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 this implementation is executed through structures and pointers. 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

```
1 struct vertex {
2         int num;
3         struct vertex* next;
4         struct vertex* prev;
5 }
```

1.4 Pseudo Codes

- **Bipartite(G)**
  G denotes the object of class graph. This returns an array which contains the set number for each vertex of graph $G(V, E)$ in case it 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: 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.

```
1  visited[i] = TRUE
2  if last == −1 then
3  |  part[i] = 1
4  else
5  |  if part[last] == 1 then
6  |  |  part[i] = 2
7  |  else
8  |  |  part[i] = 1
9  |  end if
10 end if
11 temp ← G.get(i)
12 /* get() function on graph gives the linked list containing all
      directly connected vertices to index i.                    */
13 while v in temp do
14 |  if visited[v] == TRUE then
15 |  |  if part[i] == part[v] then
16 |  |  |  return TRUE
17 |  |  else
18 |  |  end if
19 |  else
20 |  |  x ← dfs(G, visited, part, v, i)
21 |  |  if x == TRUE then
22 |  |  |  return TRUE
23 |  |  else
24 |  |  end if
25 |  end if
26 end while
27 return FALSE
```

Algorithm 3: Bipartite(G)

Input : A graph G.
Output: An array representing set of each vertex.

```
1  visited ← FALSE boolean array of size ||V||
2  part ← 0 integer array of size ||V||
3  if ||V|| < 2 then
4  |  return NULL
5  else
6  end if
7  /* If the total number of vertices are less than 2 then atleast
      one of the partitions will have to be empty which is not
      allowed.                                                    */
8  while v in G do
9  |  if visited[v] == FALSE then
10 |  |  flag ← dfs(G, visited, part, i, −1)
11 |  |  /* -1 in place of last denotes first unvisited vertex.   */
12 |  |  if flag == TRUE then
13 |  |  |  return NULL
14 |  |  else
15 |  |  end if
16 |  else
17 |  end if
18 end while
19 return part
```

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)**
  All the steps in dfs() function executes in **O(1) time** for a single vertex except the recursive calls. The recursive call is continued till either all the unvisited reachable vertices from the current position are traversed or entire graph is completed. So it takes **O(deg(v)) time** for a particular vertex v.
  Runtime Complexity = $O(deg(v))$

- **Bipartite(G)**
  Initializing visited and part takes $O(\|V\|)time$. As the dfs() function changes the visited and part arrays directly, all the vertices of the graph are traversed only once so -
  $TotalIteration <= constant * (2 * \|V\| + \sum_{n=1}^{\|V\|} deg(v_n))$
  Also by definition and pictorial representation of the graph -
  $\sum_{n=1}^{\|V\|} deg(v_n) = 2 * \|E\|$
  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 non-connected graph multiple partitions are feasible.

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 vertex it will result only in the shuffling of partitions.
However, if the graph is not connected then 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.