in a path are neighbours and hence belong to different sets in a partition, we can say that: $v_n \in V_i \cap U_{3-i}$
$\iff v_{n-1} \in V_{3-i} \cap U_i$
$\iff v_{n-2} \in V_i \cap U_{3-i}$
...
If n is even, we get, $v_n \in V_i \cap U_{3-i} \iff v_0 \in V_i \cap U_{3-i}$, which is a contradiction by **Observation 1**.
If n is odd, we get, $v_n \in V_i \cap U_{3-i} \iff v_0 \in V_{3-i} \cap U_i$, which is again a contradiction by **Observation 1**.

Corollary: If there are n connected components of a graph, then the total number of possible partitions is 2^n.
Proof: Let the n connected components of the graph $G = (N, E)$ be $G_i = (N_i, E_i)($ for $i \in \{1, 2, ..., n\}$.
Observation a: For every partition P on G there corresponds a unique tuple($P_1, P_2, ..., P_n$) where $P_i = (A_i, B_i)$ is a partition of G_i for each i in$\{1, 2, ..., n\}$
If $P = (V_1, V_2)$ is a partition of G, then $P_i = (A_i, B_i) = (V_1 \cap N_i, V_2 \cap N_i)$ is a partition of N_i.
Observation b: For every tuple $(P_1, P_2, ..., P_n)$ where $P_i = (A_i, B_i)$ is a partition of G_i for each i in$\{1, 2, ..., n\}$ there corresponds a unique partition P on G given by P = (V_1, V_2) where,
(V_1, V_2)
$= (\bigcup_{i=1}^{n} A_i, \bigcup_{i=1}^{n} B_i)$
The fact that P is a partition on G, can be seen by the fact that if $e = (x, y)$ is an edge of G,
$\Rightarrow e$ is an edge of G_i for some i (because there is no edge between G_i and G_j for $i \neq j$,
$\Rightarrow e$ crosses the cut (A_i, B_i) on G_i,
$\Rightarrow x \in A_i$ and $y \in B_i$,
$\Rightarrow x \in V_1$ and $y \in V_2$,
$\Rightarrow e$ crosses the cut (V_1, V_2) on P.

Thus there is a bijection between the sets M and N, where $M = \{P : P$ is a partition of $G\}$ and $N = \{(P_1, P_2, ..., P_n) : P_i$ is a partition of G_i for each i in$\{1, 2, ..., n\}\}$. Thus,

$$|M| = |N| = \prod_{i=1}^{n} |P_i| = \prod_{i=1}^{n} 2 = 2^n$$

Where |.| represents the cardinality of the set.

3 Complexity Analysis of Algorithm Bipartite

We perform the analysis for the worst case, which is when the graph is Bipartite.

Algorithm Bipartite:

The for loop which initializes set of all the vertices (assigns set 0 to all vertices) takes $O(|V|)$, where $|V|$ is the number of vertices in the graph.

The **for** loop(inside which the function `dfs` is called whenever the set for the vertex hasn't been assigned) is of $O(|V|)$ in the worst case (which is when flag is never zero) , excluding the time taken by the `dfs` function.

Algorithm dfs:

We note that for a vertex v which belongs to the graph, `dfs` is called exactly once. This is because the vertex is assigned set only once, and once a set has been assigned, it cannot be unassigned. If a vertex has been assigned a set, then dfs for that vertex isn't called.
And every vertex in the graph is visited at least once. This is ensured by the main for-loop in algorithm `Bipartite`.

A vertex is said to be the neighbour of another vertex if the shortest path connecting the two vertices contains only one edge.
For every vertex, the number of times the dfs function is called, is equal to the number of neighbours of that vertex. This is same as the degree of that vertex. Hence dfs for each vertex takes **O(degree(v))** time.

The sum of the degrees for all the vertices is equal to twice the number of edges, because each edge has exactly two vertices on it(say a and b); and each edge is counted twice(once in the degree of a and once in that of b); while calculating the sum of all degrees.

Hence the number of times the `dfs` function is called over the whole program is equal to $|2E|$, where E is the number of edges.

Total time taken by the program: $t = O(|V|) + \sum_{v \in V} O(degree(v))$
$=c_1 \times |V| + \sum_{v \in V}(c_2 \times degree(v) + c_3)$
$=c_1 \times |V| + c_3 \times |V| + c_2 \times \sum_{v \in V} degree(v)$
$=(c_1 + c_3) \times |V| + c_2 \times (2|E|)$
$=\mathrm{O}(|V| + |E|)$

Hence the total time is $\mathbf{O}(|V| + |E|)$