ESO207A - Programming assignment-3

Group members:
Arnav Gupta(200186)
Kushagra Sharma(200539)

Bipartition

We are given an undirected connected graph $G(V, E)$ in adjacency list representation. Now we need to check the graph for bipartition. If G is bipartite, we should return $(V1,\ V2)$ where $(V1,\ V2)$ is a partition of V such that all the edges of G are between $V1\ and\ V2$.

Implementation

We firstly claim that iff the graph G(V, E) is bipartite, it is possible to colour all the vertices in V in 2 colours such that the colour of each vertex is different from its neighbours. A neighbour of a vertex V_i is any vertex in V that has an edge to V_i. Such a graph is also said to be 2-colorable.

Proof -Let G be a 2-colorable graph, i.e. it is possible to colour all vertices such that no 2 adjacent vertices(i.e., endpoints of edges) have the same colour. Let V1 represent the vertices coloured with the 1st colour and V2 be the remaining ones. Then by definition, it is possible to colour the graph such that no two endpoints of edges have the same colour. From this statement, we can say that all edges will have one vertex in V1 and the other in V2. This implies G is bipartite. Similarly, the converse can also be shown to be true.

In our code and pseudocode, we have represented these two colours as 1 and 2. This way, the colour of a vertex will correspond to its subset in V, which is denoted by V1 and V2.

Breadth First Search(BFS):

We take an arbitrary node and assign it to V1(colour 1), and with that as reference, we implement Breadth-First Search(BFS), across all the nodes in the graph, assigning the colours(colour of successor is different from the predecessor). If we can successfully assign the colours to all the nodes without overlap(overlap occurs if the colour of the successor has already been assigned and is the same as its predecessor), then we can conclude that the graph is bipartite.

Implementation: We implement BFS using a self-made class Queue. (as taught in the lectures)

Functions implemented in class Queue. Each function takes constant time:

- Push(x) - Adds an element x to the end of the queue
- Pop() - Removes the first element from the queue
- Empty() - Returns TRUE if the queue is empty, else FALSE
- Front() - Return the first value of the queue.

Functions in Linked-List:
A circular doubly-linked list is implemented in the class Linked-List.

- Pushback(x) - Adds a node with element x as its data to the rear of the linked list.

The constructor of the class initializes the linked list as a sentinel node.

The pseudocode explains how each part of the main function proceeds:
There are 'V' nodes and 'E' edges. (Taken in input)
Initialising Adjacency List:

```
LinkedList adj[v]; //Array of v doubly linked-lists, Implemented as a class.
for(int i=0;i<E;i++) do //There are E edges.
        Input: a , b - There exists an edge between node a and b
        adj[a].pushBack(b);
        adj[b].pushBack(a);
```

//Adjacency List adj[i] stores all the neighbours of node 'i'.

Initialising Colour Array:

```
Int col[v];
for(i=0,i<V;i++) do //There are V vertices that need to be coloured
        col[v]= -1; //colour -1 indicates that colour to the vertex is not assigned.
```

//Colour Array stores the colour of the vertex (1 or 2)

Initialising self-implemented class Queue:

```
Queue q;
q.push(0); //Adds the node 0 to the queue.
col[0]=1; //Initialise colour of the first node as 1(asked in the question), it can be arbitrary.
```

Colouring nodes with BFS:

```
Int valid=1; //valid=0 would indicate the graph is not bipartite.
while(!q.empty()) do: loop breaks when the queue is empty
        Int temp = q.front(); // initialises a variable temp with value as the first member of 'q'
        q.pop(); //Removes the first element from the queue
        Node* ptr = adj[temp].head->next; // initialises ptr as the first neighbour of temp.
        while(ptr!=adj[temp].head) do: //if ptr==adj[temp].head it would mean that we
have traversed all the neighbours and reached back to the head.
                Int x = ptr->data; //Gets the value of node stored in ptr(node number of neighbour)
                if(col[x] uncoloured) do: //if colour of node 'x' is not assigned, assign the appropriate
colour
                        col[x]=3-col[temp];
        //endpoints of an edge(x and t) should have a different colour({1:2},{2:1})
                else do:
                        if(col[x]==col[temp]) do:
        // x and temp have already been assigned the same colour, so the graph is not bipartite.
```

```
                    int valid=0; //change value of valid to 0
                    Break;
            ptr=ptr->next;//Go to the next neighbour of node 'temp'
```

Output:

```
if(valid==1) do: //Graph is bipartite
        print('YES')
        for(int i=0;i<V;i++) do:
                print(col[i]); //Printing the colour assigned to each node
else do: //Graph is not bipartite
        print('NO')
```

Time Complexity Analysis:

- All defined Queue and Linked-List operations take constant time - **O(1)**

- Time Complexity to initialize Adjacency List : **O(E)** - We iterate the loop E times to get all the E edges. Adding a node to the list, in each iteration, takes constant time.

