# CSC172 Lab: Minimum Spanning Trees

November 3, 2014

## 1 Introduction

The labs in CSC172 follow a pair programming paradigm. Every student is encouraged (but not strictly required) to have a lab partner. Labs will typically have an even number of components. The two partners in a pair programming environment take turns at the keyboard. This paradigm facilitates code improvement through collaborative efforts, and exercises the programmers cognitive ability to understand and discuss concepts fundamental to computer programming. The use of pair programming is optional in CSC172. It is not a requirement. You can learn more about the pair programming paradigm, its history, methods, practical benefits, philosophical underpinnings, and scientific validation at http://en.wikipedia.org/wiki/Pair_programming .

Every student must hand in his own work, but every student must list the name of the lab partner (if any) on all labs.

This lab has 3 parts. You and your partner(s) should switch off typing each part, as explained by your lab TA. As one person types the lab, the other should be watching over the code and offering suggestions. Each part should be in addition to the previous parts, so do not erase any previous work when you switch.

The textbook should present examples of the code necessary to complete this lab. However, collaboration is allowed. You and your lab partner may discuss the lab with other pairs in the lab. It is acceptable to write code on the white board for the benefit of other lab pairs, but you are not allowed to electronically copy and/or transfer files between groups.

## 2 Random Graphs

Write a program to generate an N-node random undirected graph description as an adjacency matrix. That is, a symmetric matrix of undirected edge weights with A[i,j] being the edge weight from node i to node j. Take A[i,j] = 0 to mean there is no connecting edge. Symmetry (undirected edges) means A[i,j] = A[j,i]. A random graph function pseudocode prototype might be

```
RandomGraphAdjArr = NewRGAA(N, density, low-wt, hi-wt)
```

which produces an NxN symmetric array of random edge weights with values between low-wt and hi-wt (best if they are integers so humans can check results).

Now write a program to translate the adjacency array into an adjacency list representation. You may have written code for such a representation already; if so feel free to use it.

# 3 Do one of: Prim's Algorithm

Each programming pair should choose this section or the next.

Implement Prim's algorithm for the MST using your adjacency list representation. Test it on the graph below and on some small random examples that are easy to check by hand.

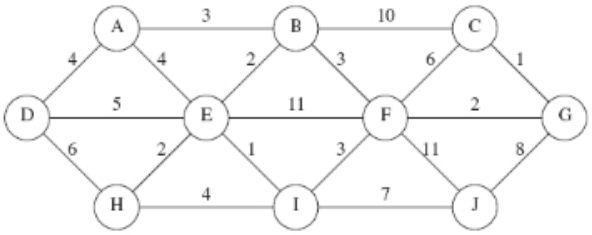


**Figure 9.84** Graph used in Exercise 9.15

Recall (Weiss 9.5.1) that Prim is like Dijkstra, so you should be able to adapt your shortest path code (if any) to this problem and save time.

# 4 Or Kruskal's Algorithm

Each programming pair should choose this section or the last.

Implement Kruskal's algorithm for the MST using your adjacency list representation. Test it on the graph above and on some small random examples that are easy to check by hand.

Recall (Weiss 9.5.2) that Kruskal uses an operation like the Union in union-find for discrete sets, so you should be able to adapt your union-find code (if any) to this problem and save time.

# 5 Experiments

Do the algorithms behave when negative weights are allowed (make lo-wt negative in NewRGAA() for some small examples and try to spot a problem.)

Timing: get a collection of running times for increasing number of nodes N and try to verify, for your MST algorithm, the running times given in the Weiss text.

# 6 Grading

172/grading.html
Each section (1-6) accounts for 15% of the lab grade (total 90%)
(README file counts for 10%)