**CPS 483/583 – Homework 2**

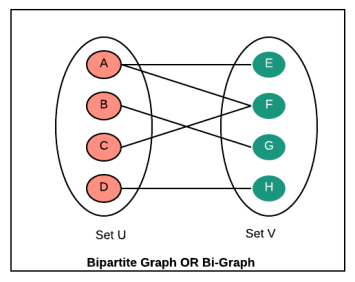
**Due:** 5 PM, Friday, 19 February, 2021

* Recall that when we say “given (di)graph *G*”, we mean the adjacency lists of *G* are given to work with.
* For vertices *x* and *y*, [*x,y*] denotes an (undirected) edge between *x* and *y* and (*x,y*) refers to the directed edge *x* → *y*.
* In a weighted graph, *weight/length of a path* is the sum of weights of edges on the path. The distance between vertices *x* and *y* is the weight of a shortest path between *x* and *y*.

1. (20 pts) Given an *arbitrary* connected graph *G*, use the discussion on bipartite graphs along with breadth first search, to present an O(*m*+*n*) time algorithm for the following: if *G* is bipartite, output *bipartite* and also find a proper/valid 2-coloring of *G*: for a vertex *v* record the color assigned to vertex *v* (1 or 2) in *v*.color.

If *G* is not bipartite, output *NOT bipartite* and as a certificate for it, the algorithm should output vertices that form a cycle of odd length in the cyclic order.

You must provide the entire algorithm for the problem.



*Algorithm to check if a graph is Bipartite:*

One approach is to check whether the graph is 2-colorable or not usi   
Following is a simple algorithm to find out whether a given graph is Birpartite or not using Breadth First Search (BFS).   
1. Assign RED color to the source vertex (putting into set U).

2. Color all the neighbors with BLUE color (putting into set V).

3. Color all neighbor’s neighbor with RED color (putting into set U).

4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where m = 2.

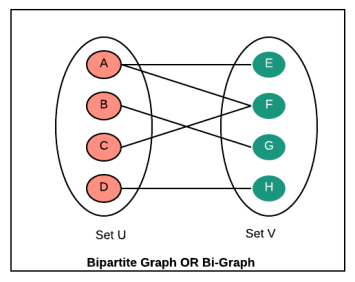
5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

Actual Code

|  |
| --- |
| #include <iostream>  #include <vector>  #include <queue>  using namespace std;    // Data structure to store a graph edge  struct Edge {      int src, dest;  };    // A class to represent a graph object  class Graph  {  public:      // a vector of vectors to represent an adjacency list      vector<vector<int>> adjList;        // Graph Constructor      Graph(vector<Edge> const &edges, int N)      {          // resize the vector to hold `N` elements of type `vector<int>`          adjList.resize(N);            // add edges to the undirected graph          for (auto &edge: edges)          {              adjList[edge.src].push\_back(edge.dest);              adjList[edge.dest].push\_back(edge.src);          }      }  };    // Perform BFS on the graph starting from vertex `v`  bool BFS(Graph const &graph, int v, int N)  {      // to keep track of whether a vertex is discovered or not      vector<bool> discovered(N);        // stores the level of each vertex in BFS      vector<int> level(N);        // mark the source vertex as discovered and      // set its level to 0      discovered[v] = true, level[v] = 0;        // create a queue to do BFS and enqueue      // source vertex in it      queue<int> q;      q.push(v);        // loop till queue is empty      while (!q.empty())      {          // dequeue front node          v = q.front();          q.pop();            // do for every edge `v —> u`          for (int u: graph.adjList[v])          {              // if vertex `u` is explored for the first time              if (!discovered[u])              {                  // mark it as discovered                  discovered[u] = true;                    // set level as the level of parent node plus 1                  level[u] = level[v] + 1;                    // enqueue vertex                  q.push(u);              }              // if the vertex has already been discovered and the              // level of vertex `u` and `v` are the same, then the              // graph contains an odd-cycle and is not bipartite              else if (level[v] == level[u]) {                  return false;              }          }      }        return true;  }    int main()  {      // vector of graph edges as per the above diagram      vector<Edge> edges = {          {1, 2}, {2, 3}, {2, 8}, {3, 4}, {4, 6}, {5, 7},          {5, 9}, {8, 9}          // if we add edge `2 —> 4`, the graph becomes non-bipartite      };        // total number of nodes in the graph      int N = 10;        // build a graph from the given edges      Graph graph(edges, N);        // Perform BFS traversal starting from vertex 1      if (BFS(graph, 1, N)) {          cout << "The graph is bipartite";      } else {          cout << "The graph is not bipartite";      }        return 0;  } |

1. (20 pts) The problem has to do with the following: given graph *G* and vertex *s*, find a shortest path from *s* to each of the other vertices.

We learnt that breadth first search can solve the problem when every edge has a weight/length of 1.



The Bipartite Graph can only work in an idealistic form

(a) Consult above for the definition of distance between vertices in a weighted graph.

Suppose you are given a weighted graph *G* in which weight of every edge is one of 1, 2, 3, 4, or 5, and vertex *s*. Present a small example to show that the simple breadth first search from *s* is *not* guaranteed to find the shortest paths from *s* to other vertices.

Clearly present the graph with vertex *s* labeled, and the outcome of a breadth first search from *s*.

**Read the following before attempting (b) through (e)**

Suppose we have a weighted graph *G* in which weight of an edge is an integer from 1 ··· 5 (i.e. it is one of 1, 2, 3, 4, or 5).

Consider the following construction to build graph *H* based on graph *G*: corresponding to every vertex of *G*, *H* also has a vertex. We will also introduce additional vertices in *H* as explained below.

Every edge in *H* will have a weight of 1.

Whenever *G* has an edge *x*−*y* with weight *k*, *instead* we will introduce in *H* a path connecting *x* and *y* with *k* edges such that each of the intermediate vertices on the path is new, and also every such vertex has degree exactly 2. Observe that we have to add *k* − 1 such new vertices.

For example, if edge *x* − *y* in *G* had a weight of 4 in *G*, in *H* we will instead have the path *x* − 0 − 0 − 0 − *y* connecting *x* and *y*. Each of the 0 vertices is new and degree of such a vertex in *H* is exactly 2.

Clearly, this means, if edge *x* − *y* has a weight of 1 in *G*, then in *H* we will also have the edge *x* − *y*.

1. Let *G* = (*V,E*) where *V* = {1*,*2*,*3*,*4} and *E* = {[1*,*2]*,*[2*,*3]*,*[3*,*4]*,*[1*,*3]*,*[1*,*4]}. Further, weight of [1*,*2] is 1, weight of [2*,*3] is 3, weight of [3*,*4] is 5, weight of [1*,*3] is 5, and weight of [1*,*4] is 2.

Present *G* (along with weights of edges) and *H* (constructed from *G* as above).

1. What is the distance between vertices 1 and 3 in *G*? What is the distance between vertices 1 and 3 in *H*?
2. Given a graph *G* in which weight of every edge is an integer from 1 ··· 5, describe how breadth first search can still be used to find shortest paths in *G* from *s* to other vertices; use observations from (b) and (c).

You do not have to write pseudocode. If you clearly list the steps, that is fine.

1. Recall that *m* is the number of edges and *n* is the number of vertices in the input graph *G*. Analyze your algorithm from (d) to establish its complexity in terms of *m* and *n*. Explain your analysis and outcome.