- Time Complexity to initialize Colour Array: **O(V)** - We iterate the loop V times. In each iteration, initialising the colour of the node to -1 (col[i]= -1) takes constant time.

- Time Complexity of Breadth-First Search: **O(V+E)**
  Each node is pushed (and popped)into the queue once.This is because we only push the node when it has not been visited by the BFS. Once it has been visited, we colour it and This is an O(V) operation.
  Since it is an undirected graph, over all the iterations of Queue, we traverse each edge twice (in the adjacency list). This is because each edge has 2 endpoints and since we visit each vertex once, the edges will each be visited twice. This is O(E).
  Inside the loops, the remaining operations take constant time. Thus the overall time complexity of this step is O(V+E).

- Printing Output: **O(V)** - Worst case scenario, we would have to iterate over all the elements of the Colour Array, giving the time complexity O(V).

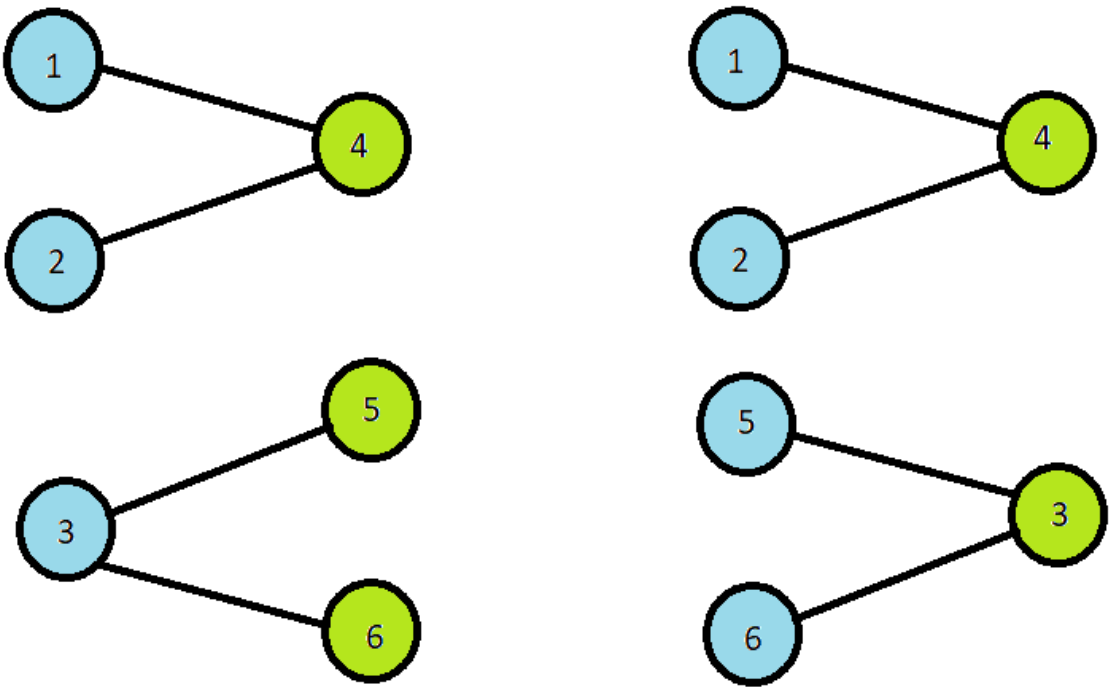
Hence the overall time complexity of the algorithm is **O(V+E)**, where V is the number of vertices and E is the number of edges in the graph.

b)

(i) If G is connected and bipartite, then the bipartition will be unique. We start by selecting any vertex and proceeding from there. All its neighbours will have to be kept in the different subsets using the conditions of a bipartite graph. Hence, we can see that at each stage the parity of the current vertex will determine the parity of its neighbours. In other words, once we fix the colour of an arbitrary vertex, all other vertices would have no choice but to be coloured in that colour which keeps the graph bipartite. This verifies our claim.

(ii) If G is not connected and bipartite, then the bipartition is guaranteed to be non-unique . We firstly apply the claim of (i) which shows that each connected component will have a unique bipartition. (If G is bipartite then all its components will be bipartite too). For simplicity, let G(V,E) consist of two connected components (Va,Ea) and (Vb,Eb). Then, it is easy to see that colours of vertices in Vb can be switched and the graph will still be bipartite. The illustration below shows this point :

In the graph below, Va = {1,2,4} and Vb = {3,5,6}



This claim can be extended to when G has >2 connected components, which verifies the answer.