Minimum Spanning Trees

- Minimum Spanning Tree Definitions?
- Greedy Algorithms
- Kruskal's Method and Partitions
- Prim's Method

Reading: Cormen, Leiserson, Rivest and Stein Chapter 23

Minimum spanning tree (MST)

Consider a group of villages in a remote area that are to be connected by telephone lines. There is a certain cost associated with laying the lines between any pair of villages, depending on their distance apart, the terrain and some pairs just cannot be connected.

Our task is to find the minimum possible cost in laying lines that will connect all the villages.

This situation can be modelled by a weighted graph W, in which the weight on each edge is the cost of laying that line. A *minimum spanning tree* in a graph is a subgraph that is (1) a spanning subgraph (2) a tree and (3) has a lower weight than any other spanning tree.

It is clear that finding a MST for W is the solution to this problem.

The greedy method

Definition A *greedy algorithm* is an algorithm in which at each stage a locally optimal choice is made.

A greedy algorithm is therefore one in which no overall strategy is followed, but you simply do whatever looks best at the moment.
For example a mountain climber using the greedy strategy to climb Everest would at every step climb in the steepest direction. From this analogy we get the computational search technique known as *hill-climbing*.

In general greedy methods have limited use, but fortunately, the problem of finding a minimum spanning tree can be solved by a greedy method.

Kruskal's method

Kruskal invented the following very simple method for building a minimum spanning tree. It is based on building a forest of lowest possible weight and continuing to add edges until it becomes a spanning tree.

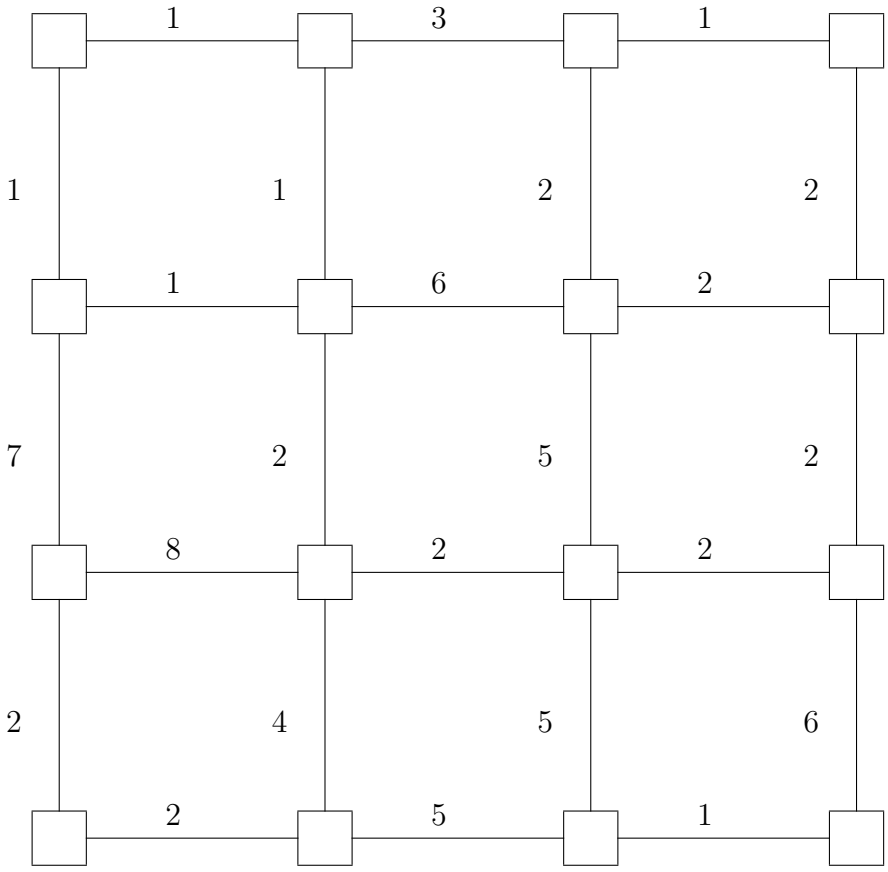
Kruskal's method

Initialize F to be the forest with all the vertices of G but none of the edges.

