

# COMP3600 Assignment 3

Felix Andrews (3285754)

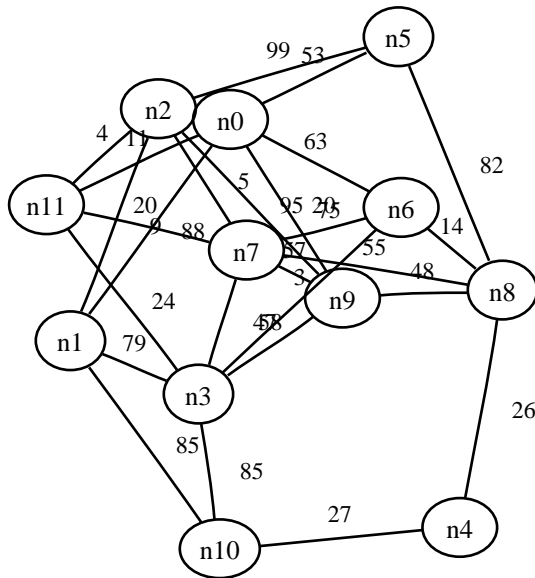
October 24, 2002

## 1 Minimum Spanning Trees

### 1.1 Random Graph Generation

We want to create a randomly-connected, randomly-weighted, undirected graph. The generation procedure takes the number of vertices  $n$  and an upper bound on the number of edges  $m$ , and randomly adds edges  $m$  times. A depth-first search is used to check whether the graph is connected. If not, the process is repeated.

In this example  $n = 12$  and  $m = \lceil n \log n \rceil$ , which works out as  $m = 52$ , but the actual number of edges was 26. This diagram<sup>1</sup> provides a visual representation of the graph. Following it is an enumeration of each edge and its associated weight.



n1 -- n0 (weight 88)	n8 -- n7 (weight 55)
n2 -- n1 (weight 9)	n9 -- n0 (weight 75)
n3 -- n1 (weight 79)	n9 -- n2 (weight 95)
n5 -- n0 (weight 53)	n9 -- n3 (weight 58)

<sup>1</sup>Created by the `neato` program, part of the `graphviz` package (<http://www.research.att.com/sw/tools/graphviz/>)

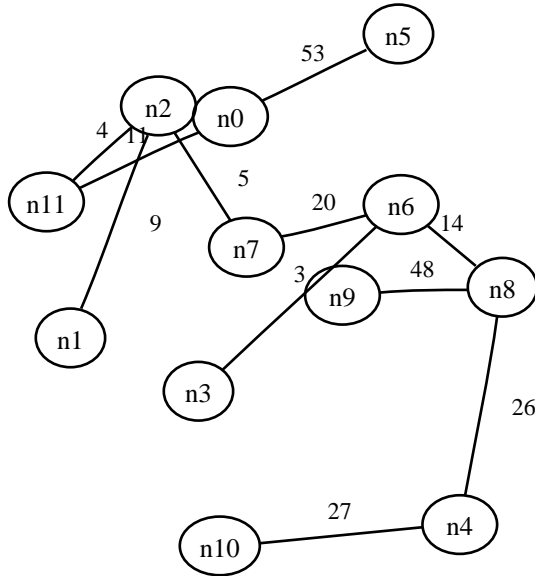
n5 -- n2 (weight 99)	n9 -- n7 (weight 57)
n6 -- n0 (weight 63)	n9 -- n8 (weight 48)
n6 -- n3 (weight 3)	n10 -- n1 (weight 85)
n7 -- n2 (weight 5)	n10 -- n3 (weight 85)
n7 -- n3 (weight 47)	n10 -- n4 (weight 27)
n7 -- n6 (weight 20)	n11 -- n0 (weight 11)
n8 -- n4 (weight 26)	n11 -- n2 (weight 4)
n8 -- n5 (weight 82)	n11 -- n3 (weight 24)
n8 -- n6 (weight 14)	n11 -- n7 (weight 20)

## 1.2 Kruskal's Algorithm

The graph was represented as an array of edge structures. An array of vertex structures was also used within the algorithm, organised into a *disjoint set forest*. The standard scheme of MAKE-SET, FIND-SET and UNION operations was used to manipulate the forest. For efficiency, the *union by rank* and *path compression* heuristics were implemented.

