Notes on Kruskal’s Algorithm for Minimal Spanning Tree

In Kruskal’s algorithm (§23.2) the edges are ordered $e_1, \ldots, e_E$ by weight and $e_i$ is added to the tree if and only if its addition does not cause a cycle. The data structure that does this efficiently is covered in detail in Chapter 21, which we are not covering. Instead, these notes give a specific implementation of the algorithm. Assume the edges have already been ordered by weight and $x_i, y_i$ are the vertices of $e_i$. To each vertex $x$ we have functions $\pi(x)$ and $\text{SIZE}(x)$, initially all $\pi(x) \leftarrow x$ and all $\text{SIZE}(x) \leftarrow 1$.

For $i = 1$ to $E$ we set (for notational convenience) $x \leftarrow x_i$, $y \leftarrow y_i$ and do the following:

WHILE $\pi(x) \neq x$
   $x \leftarrow \pi(x)$ (*going down the stairs*)

WHILE $\pi(y) \neq y$
   $y \leftarrow \pi(y)$ (*going down the stairs*)

IF $x \neq y$ then DO
   IF $\text{SIZE}(x) \leq \text{SIZE}(y)$ then DO
      $\pi(x) \leftarrow y$
      $\text{SIZE}(y) \leftarrow \text{SIZE}(y) + \text{SIZE}(x)$
   OTHERWISE DO
      $\pi(y) \leftarrow x$
      $\text{SIZE}(x) \leftarrow \text{SIZE}(x) + \text{SIZE}(y)$

Add $e_i$ to Minimal Spanning Tree

At any time the $\pi(x)$ will give a rooted forest with $\pi(x) = x$ exactly when $x$ is a root. In that case $\text{SIZE}(x)$ will be the size of the forest. Certain edges will have already been put in the Minimal Spanning Tree so that the structure will be a forest. That forest and the forest given by $\pi(x)$ will have the same components (though they may have different edges).

Example. $e_1 = (a, c)$, $e_2 = (c, b)$, $e_3 = (d, e)$, $e_4 = (a, d)$, $e_5 = (b, d)$. With $i = 1$ we add $e_1$ to tree, $\pi(a) \leftarrow c$ and $\text{SIZE}(c) \leftarrow 2$. With $i = 2$ we add $e_2$ to tree, as $\text{SIZE}(c) > \text{SIZE}(b)$ we set $\pi(b) \leftarrow c$ and $\text{SIZE}(c) \leftarrow 3$.

With $i = 3$ we add $e_3$ to tree, $\pi(d) \leftarrow e$ and $\text{SIZE}(e) \leftarrow 2$. Now $i = 4$ so $x \leftarrow a; y \leftarrow d$. The WHILE parts trace $x$ down to its root $c$ and $y$ down to its root $e$. As $\text{SIZE}(c) > \text{SIZE}(e)$ we set $\pi(e) \leftarrow c$, $\text{SIZE}(e) \leftarrow 5$ and add $e_4$ to the tree. Note that the current state of the Minimal Spanning Tree and the forest given by $\pi$ have different edges but the same components. Now with $i = 5$, $x \leftarrow b; y \leftarrow e$. Both $x, y$ trace down with the WHILE loops to the same $c$ so we do nothing and $e_5$ is not added to the tree.
To analyze the time we note that the process is done $E$ times, so we analyze the process with a particular $x = x_i, y = y_i$. The key aspect to the time is we must iterate $\pi(x) \leftarrow x$ until reaching a root. (Similarly for $y$.) At first blush, this seems like it might take time $V$. ($V$ is number of vertices.) However, here we use the fact that when we earlier considered an edge $x, y$ and we moved them down to their roots we then reset $\pi(x) \leftarrow y$ where $\text{SIZE}[y]$ had been bigger than $\text{SIZE}[x]$. Now the new $\text{SIZE}[y]$ became the old $\text{SIZE}[x] + \text{SIZE}[y]$. That is, the new $\text{SIZE}[y]$ is at least double the old $\text{SIZE}[x]$. As $x$ is no longer a root its value of $\text{SIZE}[x]$ will never change. The value of $\text{SIZE}[y]$ may change later, but it can only get larger. Hence we will have $2 \cdot \text{SIZE}[x] \leq \text{SIZE}[y]$ forevermore. Therefore, as we look at a path $x, \pi(x), \pi(\pi(x)), \ldots$ the value $\text{SIZE}(\cdot)$ at least doubles each iteration. Therefore the path can only be of length $\log V$. This is a big savings over the length $V$ without this aspect. Now the process with a particular $x, y$ takes time $O(\log V)$ and therefore the total time is $O(E \log V)$.

**Path Compression:** (This is extra material and not on the final!) Before we move “down the stairs” we save, temporarily, the original value of $x$. (same for $y$) with

$$\text{originalx} \leftarrow x$$

Then we go “down the stairs” to the new value of $x$. Now we go back up to $\text{originalx}$ and reset the entire path to arrow the new $x$:

$$\begin{align*}
z &\leftarrow \text{originalx} \\
\pi[z] &\leftarrow x \\
\text{WHILE} &\pi[z] \neq z \\
\pi[z] &\leftarrow x \\
z &\leftarrow \pi[z]
\end{align*}$$

That is, the entire path from $\text{originalx}$ to $x$ is now pointing directly to $x$. This has effectively doubled the time, as we go down the WHILE loop twice. However, when later in the program we have a WHILE loop that hits $\text{originalx}$ it will jump directly to $x$. That is, the path has been *compressed*. Analysis of path compression is remarkably subtle (mathematicians love it!) but lets just say that it gives an improved running time for MST when $n$ is really large.