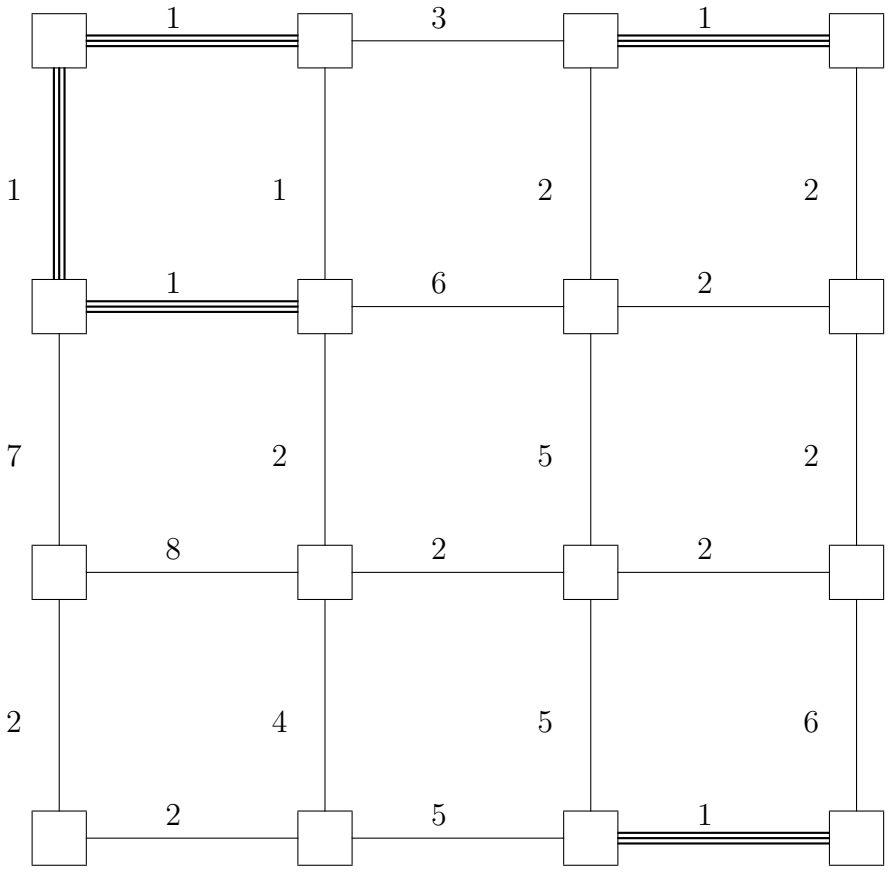
repeat
  Pick an edge e of minimum possible weight
  if $F \cup \{e\}$ is a forest **then**
    $F \leftarrow F \cup \{e\}$
  end if
until F contains $n - 1$ edges

Therefore we just keep on picking the smallest possible edge, and adding it to the forest, providing that we never create a cycle along the way.

