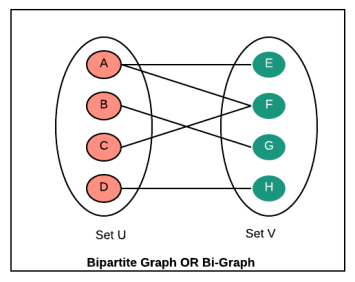
**CPS 483/583 – Homework 2**

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

1. (20 pts)

You must provide the entire algorithm for the problem.



The Bipartite algorithm for testing a graph is:

One way is to look at whether the graph has 2 colors or not

The following is the algorithm

1. Assign a color to the vertex of the source .

2. Color all the neighbors in any other color of your liking (by setting it in V).

3. Color the whole neighbor

4. In this way, color in all the vertices to satisfy all the problems of the calculation problem in the form of m when m = 2.

5. When rendering the vertex by color- if we find a neighboring vertex sharing the same color as the current vertex then the graph cannot be coded in the same color so can be stated as not bipartite

Actual Code

|  |
| --- |
| # enter <iostream>  # include <vector>  # include <ueue>  using namespace std;    Edge Edge {  int src, end;  };    Class graph  {  community:  vector <vector <int>> adjList;    Graph (vector <Edge> const & edges, int N)  {  adjList.resize (N);    for (auto & edge: edges)  {  adjList [Edge.src] .push\_back (Edge.dest);  adjList [edge.dest] .push\_back (Edge.src);  }  }  };    // Create BFS on the graph from vertex `v`  bool BFS (Graph const and graph, int v, int N)  {    vector <bool> found (N);      vector <int> level (N);    found [v] = true, level [v] = 0;    line <int> q;  q. push (v);    // loop until the line is empty  while (! q.empty ())  {  v = q forward ();  q.pop ();      of (int u: graph.adjList [v])  {  if (! get [u])  {  found [u] = true;    level [u] = level [v] + 1;    // insert vertex  q. push (u);  }  The // vertex level `u` and` v` are the same, and then  // graph contains irregular cycles and is not bipartite  otherwise if (level [v] == level [u]) {  return false;  }  }  }    true return;  }    int main ()  {  vector <Edge> edges = {  {1, 2}, {2, 3}, {2, 8}, {3, 4}, {4, 6}, {5, 7},  {5, 9}, {8, 9}  // if we add to the edge `2 -> 4`, the graph becomes non-bipartite  };    // the total number of nodes in the graph  int N = 6;    // construct a graph from the given edges  Graph graph (edge, N);    // Make BFS crossings from vertex 1  if (BFS (graph, 1, N)) {  cout << "Graphite bipartite";  } more {  cout << "Graph is not bipartite";  }    return 0;  } |

1. (20 pts)

The Bipartite Graph can only work in an idealistic form

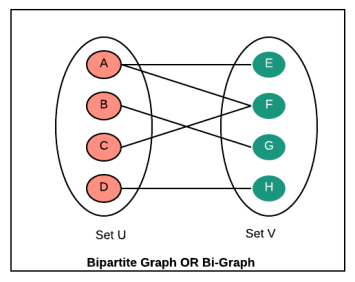
Actual Code

|  |
| --- |
| #include <iostream>  # include <vector>  # include <ueue>  using namespace std;    // Data structure to maintain the edge of the graph  Edge Edge {  int src, end;  };    // The section that will represent the graph object  Class graph  {  community:  // vector of vectors to represent the list of adjacency  vector <vector <int>> adjList;    // Graph constructor  Graph (vector <Edge> const & edges, int N)  {  // vector size to hold `N-type objects` vector <int>`  adjList.resize (N);    // add edges to the non-target graph  for (auto & edge: edges)  {  adjList [Edge.src] .push\_back (Edge.dest);  adjList [edge.dest] .push\_back (Edge.src);  }  }  };    // Create BFS on the graph from vertex `v`  bool BFS (Graph const and graph, int v, int N)  {  // tracking whether vertex was detected or not  vector <bool> found (N);    // maintains the level of each vertex in BFS  vector <int> level (N);    // mark the vertex of the source as found again  // set its value to 0  found [v] = true, level [v] = 0;    // create a BFS generating line and enter a line  // vertex source on it  line <int> q;  q. push (v);    // loop until the line is empty  while (! q.empty ())  {  // create a front docket  v = q forward ();  q.pop ();    // do in all edges `v -> u`  of (int u: graph.adjList [v])  {  // if vertex `u` is first checked  if (! get [u])  {  // mark as found  found [u] = true;    // set the level as the parent node level and 1  level [u] = level [v] + 1;    // insert vertex  q. push (u);  }  // if vertex already found and file for  The // vertex level `u` and` v` are the same, and then  // graph contains irregular cycles and is not bipartite  otherwise if (level [v] == level [u]) {  return false;  }  }  }    true return;  }    int main ()  {  // vector of graph edges according to the diagram above  vector <Edge> edges = {  {1, 2}, {2, 3}, {2, 8}, {3, 4}, {4, 6}, {5, 7},  {5, 9}, {8, 9}  // if we add to the edge `2 -> 4`, the graph becomes non-bipartite  };    // the total number of nodes in the graph  int N = 6;    // construct a graph from the given edges  Graph graph (edge, N);    // Make BFS crossings from vertex 1  if (BFS (graph, 1, N)) {  cout << "Graphite bipartite";  } more {  cout << "Graph is not bipartite";  }    return 0;  } |

(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*.



Following the breadth first search algorithm that follows certain rules

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer-wise or through certain levels thus exploring the neighboring nodes and then to the next node levels .

As the name of the BFS suggests, you need to limit the width of the graph as follows:

1. First go horizontally or access the layer from the left to the right as layered out

2. Move on to the next layer

However, there is no weighting factor for vertex s to help our transversal with the neighbor nodes.

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

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).

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).

For every single graph vertex the following interconnection holds

1. What is the distance between vertices 1 and 3 in *G*? What is the distance between vertices 1 and 3 in *H*?

Vertex Formation

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.

Vertices in

Further, weight of [1*,*2] is 1,

weight of [2*,*3] is 3,

weight of [3*,*4] is 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.

Distance in G in Half the Distance in H based on the proof that

Proof=> 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.

• Provide distance to a minimum number of edges

• Previous vertex. The previous vertex of the source is a special value, such as null, which indicates that it has no predecessor.

For example, here is an unadjusted graph with five vertices, numbered 0 to 5,

- with vertex numbers appearing above or below the vertices. Within each vertex there are two numbers: the distance from the source, followed by the previous one on the shortest path from the source.

In BFS, we initially set the distance and precursor of each vertex to a special value (null).

We start searching for the source and give you a 0 rating.

After that we visit all the neighbors of the source and give each neighbor a grade 1 and set the predecessor to be the source.

We then visit all the neighbors of the 1st distant vertices and who have never been visited before, and give each of these vertices a 2nd grade and set the precedent to be the vertex from which we come.

We continue until all the corners are located from the vertex of the visited source, we always visit all the vertices away kkk from the source before visiting any vertex at a distance of k + 1k + 1k, plus, 1.

Being complex

BFS time difficulty is O (V + E),

where V - number of nodes and

E- end number.

Therefore, the following is the structure of Complextiy graph

O (m + n)