Project 3: Maximum Bipartite Matching

Project start: Friday, October 30
Early submission: Monday, November 9, 11:57pm
Project due date: Monday, November 16, 11:57pm

Purpose: Explore several important graph algorithms

Input: An undirected graph by adjacencies
Output: A maximum matching if the graph is bipartite, an error message if it is not

Please read the description very carefully. You will stand to lose significant amount of points if you do not follow instructions perfectly in your projects, we will not be lenient!

Project skeleton

The project skeleton consists of the following files:

- Graph related: Graph.java dependencies In.java, StdIn.java, StdOut.java, Bag.java, Stack.java
- FlowNetwork related: FlowNetwork.java, FlowEdge.java , FordFulkerson.java
- Project3.java contains the empty methods that you need to implement.

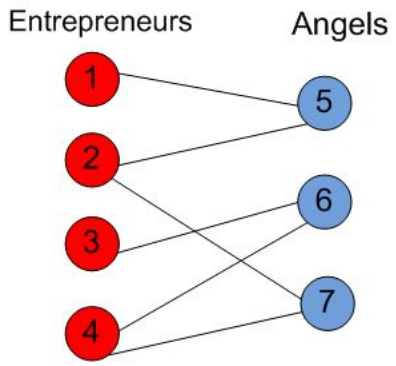
Description

You are working for service that matches budding entrepreneurs with angel financing. The company logo is TGIF (Thanks to George it'd financed).[1] It is getting time for the annual mega match-up event, where an entrepreneur meets an investor, in person. The matching is done using a data base that records interest compatibility. That is, the project pitched by the entrepreneur has to be of interest to the investor. Your job is to identify as many compatible match-ups as possible and to send invitations.

As a matter of due diligence, you check that for each potential match there is one investor and one entrepreneur. Matching two angels would be a mistake, as would matching two entrepreneurs. Also, an angel takes on one project at a time.

The data you are to work with comes as a graph. The vertices of the graph are entrepreneurs and investors. There is an edge between vertices v and w if the proprietary algorithm of TGIF has determined that v and w are compatible and can be expected to have a productive meeting at the event. Vertex 1 is always an entrepreneur.

[1] George was the fabled founder of TGIF. Later he used his accumulated wealth to start a restaurant chain.



Compatibility Graph.

This information comes as the following input:

Line 1: n
n gives the number of vertices, numbered 1 to n

Lines 2 to n+1: p : v_1, v_2, …, v_k
the adjacencies of vertex p to vertices v_j where 1<=j<=k.
k depends on the vertex p of course.

Note that the adjacencies are not necessarily sorted. Also, the graph must have at least two vertices and one edge.

None of the lines contain blanks. The following is an example:

```
3
1:2,3
2:3,1
3:2,1
```

A vertex has at least one incident edge. Execute the following tasks:

Task 1 – Read the graph, test correctness, build the graph

1. createGraph: reads user input and constructs an adjacency list representation of an undirected graph G. You should check that the input format is observed, i.e., no embedded blanks and correct separators.
   The empty graph can be input, and is specified by a single line containing just 0. After reading it, print "Warning: empty graph" and stop further processing.
2. validateGraph: checks that the input is a correct, undirected graph and is done as follows:
   a. All vertex numbers are between 1 and n.
      If not, you print "Error: vertex number out of range"
   b. For the adjacency p : … q … there should also be the adjacency q : … p … If not, then you must add the missing adjacency. No error message is issued.
   c. For the adjacency list p : v_1, v_2, …, v_k all adjacent v_j are unique and are different from p. If not, you print "Error: illegal adjacency"

d. The adjacency lists will have arbitrary order in the input, but you should sort each list to be in ascending order if it is not yet.
e. Check that the graph is connected by performing a DFS. Start the DFS with vertex 1. Process adjacencies in order. If the graph is not connected, print an error message "Error: graph is not connected".

If the graph has fewer than 10 vertices, you are to print out the graph in input format. No leading blanks. No blanks on either side of the colon or comma. All adjacencies are shown, including those you filled in step (b). If the graph has 10 or more vertices you print the message "Graph passes"

If an error message is issued you quit; no further processing takes place.

Task 2 – Check that the graph is bipartite and identify the partition

In this task you will implement the following method inside project3.java:
checkBipartite: checks that the graph is bipartite using BFS. Using BFS is mandatory. Start the BFS with vertex 1. Process the adjacencies in sorted order.

If the graph is not bipartite, you print "Error: graph is not bipartite". In that case, all processing stops.

If the graph is bipartite and has fewer than 10 vertices, you print out the vertex partition in two lines. The first line contains the partition, called X, containing vertex 1, the second line contains the other partition, called Y. The format is no leading blanks, the list items are comma-separated, each comma followed by one blank.

