ESO207A Programming Assignment-3

Aditya Tanwar
200057

Akhil Agrawal
200076

November 2021

Q1 (Marks 20 + 5 + 25) An undirected graph $G(V, E)$ is said to be bipartite if V can be partitioned into two sets V_1, V_2 such that all edges of G are between sets V_1 and V_2 (That is, each edge of G has one endpoint in V_1 and other endpoint in V_2).

More mathematically, there exists non-empty and disjoint sets V_1 and V_2 s.t. $V = V_1 \cup V_2$ and $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$.

(a) You are given an undirected connected graph $G(V, E)$ in adjacency list representation. Write pseudo-code $Bipartite(G)$, which answers if G is bipartite or not. If G is bipartite, it 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. Your algorithm should work in $\mathcal{O}(|V| + |E|)$ time.

(b) In part (a), if G is bipartite then is the partition of vertices unique? What best can you say about it. What if G is not connected?

(a) *pseudo-code*

- **G(V, E)**- G, as an object, is assumed to contain V, E, and $adjacencylist$ within itself, and they are hence accessed in functions.
- Notations like V and $G.V$, for example, are used interchangeably in the pseudo-code for functions.
- V_1 and V_2 are initialised as empty sets. A simple array could be used as the underlying data structure, just as long as $\mathcal{O}(1)$ time is taken for both insertion and access.
- Array implementation of a stack is assumed, (and implemented in actual code), as it ensures constant time *peek*, *pop* and *push* operations.
- The distances assigned in each connected component are from an arbitrary vertex in the same connected component. Distances of are **not** calculated from the same vertex, for vertices in different connected components.

Algorithm 1: Bipartite(G)

```
Data: A graph G(V, E) where V is set of Vertices and E is set of Edges
Result: Returns the partition (V1, V2) if G is bipartite and false otherwise.
n ← V.size                                            //Number of vertices in G
let distance[n]        //Array storing min. distance from a particular vertex
let visited[n]                //Array storing whether a node is visited or not
for all v ∈ V do
  distance[v] ← ∞                                             //Initialization
  visited[v] ← false
end for
for all v ∈ V do
  if visited[v] == false then
    distance[v] = 0              //Executing dfs on every connected component
    dfs(G, v, distance, visited)
  end if
end for
for all v ∈ V do
  for all u ∈ adjacencyList(v) do
    if |distance[v] − distance[u]| ≠ 1 then
      return false
    end if
  end for
end for
V1 ← ∅, V2 ← ∅                                   //The required partition of V
for all v ∈ V do
  if distance[v] ≡ 1 mod 2 then
    V2.insert(v)
  else
    V1.insert(v)
  end if
end for
return (V1, V2)
```

Algorithm 2: dfs(G, x, $distance$, $visited$)

Data: A graph $G(V,\, E)$, a vertex $x \in V$, and arrays with the same purpose as that in Bipartite

Result: Fills the *distance* and *visited* arrays correctly in the connected component containing x

```
let stack_dfs                                    //Stack to manage vertices
visited[x] = true
stack_dfs.push(x)
while not stack_dfs.isEmpty() do
  u ← stack_dfs.peek()
  stack_dfs.pop()
  for all v ∈ G.adjacencyList(u) do
    if visited[v] == false then
      visited[v] = true                          //Ensures that each vertex is pushed
      stack_dfs.push(v)                          //into the stack exactly once
    end if
    if distance[v] > distance[u] + 1 then
      distance[v] = distance[u] + 1              //Assigns min distance to node v
    end if
  end for
end while
```

Complexity analysis

- **dfs**- Executes "depth-first search" on the connected component. Let us call the component C_i, and let the set containing vertices in C_i be denoted by V_{C_i}, and the set of edges in C_i be E_{C_i}. Each vertex $v \in V_{C_i}$ is pushed to the stack exactly once and further, each edge is inspected exactly twice. So, its run-time is-

$$
\begin{aligned}
&a_1 + a_2 \cdot |V_{C_i}| + 2a_3 \cdot |E_{C_i}| \\
&= \mathcal{O}(|V_{C_i}| + |E_{C_i}|)
\end{aligned}
$$

- **Bipartite**- These are broadly the steps in the algorithm-
  - ⋆ Initialization- Initializes arrays of length $|V|$. $\mathcal{O}(|V|)$
  - ⋆ dfs- Executes dfs on each connected component, but as the graph is given to be connected, $V_{C_i} = V$ and $E_{C_i} = E$, and so, the runtime is- $\mathcal{O}(|V| + |E|)$
  - ⋆ Check if the graph is bipartite- We traverse through each edge in the graph exactly twice to check if the distances of the vertices are of opposite parity (i.e., odd and even). $\mathcal{O}(|E|)$
  - ⋆ Assignment of partition- Each vertex is assigned a set in constant time. $\mathcal{O}(|V|)$

Combining these results, we conclude that Bipartite works in $\mathcal{O}(|V| + |E|)$ time for a connected graph. The runtime for Bipartite in disconnected graph remains the same, and is elaborated in **(b)** part.

(b) We assume the graph is bipartite. If the graph is not connected, it has say $\mathcal{N}(>1)$ separate connected components. Now, for each connected component we can uniquely partition it such that it is bipartite. But to partition the entire graph, we can combine the partitions of $\mathcal{N}$ separate connected components in any order. Hence we will have a total of $2^{(\mathcal{N}-1)}$ unordered partitions of the graph possible. If the given graph is fully connected, then the partition of the graph will be unique.
Thus, the (unordered) partition is unique only if the whole graph is connected, else there are $2^{(\mathcal{N}-1)}$ ways possible to partition it.
The run-time of Bipartite for a disconnected graph is as follows-

- Initialization- Same as that for a connected graph. $\mathcal{O}(|V|)$
- dfs - Executed on each connected component of the graph. $\mathcal{O}(|V_{C_i}| + |E_{C_i}|)$
- Check if the graph is bipartite by traversing through each edge twice. $\mathcal{O}(|E|)$
- Partition Assignment- Each vertex is assigned a set in constant time. $\mathcal{O}(|V|)$

Thus, combining these results we get-

$$= c_0 \cdot |V| + \sum_{i=1}^{\mathcal{N}} c_1 \cdot (|V_{C_i}| + |E_{C_i}|) + c_2 \cdot |E| + c_3 \cdot |V|$$

$$\texttt{Using } \sum_{i=1}^{\mathcal{N}} |V_{C_i}| = |V| \;\texttt{\&}\; \sum_{i=1}^{\mathcal{N}} |E_{C_i}| = |E|$$

$$= c_0 \cdot |V| + c_1 \cdot (|V| + |E|) + c_2 \cdot |E| + c_3 \cdot |V|$$
$$= \mathcal{O}(|V| + |E|)$$

<u>Note</u> $\sum_{i=1}^{\mathcal{N}} |V_{C_i}| = |V|$ as each V_{C_i} is pair-wise disjoint with any other V_{C_j} by definition of a disconnected graph. Similarly, for $\sum_{i=1}^{\mathcal{N}} |E_{C_i}| = |E|$.

□□□