Project 3: Maximum Bipartite Matching

Project start: Thursday, October 29

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.

Entrepreneurs   Angels

31
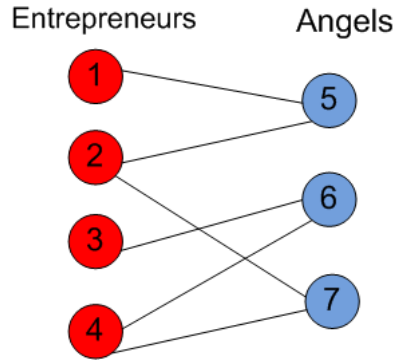
32   Compatibility Graph.

33 This information comes as the following input:

34 <u>Line 1:</u>   n
35   n gives the number of vertices, numbered 1 to n

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

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

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

42   3
43   1:2,3
44   2:3,1
45   3:2,1

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

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

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

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

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

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

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

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

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

74   If the graph is bipartite and has fewer than 10 vertices, you print out the vertex partition in two lines.
75   The first line contains the partition, called X, containing vertex 1, the second line contains the other
76   partition, called Y. The the list items are comma-separated.

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

78   Task 3 – Build the flow network

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

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

84   a) Add source vertex 0, and, for every vertex v in X, add the directed edge (0,v) from the source
85   vertex to v in X.
86   b) Add the sink vertex numbered n+1. For every vertex w in Y, add the directed edge (w,n+1).
87   c) There are no other edges to be added.
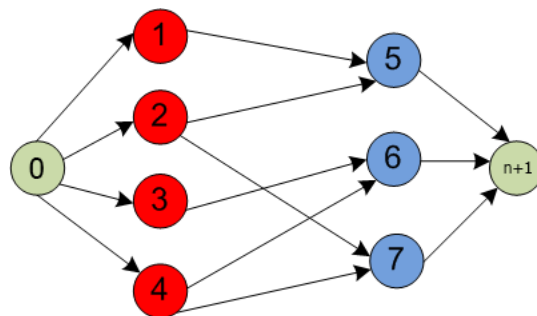
88
89   The sink vertex is vertex 8.

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

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

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

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

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

Task 5 -- Print the bipartite matching

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

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

Notes

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

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

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

119 Task 3: Let X be the set of red vertices and Y the set of blue vertices. Recall that vertex 1 is red and
120   therefore in X. Convert the graph into a directed graph where all edges go from X to Y. Note
121   that this entails dropping all adjacencies of vertices in Y. Add two additional vertices, s and t,
122   numbered 0 and n+1, respectively. There is a directed edge from vertex s to every vertex in X,
123   and a directed edge from every vertex in Y to vertex t. There are no other edges.

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

127   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,
128   comma-separated.

129 **Example Files**

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

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

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

133 test_data_2.txt

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

135 test_data_3.txt

136 **Submission Instruction:**

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

140 Your file structure should looks like the following:

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

143 **Note:**

144   1. you should not literally copy the following commands.

145   2. Replace yourLogin with your login ID like "zhan1015".

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

147 ssh yourLogin@data.cs.purdue.edu
148 cd directoryContainsProject3
149 turnin -v -c cs251 -p project3 project3
150

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

154