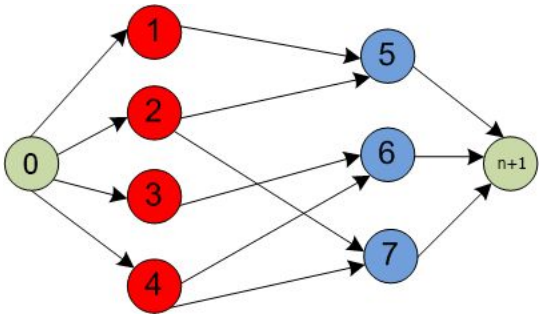
If the graph is bipartite and has 10 or more vertices, print the message "Graph is bipartite"

Task 3 – Build the flow network

In this task, you will implement createFlowNetwork inside project3.java that receives an undirected graph as parameter and creates a directed flow network graph where all edges go from the vertex partition X (containing 1) to the other vertex partition Y (see Task 2). The easiest way to do this is to delete all adjacencies lists of the vertices in the Y partition.

Having so obtained a directed graph G', you now augment the graph as follows:

a) Add source vertex 0, and, for every vertex v in X, add the directed edge (0,v) from the source vertex to v in X.
b) Add the sink vertex numbered n+1. For every vertex w in Y, add the directed edge (w,n+1).
c) There are no other edges to be added.



The sink vertex is vertex 8.

If the original graph has less than 20 vertices, print the flow network in the exact same format as you did before in Task 1. Otherwise print the message "Flow network built".

Task 4 -- Run the Ford-Fulkerson max flow algorithm provided

In this task you will use the Ford-Fulkerson algorithm (see Text, chapter 16) that finds the maximum flow and the subset of edges (u,v), $u \in X$, $v \in Y$ with non-zero flow. Build the representation that the Ford-Fulkerson algorithm expects.

Every network edge should have capacity 1, so that an edge between X and Y has a nonzero flow, then it is an edge of the corresponding bipartite matching. You may add the class members and a constructor as needed.

Run the Ford-Fulkerson algorithm to determine the max flow. Note that an edge with nonzero flow that is not incident to source or sink vertex is a matching edge. There should be as many matching edges as there is flow.

Modify the Ford-Fulkerson code such that it does not print the network unless the original graph has less than 20 vertices.

Task 5 -- Print the bipartite matching

As provided, the Ford-Fulkerson algorithm determines both maxFlow and minCut. Thereafter, you will print the matching found. Each edge in the matching is printed out as (u, v), where u is in partition X and v in partition Y. Source and sink edges are not printed. Print the edges by ascending order of X. Edges are separated by comma. As before, a comma is followed by a single blank.

Print a separate line "Matching found" preceding the output of the matching edges. Following the line with the matching edges print "Number of edges N", where N is the number of matching edges.

Notes

Task 1: You check the obvious things. You can leverage your code from Project 0 to read the input. Checking the absence of repeated edges and self loop edges (v,v) can be done by sorting the adjacency lists.

Task 2: This can be done using BFS or DFS. However, you are required to use DFS and color the vertices red or blue by whether the distance from the root (i.e., the vertex depth) is odd or even. Check that the back edges never go blue-to-blue nor red-to-red. If the graph is not connected or not bipartite issue an error message to that effect.

The root of your DFS must be the vertex numbered 1. Vertex 1 is going to be red/ entrepreneur, and is in Partition X.

Task 3: Let X be the set of red vertices and Y the set of blue vertices. Recall that vertex 1 is red and therefore in X. Convert the graph into a directed graph where all edges go from X to Y. Note that this entails dropping all adjacencies of vertices in Y. Add two additional vertices, s and t,

numbered 0 and n+1, respectively. There is a directed edge from vertex s to every vertex in X, and a directed edge from every vertex in Y to vertex t. There are no other edges.

Task 4: Assume that every edge has the capacity 1 when running the Ford-Fulkerson algorithm. After finding the maximum flow, the subset of edges (u,v), where u is in partition X and v in Partition Y, with non-zero flow is an edge in the corresponding bipartite matching.

Task 5: Output the edges (u,v), where u is in X and v is in Y, on a single line, sorted by the vertex u, comma-separated.

Example Files

The following files create various error messages by Task 2, graph check.

test_data_1.txt (Because it is not a bipartite graph)

The following file is bipartite and should find a max flow of value 4.

test_data_2.txt

The following file is a big bipartite with 500 vertices, you should be able to find a max flow of value 250.

test_data_3.txt

Submission Instruction:

You must create a directory “project3”. You must Put all your .java files under “project3” directly. If you are using Eclipse or other IDE, make sure you put .java files under project3 directly. Failing to do so will result in 10 points off.

Your file structure should looks like the following:

```
project3:
  |- all java files(No extra folder like "src")
```

Note:

1. you should not literally copy the following commands.
2. Replace yourLogin with your login ID like “zhan1015”.

To resubmit, you just need to retype the following commands and type “yes” after the third command.

```
ssh yourLogin@data.cs.purdue.edu
cd directoryContainsProject3
turnin -v -c cs251 -p project3 project3
```

After the third command is executed, the system will give you some feedback about which files and folders have been submitted. If you resubmit a project, the previously submitted files will be overwritten.