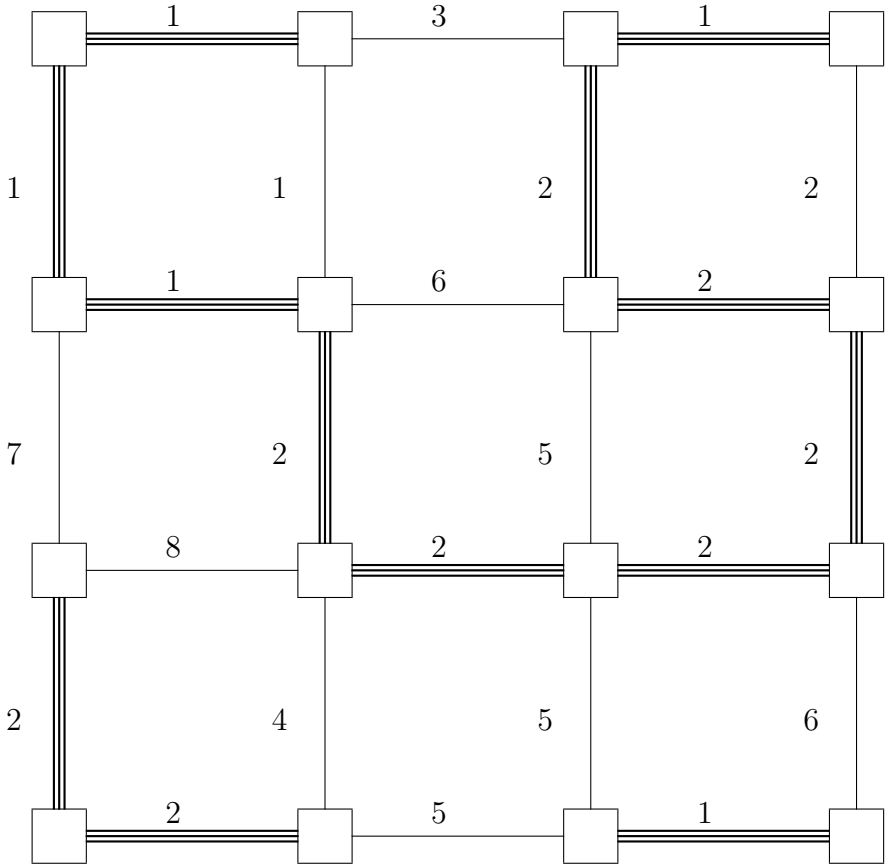
Example



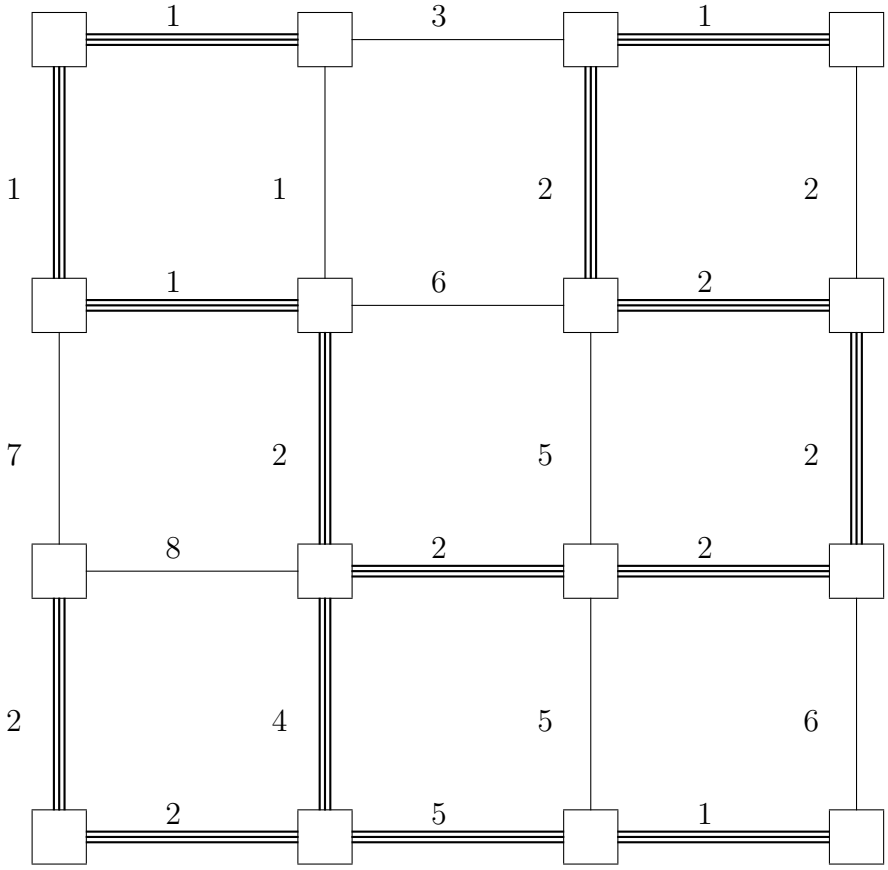
After using edges of weight 1



After using edges of weight 2



The final MST



Prim's algorithm

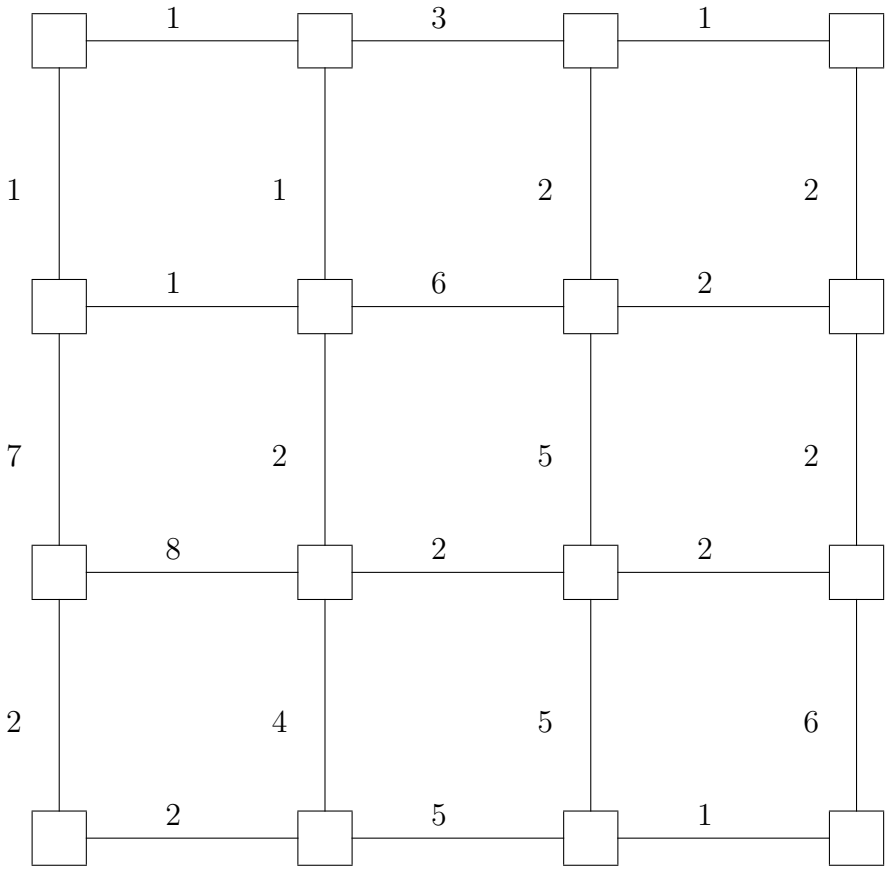
Prim's algorithm is another greedy algorithm for finding a minimum spanning tree.

