University of Toronto
CSC 270/B70: Fundamental data structures and techniques
Fall 1999

Assignment 2 - Minimum Spanning Trees

Due Date: Nov 1, 1999, at 10:00 a.m. (St George campus)

Purpose:

There are two purposes to this assignment:

- to gain experience in using basic data structures, such as link lists, and in incorporating them in C programs.
- to gain familiarity with graph theoretical problems and algorithms

Introduction:

Many problems arising in the real world are best modelled as minimum spanning tree problems.

For example, suppose that you have a large number of communication points and you need to link them so that any site can communicate with every other site. You are not concerned with how long it takes for the message to be routed between sites (possibly through other sites), you just need there to be at least one route by which the message can be sent. Your only concern is to minimize the cost of building the links. Not every pair of sites can be feasibly linked. For each pair which can be linked, you know the cost to link them.

The best approach is to model this problem with a graph. Each site is represented by a node. If two sites can be linked then there is an undirected edge between the two corresponding nodes, and that edge has a weight equal to the cost of building such a link. The cheapest way to build the links will be represented by a *minimum weight spanning tree* of the graph, i.e. a subtree which meets every node, and which has the smallest total weight of all such trees.

Fortunately, there are some very efficent algorithms for solving the minimum spanning tree problem, the two most common being Kruskal's algorithm and Prim's algorithm. The first algorithm is described in section 9.5 of the graph theory portion of your readings. This assignment will focus on the other one.

Your assignment is to implement Prim's algorithm in C.

Prim's Algorithm: Prim's algorithm runs by slowly building a tree, one vertex(edge) at a time. At first the tree, T, contains only one vertex (it doesn't matter which vertex you start with). At each step, it adds a new vertex to the tree, along with an edge joining that vertex to a vertex which is already in the tree. The vertex/edge added is chosen by taking the edge which has the lowest weight amongst all edges which could be chosen. (If a number of edges are tied for the lowest weight, then one of them is chosen arbitrarily.)

If, before completing $n-1$ steps, there is no vertex/edge which can be legally added, then the input graph was disconnected and so it has no spanning tree.

In other words:

1. Choose any vertex v. Set $V(T) = \{v\}$. Set $E(T) = \emptyset$.
2. Repeat the following step $n-1$ times:
  (a) Consider the set of all edges of the form ab where $a \in V(T), b \notin V(T)$.
  (b) Let ab be a cheapest edge in that set.
  (c) Set $V(T) = V(T) + \{b\}, E(T) = E(T) + \{ab\}$.
3. $E(T)$ contains the edges of a minimum spanning tree.

As usual, we assume that our graph has n vertices and m edges. For each iteration of Step 2, we must choose from amongst at most m edges. This can be done in $O(m)$ time. Since we perform Step 2 $n-1$ times, Prim's algorithm has a running time of at most $O(mn)$. Clever implementations of Prim's algorithm have even better running times (the best is almost $O(m+n)$).

Programming advice and imperatives: The input graph will be in the form of an adjacency matrix representation of a weighted undirected graph. If ij is an edge of the graph, then the entry $A(i,j)$ will be equal to the weight of the edge ij, which will always be a non-negative integer (i.e. at least 0). Iff ij is not an edge of the graph, then $A(ij)$ will be a negative integer.

Your program must then convert the input into an adjacency list representation of the same graph. You must use that representation when implementing Prim's algorithm.

Your output will simply be a list of the edges in a Minimum Spanning Tree. Enter this list of edges into the array `MST_list` in any order. The starter code contains a routine which will sort these edges into lexicographically increasing order before outputting them - you do not need to write this part.

You must split your code into several files, as indicated by the starter code, and then merge them using a makefile.

There are several ways to implement Prim's algorithm. It is up to you to find the most efficient way that you can. It is not important that you use the very best way possible, but if your implementation is extremely inefficient, then you might lose marks.

Error messages: Make sure that your program reports any errors that you feel are appropriate. At the very least, an error should be reported if the input graph is disconnected. The starter code contains a check to see whether the input is a valid adjacency matrix.

Testing: You must come up with several test inputs, including some on very large graphs, of at least 100 vertices. (The sky's the limit - if you want to run it on 1000 vertices, or more, please do - just use some judgement when printing out your results.) Of course, this raises the issue of how to provide these inputs, since you certainly don't want to construct many 100 by 100 adjacency matrices by hand. So part of your assignment is to do the following:

Write a program which will generate large adjacency matrices which you can use as test input.

There are several ways that you can do this. Try to think of different methods. As we get closer to the due date, a suggested method will be posted on the course web page.

Reporting: You must prepare a report containing the following.

- An interpretation of the results of several test runs. Make these runs using increasingly large inputs. Compare the running times to the size of the input graphs, and discuss this in light of the theoretical bound on the running time of the algorithm.
- List at least three real world applications which are best modelled by a minimum spanning tree problem - your applications should involve both practicality and originality.

What to hand in:

A printout of your program files.
The inputs/outputs and running times of several test runs (use some judgement so that large inputs/outputs don't take up too much space).
Your report.

You will also submit your code and test files electronically. More details regarding electronic submission will appear on the course web page.