Project 3 for CSE 332 (Spring 2011)   v.1.0 (Monday, May 16)

Social Connections and Parallelism

Phase A due Wednesday, May 25 at 11:00 PM via Catalyst CollectIt.
Phase B due Tuesday,   June 1 at 11:00 PM via Catalyst CollectIt.

CSE 332: Data Abstractions
The University of Washington, Seattle, Spring 2011

Overview: This project is about graph data structures and algorithms, including parallelism with the Fork-Join framework. In Phase A, you'll implement Kruskal's algorithm for finding a minimum spanning tree. In Phase B, you'll implement a friend-recommendation method for a social network. If you do the extra-credit option, you'll also design and implement a "Friend-of-the-Week" service.

Purposes: (a) gain implementation experience with graph algorithms, (b) get acquainted with the UNION-FIND ADT (c) implement a priority queue as a binary heap, (d) work with the fork-join parallelism framework, (e) handle concurrent access to shared data, (f) explore how graph algorithms can be used to solve social-network problems, and (g) gain experience measuring speedup due to parallelism.

Resources Provided: You'll start with the following resources:
1. The Visual Stack Applet and associated files from Project 1.
2. A "Visual Graph Applet" that handles the construction and display of both directed and undirected graphs. This will allow you to more readily see what your graph algorithms are doing and thus detect and fix errors.

Required Functionality for Phase A: You will implement Kruskal's algorithm for finding a minimum spanning tree in an undirected graph. You'll implement a command MST that invokes Kruskal's algorithm. As parts of your implementation, you'll implement the UNION-FIND ADT and you'll also implement a priority queue as a binary heap.

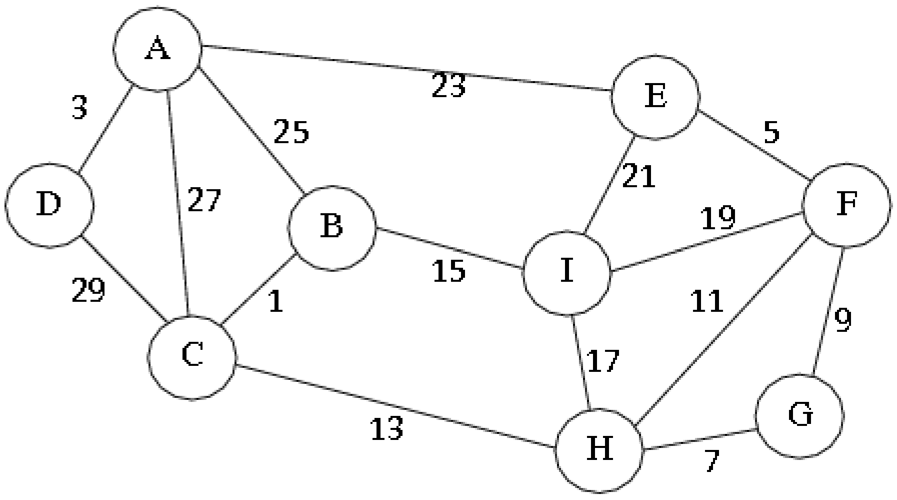
Phase B Functionality: You'll implement a friend-suggestion algorithm for a social network that is represented as an undirected graph without edge-cost information. You'll parallelize this implementation using the Fork-Join parallelism framework. If you do the extra credit options, then you'll also design and implement a "Friend-of-the-Week" service.

Recommended Development Sequence: The following steps are recommended.

1. Download the starter code: VisualGraphApplet.java. Create a project VisualGraphApplet with the starter code (and VisualDataStructureApplet.java from the earlier projects). Compile and run. Get familiar with starter code: especially the representation of the graph.

2. Create a new project VisualKruskalsMST, creating a corresponding Java file

VisualKruskalsMST.java that starts out as a copy of VisualGraphApplet.java. Modify the preloaded command sequence so that it constructs the first test example KT1. This graph is illustrated below. Choose X and Y coordinates for each vertex so that the graph, when displayed in your applet, looks like the illustration.



3. (a) Assign one partner to implement the UNION-FIND ADT (the basic method described on p295 and following pages in Weiss is fine). (b) Assign another partner to implement priority queues using binary heaps. (c) Both partners (or a third partner) implement Kruskal's algorithm for computing minimum spanning trees, making use of the UNION-FIND ADT and the priority queue implementations. Implement code to handle a command MST that will run Kruskal's algorithm on the current graph. If there is no numeric cost label for an edge, the algorithm should assume that the cost is 1. If the current graph is a directed graph, then MST should report: "Minimum Spanning Trees are not defined for directed graphs."

