Fall 2000 **CMSC 451: Programming Assignment** Clyde Kruskal

Due last day of class

This is a *special programming assignment.* It can only *help (but not guarantee) borderline students* move from a D to a C. If you choose to do this assignment, you must do it with little guidance from the instructor.

You may use any "reasonable" computer and any "reasonable" computer language. Your overall goal is to implement both the $\Theta(m \log n)$ version of Dijkstra's algorithm and Floyd's algorithm on *undirected* graphs and and compare them.

You will need a priority queue with the following operations: insert, delete minimum, and update to a *smaller* value. The logarithmic time priority queue can be implemented as a heap. An insert puts an item at the end of the heap and sifts up. A delete minimum removes the root; conceptually places the last item in the heap as the new root; and sifts it down to its proper location. (This last operation is more efficiently implemented by placing the last element to the side leaving the root as a "hole"; sifting the hole down to the bottom of the heap; placing the last element into the new bottom hole; and sifting the element up. This also has the advantage that you can reuse the sift up routine.) Update to a smaller value sifts an element up.

(a) Implement Floyd's algorithm.

(b) Implement the $\Theta(m \log n)$ version of Dijkstra's algorithm. Use an adjacency list to store the graph. You will will have to implement a heap, and be able to *delete* a node from the heap and *update* the value of a node on the heap.

(c) Execute your programs on large dense and sparse graphs, for a variety of graph sizes. Compare the running times of the two programs on each graph. Graph your results. What do you conclude?

Informally, a dense graph has many edges (relative to the number of vertices). It is sufficient to have every possible edge ($n(n-1)/2$). You can give each node a random weight.

Informally, a sparse graph has few edges. You should restrict yourself the graphs with a linear number of edges (with respect to the number of vertices). One way to do this is to make every node adjacent to some fixed number of other randomly chosen nodes (say three or four), and give the edges random weights. Another way is to give every possible edge a small probability (proportional to about $1/n$) of actually being an edge. One difficulty with either way is that such a graph may end up with only a few nodes having paths from the source, thereby producing trivial shortest-path problems. To avoid this, it is a good idea to initially imbed a tree in the graph (with largish weight edges). The simplest tree is a simple path starting at the source.

In order to get timings, it is sometimes useful to execute the program on the same graph many times (and then divide your time by the number of iterations). This has the advantage of being able to time "fast" programs on "small" graphs.

You should hand in a report that includes the following:

(a) A copy of your program for Floyd's algorithm. Some test cases that "show" that your program executes correctly.

(b) A copy of your program for Dijkstra's algorithm. Some test cases that "show" that your program executes correctly.

(c) Explain how your random graphs were chosen.

(d) One or more (real) graphs that plot graph size versus running time for each program on a variety of data sets. Identify the graph sizes and densities where one program (algorithm) is better than the other.

Feel free to extend the assignment. For example, you might compare different ways of producing random graphs, or different densities of graphs. Is it really necessary to imbed a tree in order to obtain interesting performance data on sparse graphs?