The idea behind Prim's algorithm is to grow a minimum spanning tree edge-by-edge by always adding the shortest edge that touches a vertex in the current tree.

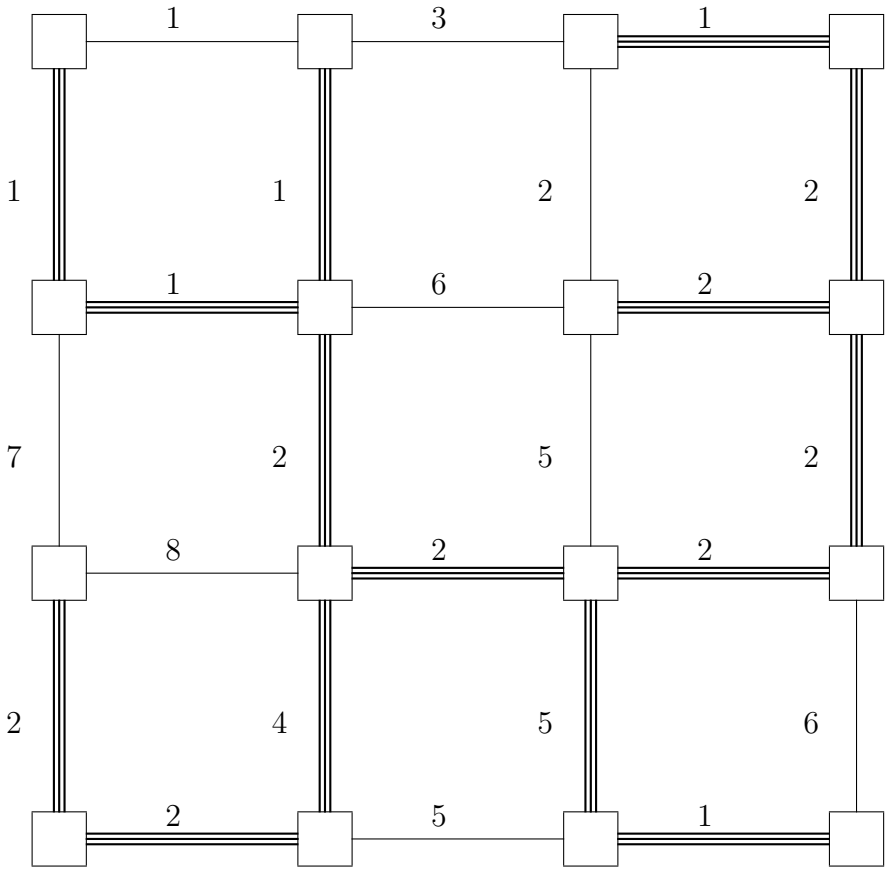
Notice the difference between the algorithms: Kruskal's algorithm always maintains a spanning subgraph which only becomes a tree at the final stage.