4. Make the algorithm (a) highlight the edges of the MST as they are tried; unhighlight them if they are rejected. At the end of Kruskal's algorithm, all MST edges should be highlighted. (b) Write to the history window: (ii) the number of edges tried, (ii) the number of comparisons performed by the binary heap, (iii) the number of links traversed by the UNION-FIND implementation (assuming you used "up-trees"), and (iv) the cost of the final MST.

5. Comment your source code. At the beginning of the file, explain who did what, how you implemented the UNION-FIND ADT (up-trees, arrays, etc), and any special features you added to the program.

6. Create a JAR file, and post your applet on the web. Turn in your source code and an HTML file with a link to your applet.

7. (You're now starting Phase B.) Beginning again with VisualGraphApplet.java, create a new project FriendRecommender, with a source file FriendRecommender.java. Arrange for this applet to preload the friends test example FT1 (TBA).

8. Implement a method verticesAtDistanceK(GraphVertex v, int k) that returns a list of vertices in the graph that can be reached from v using a path of length k, but cannot be reached from v via any shorter path. In the following steps, the notation "<name>" means an identifier like "John", and the notation "vertex(<name>)" means an instance of the class GraphVertex whose name field matches the string <name>. In the starter code, you get a vertex from a name by looking up the name in the hash table vertexFromName. Implement a command TWO-AWAY <name> that applies verticesAtDistanceK(vertex(<name>), 2) and reports its answers in the history window and console. Also implement a similar command THREE-AWAY.

9. Implement a method neighborsInCommon(GraphVertex v1, GraphVertex v2), that returns a list of vertices that are both adjacent to v1 and adjacent to v2. Implement a command FRIENDS-IN-COMMON <name1> <name2>
that calls neighborsInCommon(vertex(<name1>), vertex(<name2>)), sorts the resulting list alphabetically by name, and reports the list of names in the history window and in the console.

10. Implement a method recommendFriends(GraphVertex v) that first gets a list of candidate new friends (here a list of vertices) for v by calling verticesAtDistanceK(v, 2), and then for each vertex vc in this list, calling neighborsInCommon(v, vc), getting a list NIC(v, vc) of neighbors in common. Finally, it should return a list of recommendations, where each recommendation is a structure of the form:
[vc_i; k_i friends in common: $v_{i,1}$, $v_{i,2}$, ... , $v_{i,k}$].
The list of recommendations should be sorted so that the candidate with the most friends in common with v is first in the list, etc. Then implement a command RECOMMEND-FRIENDS <name>. This should call recommendFriends(vertex(<name>)) and report (in the history window and console) the information returned by recommendFriends, in the format illustrated by the following example:

```
RECOMMEND-FRIENDS John
Suggested friends for John:
  Zach (5 friends in common: Beth, Kai, Pelle, Ruth, Tran);
  Mary (3 friends in common: Carrie, Pelle, Stan);
  Annie (1 friend in common: Ruth);
  Karl (1 friend in common: Kai);
```

11. Implement a command RECOMMEND-FOR-ALL that can use fork-join parallelism (with the Fork-Join framework) to find the friend recommendations for all people in the social network. You should also implement a command NPROCESSORS <p> that sets the expected number of available processors, thus influencing how your RECOMMEND-FOR-ALL creates new recursive tasks. If p=1, then RECOMMEND-FOR-ALL should not use any parallelism, even if it is available. This will allow you to measure the work. On the other hand, if p>1, then you can create any number of recursive tasks, not necessarily equal to p. For example, if would be fine to have p=2 and 50 recursive tasks (if that is reasonable for the graph your program is processing).

12. (optional for extra credit) You have been hired by FacePlace.com to implement a new

service called "Friend of the week". Members who opt into this service will not only get friend recommendations regularly, but once a week, they will automatically get a new friend (unless no-one is available). This friend will be determined by the system. The new friend will be someone from the recommendations list.

Design and implement an algorithm to determine the new friend pairs. Add a new command JOIN-FOW <name> that indicates that the member with name <name> is opting into Friend-of-the-week. Next, add a new command FOW. This command should run your new algorithm, add an edge to the graph for each new friendship, and highlight the edges just added. If it is run again, it should unhighlight any previously highlighted edges, so that only the most recently added friendship edges are highlighted.

13. (optional for extra credit) Modify your solution to (12) to work in parallel. Be sure that your parallel tasks do not assign more than one friend to each member who has opted into the service. This may require locking prior to trying to commit a new friendship. Your program should print out, in the console, what each recursive task is doing as it does it. For example, it might print out:

```
"Recursive task 13 is looking for a FOW for Mary."
"Recursive task 7 is finding candidate friends for Jim."
"Recursive task 5 has found Jim a FOW in Thomas."
```

14. Get your applet to correctly run the canonical demo for Phase B.
15. Jar-up your applet and put it on the web.
16. Turn in your advanced applet code and link file.

Further Instructions and Information: Updates and clarifications to this project will be posted in the Catalyst GoPost discussion topic “Project 3”.