ESO207 Programming Assignment 3

Devansh Kumar Jha(200318) and Divyansh Gupta(200351)

2021–11-8

1 Solution to Part A

1.1 Data Structure Usage

The graph is represented in **Adjacency List Representation**. At the core the implementation is run fully through structures and pointers only. For data abstraction and easy handling it is arranged inside a class **graph** and uses structure **vertex**.

1.2 Strategy Used

It is assumed that the graph is bipartite and the following algorithm is run. If there is a contradiction within the working of the algorithm it will indicate that our initial assumption of a bipartite graph is wrong and the graph is not actually bipartite.

- The first unvisited vertex encountered is assumed to be in the partition V_1 of graph.
- All the vertices of the graph G which are directly connected to the current vertex are made a part of the partition opposite to that of current vertex if they are not already visited.
- If an already visited vertex is encountered then it is checked for consistency with the changes intended by the above step. If inconsistent then our initial assumption of bipartite graph is wrong and the program will exit here otherwise no changes are done.
- If the situation in step 3 does not arise then we perform the 2nd step for all un-visited vertices directly connected to the current vertex.
- After completing step 4 for all connected vertices go back to step 1.

By this strategy the non-connected vertices or components of the graph are automatically assigned to the partition V_1 and then continued however these vertices are actually floating and they can be kept in any of the two partitions giving rise to multiple possible partitions.

1.3 Structure Used

The structure **vertex** used is defined as under -

Algorithm 1: Structure Declaration

```
struct vertex {
        int data;
        struct vertex* next;
        struct vertex* prev;
}
```

1.4 Pseudo Codes

- **Bipartite(G)**
  G denotes the object of class graph. This returns the sets V_1 and V_2 which are the partition for the set V of vertices of graph $G(V, E)$ in case G is bipartite otherwise returns a NULL.

- **dfs(G,visited,part,i,last)**
  Here G is a graph,visited and part are arrays of size $\|V\|$ and i and last are the numbers denoting index of the current vertex and the previous vertex being worked upon.This is the main working function which implements most of the working of the algorithm as discussed in "Strategy Used" header. It returns a boolean value which is **false** if the graph is bipartite till the vertices examined or **true** otherwise.

Algorithm 2: Bipartite(G)

Input : A graph G.

Output: Two arrays representing sets V_1 and V_2.

```
visited ← FALSE boolean array of size ‖V‖
part ← 0 integer array of size ‖V‖
while v in G do
    if visited[v] == FALSE then
        flag ← dfs(G, visited, part, i, −1)
        /* -1 in place of last denotes first unvisited vertex.    */
        if flag == TRUE then
            return NULL
        else
        end if
    else
    end if
end while
V1 , V2 ← empty number array
len1 , len2 ← 0
while v in G do
    if part[v] == 1 then
        V1[len1] = v
        len1 = len1 + 1
    else
        V2[len2] = v
        len2 = len2 + 1
    end if
end while
return (V1, V2)
```

Algorithm 3: dfs(G,visited,part,i,last)

Input : A graph G, sets of size $\|V\|$ for visited and part and index number of the current and last vertex. last=-1 if it is the first unvisited vertex.

Output: A boolean value representing whether any contradiction in the initial assumption of bipartite graph is found.

```
visited[i] = TRUE
if last == -1 then
    part[i] = 1
else
    if part[i]! = 0 then
        if part[i]! = part[last] then
            return FALSE
        else
            return TRUE
        end if
    else
        if part[last] == 1 then
            part[i] = 2
        else
            part[i] = 1
        end if
    end if
end if
temp ← G.get(i)
/* get() function on graph gives the linked list containing all
   directly connected vertices to index i.                        */
while v in temp do
    if v → num! = last then
        x = dfs(G, visited, part, v → num, i)
        if x == TRUE then
            return TRUE
        else
        end if
    else
    end if
end while
return FALSE
```

1.5 Runtime Analysis

1.5.1 Intuition

The graph is stored in **Adjacency List** so the size of the storage is $O(\|V\| + \|E\|)$. The algorithm runs over all the vertices and edges once. So it could be intuitively seen that the time complexity for the algorithm would be same as the size of the adjacency list representation.

1.5.2 Detailed Explanation

- **dfs(G,visited,part,i)**
  The dfs() function executes in **O(1) time** for a single vertex and then it calls recursively all the vertices on edges passing through the current vertex. This call is continued till either all the unvisited reachable vertices from the current position are traversed or entire graph is completed. So let us assume V_i vertices are connected to the current vertex then surely $\|V_i\| <= \|V\|$. Also the function takes **O(1) time** for travelling on any edge. So let the edges to be traversed are E_i. So $\|E_i\| <= \|E\|$.
  Runtime Complexity = $O(\|V_i\| + \|E_i\|)$

- **Bipartite(G)**
  Initializing visited, part, V_1 and V_2 all take $O(\|V\|)time$. As the dfs() function changes the visited and part arrays directly, evidently all the vertices and edges of the graph are traversed only once. Also the last while loop for putting values into V_1 and V_2 takes $O(\|V\|)time$ so -
  $TotalIteration <= constant * (4 * \|V\| + \|V\| + \|E\| + \|V\|)$
  Runtime Complexity = $O(\|V\| + \|E\|)$

2 Solution to Part B

Answer to this part is given assuming the graph $G(V, E)$ to be **Bipartite**.

2.1 Answer

If the graph G is connected then the partitions created for this graph will be unique however for a un-connected graph it will not be unique.

2.2 Explanantion

According to the algorithm it is easy to see that when a graph is connected, as soon as we assume one of its vertices to be the part of a partition all other vertices will have to join a partition accordingly and the fate for each vertex will be well defined. Even if we reverse the assumption for first node it will result only in the shuffling of partitions.
However, in case of un-connected graph, as soon as we encounter a vertex which could not be reached by any of the previous vertices, there generates a possibility for keeping this to anyone of the partitions and thus the final sets formed would not be unique.