On the other hand, Prim's algorithm always maintains a tree which only becomes spanning at the final stage.

Prim's algorithm in action



One solution



Problem solved?

As far as a mathematician is concerned the problem of a minimum spanning tree is well-solved. We have two simple algorithms both of which are guaranteed to find the best solution. (After all, a greedy algorithm must be one of the simplest possible).

In fact, the reason why the greedy algorithm works in this case is well understood — the collection of all the subsets of the edges of a graph that do not contain a cycle forms what is called a (graphic) **matroid** (see CLRS, section 16.4).

Loosely speaking, a greedy algorithm always works on a matroid and never works otherwise.

Implementation issues

In fact the problem is far from solved because we have to decide how to *implement* the two greedy algorithms.

The details of the implementation of the two algorithms are interesting because they use (and illustrate) two important data structures — the *partition* and the *priority queue*.

Implementation of Kruskal

The main problem in the implementation of Kruskal is to decide whether the next edge to be added is allowable — that is, does it create a cycle or not.

Suppose that at some stage in the algorithm the next shortest edge is $\{x, y\}$. Then there are two possibilities:

x and y lie in different trees of F: In this case adding the edge does not create any new cycles, but merges together two of the trees of F

x and y lie in the same tree of F: In this case adding the edge creates a cycle and the edge should not be added to F

Therefore we need data structures that allow us to quickly find the tree to which an element belongs and quickly merge two trees.

Partitions or disjoint sets

The appropriate data structure for this problem is the *partition* (sometimes known under the name disjoint sets). Recall that a partition of a set is a collection of disjoint subsets (called cells) that cover the entire set.

At the beginning of Kruskal's algorithm we have a partition of the vertices into the discrete partition where each cell has size 1. As each edge is added to the minimum spanning tree, the number of cells in the partition decreases by one.

The operations that we need for Kruskal's algorithm are

Union(cell,cell) Creates a new partition by merging two cells

Find(element) Finds the cell containing a given element

Naive partitions

One simple way to represent a partition is simply to choose one element of each cell to be the "leader" of that cell. Then we can simply keep an array π of length n where $\pi(x)$ is the leader of the cell containing x.

Example Consider the partition of 8 elements into 3 cells as follows:

$$\{0, 2 \mid 1, 3, 5 \mid 4, 6, 7\}$$

We could represent this as an array as follows

x	0	1	2	3	4	5	6	7
$\pi(x)$	0	1	0	1	4	1	4	4

Then certainly the operation **Find** is straightforward — we can decide whether x and y are in the same cell just by comparing $\pi(x)$ with $\pi(y)$.

Thus **Find** has complexity $\Theta(1)$.

Updating the partition

Suppose now that we wish to update the partition by merging the first two cells to obtain the partition

$$\{0, 1, 2, 3, 5 \mid 4, 6, 7\}$$

We could update the data structure by running through the entire array π and updating it as necessary.