For sorting the edges by weight, the QUICKSORT algorithm was used.

The final minimum spanning tree was represented as an array of edges. Here is a diagram of the tree (it is of course a subset from the original graph). Following it is a list of the edges making up the MST. The algorithm has sorted them by weight. The total weight is 220.



n6 -- n3 (weight 3)	n7 -- n6 (weight 20)
n11 -- n2 (weight 4)	n8 -- n4 (weight 26)
n7 -- n2 (weight 5)	n10 -- n4 (weight 27)
n2 -- n1 (weight 9)	n9 -- n8 (weight 48)
n11 -- n0 (weight 11)	n5 -- n0 (weight 53)
n8 -- n6 (weight 14)	

### 1.3 Prim's Algorithm

For Prim's algorithm the graph was represented as an adjacency matrix. This is not the optimal structure for iterating over adjacent nodes (a linked list would include only the relevant entries) but in practice skipping over null entries is trivial. The key to Prim's algorithm is the EXTRACT-MIN function, which extracts a *light edge* crossing the MST boundary (the front moving over the graph maintaining the MST property). Ideally this would be implemented using a binary heap or Fibonacci heap, but here it is a simple linear search.

The resulting MST is represented as a predecessor array, mirroring the tree structure developed with the algorithm. In this case it is the same as for Kruskal's algorithm (it is possible to have different but equally valid MSTs). Following is an edge enumeration:

```
n1 -- n2 (weight 9)           n7 -- n2 (weight 5)
n2 -- n11 (weight 4)         n8 -- n6 (weight 14)
n3 -- n6 (weight 3)          n9 -- n8 (weight 48)
n4 -- n8 (weight 26)         n10 -- n4 (weight 27)
n5 -- n0 (weight 53)         n11 -- n0 (weight 11)
n6 -- n7 (weight 20)
```

### 1.4 Running Times

Here are the real running times for calculating an MST from graphs of each size and density. The first block shows the actual value of  $m$  (number of edges) for each test. The second block gives the times (in seconds, averaged over several runs)—these are for the algorithm only, without the graph generation or printing. All tests were run on an AMD Athlon 850MHz CPU.

edges	$m = \lceil 1.5n \log n \rceil$	$m = \lceil n^{4/3} \rceil$	$m = \lceil n^2/8 \rceil$
$n = 13$	73	31	22
$n = 120$	1242	592	1800
$n = 500$	6787	3969	31250
$n = 1000$	14948	10000	125000
seconds	$m = \lceil 1.5n \log n \rceil$	$m = \lceil n^{4/3} \rceil$	$m = \lceil n^2/8 \rceil$
$n = 13$	0.00	0.00	0.00
$n = 120$	0.00	0.00	0.00
$n = 500$ (Prim)	0.02	0.02	0.02
$n = 500$ (Krus.)	0.02	0.01	0.30
$n = 1000$ (Prim)	0.08	0.08	0.09
$n = 1000$ (Krus.)	0.10	0.05	5.05

With  $n = 120$ , occasionally one algorithm is slightly slower than the other, enough that it registers as 0.01 seconds.

From these results it seems that Prim's algorithm scales well with increasing graph density (running time was almost the same over an order of magnitude increase). On the other hand, Kruskal's algorithm shows a dramatic increase in running time as graph density increases. However, Kruskal's algorithm was the fastest on a relatively sparse graph (average 10 edges per node). This trend was confirmed with  $n = 1000$  on a very sparse graph (average 3 edges per node:  $m = 3000$ ) where Prim ran in 0.07 seconds and Kruskal ran in just 0.01 seconds.

This makes intuitive sense considering the action of each algorithm, ignoring the data structures used.