HW 4: Experimental Assignment II

Implement Kruskal's and Prim-Dijkstra's MST finding algorithms and compare their execution times on random graphs of varying sizes (number of nodes) and density (number of edges).

Your report should include

1. A source code listing for each algorithm. The code must be adequately documented and formatted.

2. A brief description about the computer system you employed (if a PC, then state clock rate, and CPU's name), language used (C recommended), and the random number generator you used.

3. A comparison plot as described below.

Notes:

1. *Comparison Plot (30 points)*: The range of sizes is not prescribed because the maximum size graph which can be run on your machine within a reasonable time (at most a minute per graph) will vary. Use n = 16, 32, 64, 128, 256, or more.

For a particular n, generate graphs with varying m = n, 2n, 4n, 8n, 16n, ..., n(n - 1)/2.

For each combination of n and m, generate five different random graphs, and find average time for each algorithm on those graphs, and plot. Be careful NOT to include graph generation time in the execution time.

Use density, m/n, as the x-axis and the execution time as the y-axis. Use logarithmic scale for x-axis so the x-axis will have 1, 2, 4, 8, ... at equal spacing.

Connect those points which pertain to same n and same algorithm together as a single curve. And label it with the algorithm's name and n such as (32, Kruskal) or (32, Prim). Thus, you will have a maximum of 10 different curves all on the same plot.

2. *Graph Generation (10 points):* For a given n and m, generate a pseudo-random graph as follows: Initialize a n x n weight matrix W to 0. Then deterministically generate the edges {1, 2}, {2, 3}, · · ·, {n - 1, n} to ensure that the graph gets connected. Thereafter, generate the rest of the m - n edges randomly. Generate each edge by generating three random numbers i, j, w where 1 <= j <= n, and 1 <= w <= 1000. Then set W [i, j] = w if W [i, j] was 0 before. Else, try another random edge until m edges have been generated. *(5 points)*

For the special case when m = n(n - 1)/2, the random procedure to generate this complete graph could be very inefficient, and take too long. So, in this case, simply generate n(n - 1)/2 different weights w, and fill in the upper triangular submatrix of W (above the main diagonal). (2 points)

For dense graphs, rather than trying to generate new edges, it might be a better idea to start with a complete graph as described in the preceding paragraph, and then delete random edges until only m edges remain. In this process, do not delete an edge of the form {i + 1, i} to ensure that the graph remains connected. (3 points)

3. *Data Structures (35 points)*: For Kruskal's algorithm, you will need (i) a min-heap to store all the edges prioritized by their weights (10 points), (ii) n trees representing n connected components in the beginning (5 points), (iii) a variable *TotalWeight* which will accumulate the weight of the edges in the spanning forest as it is formed, and (iv) an array T [1..n - 1] implementing the set of edges in the spanning tree. The spanning tree is needed only to verify the correctness of your algorithm.

Possible implementation of various data structures: The min-heap will be an array H[1..m] of records containing fields i, j, and w representing the edge {i, j} with weight w. The n trees can be implemented as an array [1..n] of records each containing two integers parent and rank. Parent is the node to which this node points to, and rank is the number of nodes in the subtree rooted at this node. Rank needs to be updated for only the root nodes, those which point to themselves. Notice that this avoids using explicit pointers. Else, one can implement the same employing trees with explicit pointers as well. (Your book has the pseudocode for W-Union (5 points) and C-Find (5 points), so use it),

For Prim's algorithm, the major data structures are (i) the array NEAR[1..n], (ii) a min-heap to store the nodes prioritized by their distance to the partial MST, supporting extract-min and decrease key (5 points) operations. Thus, we need to use the version of Prim's algorithm which can adapt to the density of a graph by using a min-heap. The array T can be used here as well.

Both Kruskal's and Prim's algorithm can use adjacency list structures to represent the graph (5 points). Prim's needs the list structures to enable quickly going through the neighbors of a node. Kruskal's can also scan through the upper/lower triangular portion of the weight matrix to collect all the edges into the heap. Do not include the adjacency list construction in Prim's or scanning of weight matrix to get edges into the heap array in Kruskal's algorithm into timing. Construction of heap itself is to be timed though.