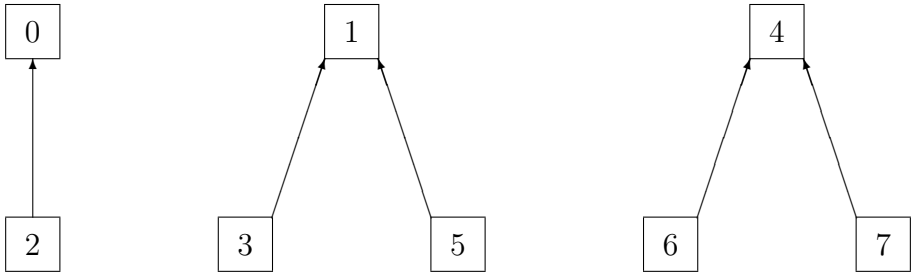
x	0	1	2	3	4	5	6	7
$\pi(x)$	0	0	0	0	4	0	4	4

This takes time $\Theta(n)$, and hence the merging operation is rather slow.

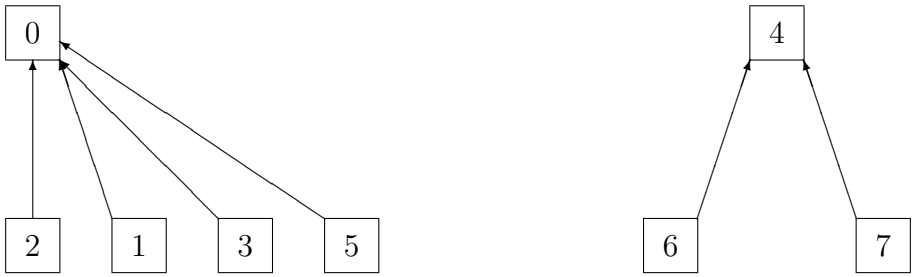
Can we improve the time of the merging operation?

The disjoint sets forest

Consider the following graphical representation of the data structure above, where each element points (upwards) to the "leader" of the cell.



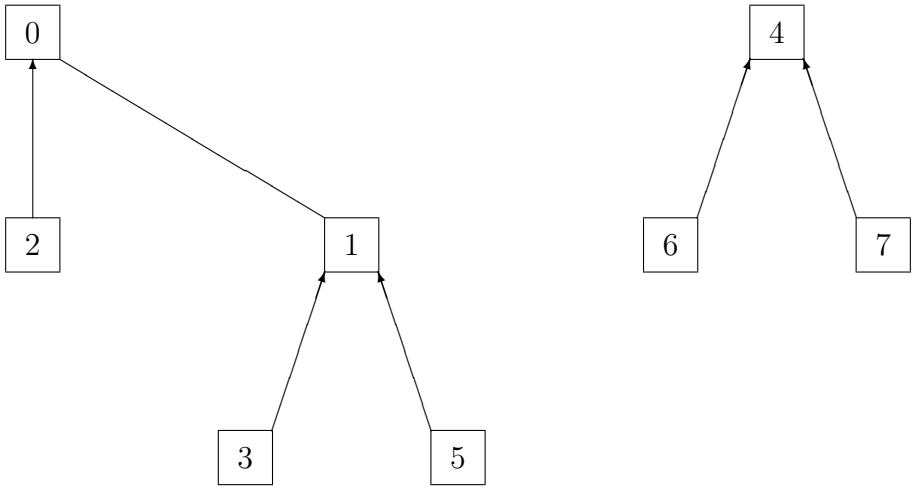
Merging two cells is accomplished by adjusting the pointers so they point to the new leader.



Suppose we simply change the pointer for the element 1, by making it point to 0 instead of itself.

The new data structure

Just adjusting this one pointer results in the following data structure.



procedure Find(x)
  while $x \neq \pi(x)$
    $x = \pi(x)$

This new find operation may take time $O(n)$ so we seem to have gained nothing.

Union-by-rank heuristic

There are two heuristics that can be applied to the new data structure, that speed things up enormously at the cost of maintaining a little extra data.

Let the *rank* of a root node of a tree be the height of that tree (the maximum distance from a leaf to the root).

The *union-by-rank* heuristic tries to keep the trees *balanced* at all times. When a merging operation needs to be done, the root of the shorter tree is made to point to the root of the taller tree. The resulting tree therefore does not increase its height unless both trees are the same height in which case the height increases by one.

Path compression heuristic

The path compression heuristic is based on the idea that when we perform a **Find(x)** operation we have to follow a path from x to the root of the tree containing x.

After we have done this why do we not simply go back down through this path and make all these elements point *directly* to the root of the tree, rather than in a long chain through each other?

This is reminiscent of our naive algorithm, where we made *every* element point directly to the leader of its cell, but it is much cheaper because we only alter things that we needed to look at anyway.

Complexity of Kruskal

In the worst case, we will perform E operations on the partition data structure which has size V. By the complicated argument in CLRS we see that the total time for these operations if we use both heuristics is $O(E \lg^* V)$. (Note: $\lg^* x$ is the iterated log function, which grows *extremely* slowly with x; see CLRS, page 56)

However we must add to this the time that is needed to sort the edges — because we have to examine the edges in order of length. This time is $O(E \lg E)$ if we use a sorting technique such as mergesort, and hence the overall complexity of Kruskal's algorithm is $O(E \lg E)$.

Implementation of Prim

For Prim's algorithm we repeatedly have to select the next vertex that is *closest* to the tree that we have built so far. Therefore we need some sort of data structure that will enable us to associate a value with each vertex (being the distance to the tree under construction) and rapidly select the vertex with the lowest value.

From our study of Data Structures we know that the appropriate data structure is a *priority queue* and that a priority queue is implemented by using a *heap*.

Prim's algorithm

It is now easy to see how to implement Prim's algorithm.

We first initialize our priority queue Q to empty. We then select an arbitrary vertex s to grow our minimum spanning tree A and set the key value of s to 0. In addition, we maintain an array, π, where each element $\pi[v]$ contains the vertex that connects v to the spanning tree being grown.

Here we want low key values to represent high priorities, so we will suppose that the highest priority is given to the lowest key-value.

Next, we add each vertex $v \neq s$ to Q and set the key value $key[v]$ using the following criteria:

$$key[v] = \begin{cases} weight(v, s) & \text{if } (v, s) \in E \\ \infty & \text{otherwise} \end{cases}$$

Since low key values represent high priorities, the heap for Q is so maintained that the key associated with any node is smaller (rather than larger) than the key of any of its children. This means that the root of the binary tree always has the smallest key.

We store the following information in the minimum spanning tree A: $(v, key[v], \pi[v])$. Thus, at the beginning of the algorithm, $A = \{(s, 0, \texttt{undef})\}$.

At each stage of the algorithm:

1. We extract the vertex u (using `dequeue()`) that has the highest priority (that is, the lowest key value!).
2. We add $(u, key[u], \pi[u])$ to A

3. We then examine the neighbours of u. For each neighbour v, there are two possibilities:

   (a) If v is is already in the spanning tree A being constructed then we do not consider it further.
   (b) If v is currently on the priority queue Q, then we see whether this new edge (u, v) should cause an update in the priority of v. If the value $\mathit{weight}(u, v)$ is lower than the current key value of v, then we change $\mathit{key}[v]$ to $\mathit{weight}(u, v)$ and set $\pi[v] = u$. This can be achieved by adding the element v into the priority queue with the new lower key-value.

At the termination of the algorithm, $Q = \emptyset$ and the spanning tree A contain all the edges, together with their weights, that span the tree.

Complexity of Prim

The complexity of Prim's algorithm is dominated by the heap operations.

Every vertex is extracted from the priority queue at some stage, hence the **extractmin** operations in the worst case take time $O(V \lg V)$.

Also, every edge is examined at some stage in the algorithm and each edge examination potentially causes a **change** operation. Hence in the worst case these operations take time $O(E \lg V)$.

Therefore the total time is

$$O(V \lg V + E \lg V) = O(E \lg V)$$

Which is the better algorithm: Kruskal's or Prim's?

Summary

1. Finding minimum spanning trees is a typicaly graph optimization problem.
2. Greedy algorithms take locally optimum choices to find a global solution.
3. MST's can be found using Kruskal's or Prim's algorithm
4. An $O(E \lg E)$ implementation of Kruskal's algorithm relies on the partition data structure.
5. An $O(E \lg V)$ implementation of Prim's algorithm relies on the priority queue data structure.