Kruskal’s Minimum Spanning Tree
Algorithm

Jonathan Turner
January 31, 2013

Kruskal’s algorithm for the minimum spanning tree problem can be viewed as a special case of the general greedy method. It applies the following coloring rule to the edges in increasing order of their weight.

**Coloring Rule.** If the current edge has both of its endpoints in the same blue tree, color it red; otherwise color it blue.

Expressed in algorithmic notation, this becomes

```plaintext
procedure minspantree(graph G = (V, E), modifies set blue)
    vertex u, v; set edges;
    blue := {}; edges := E;
    Sort edges by weight;
    for {u, v} ∈ edges ⇒
        if u and v are in different blue trees ⇒
            blue := blue ∪ {u, v};
        fi;
    rof; // edges not added to blue are implicitly red
end;
```

The time required for the main loop is determined by the time required to test whether or not u and v are in the same blue tree. One way to do this is to perform a tree traversal starting from u, using the blue edges. If v is visited during this traversal, then u and v are in the same tree, otherwise they are not. However, this approach takes \( \Omega(n) \) time, yielding an overall running time that grows as \( \Omega(mn) \).

We can dramatically reduce the time for the main loop using a simple data structure that maintains a partition on the vertices. The **partition data structure** (also known as “disjoint sets” or “union-find”) divides the vertices
into disjoint subsets, with each subset identified by one of its elements (called the \textit{canonical element}). We define the following operations.

\begin{itemize}
  \item \textit{partition}(S): creates a partition on the set S, with each element of S forming a separate subset
  \item \textit{find}(x): returns the canonical element of the subset containing \textit{x}
  \item \textit{link}(x): merges the two subsets with canonical elements \textit{x} and \textit{y} and returns the canonical element of the new subset. This new subset replaces the original subsets.
\end{itemize}

This data structure is efficient, easy implement and can be used in a variety of different applications. Let’s see how it can be used with Kruskal’s algorithm.

```c
procedure minspantree(graph \( G = (V, E) \), modifies set \( blue \))
vertex \( u, v \); set edges; partition(\( V \));
\( blue := \{\}; \) \( edges := E; \)
Sort edges by weight;
for \( \{u, v\} \in \text{edges} \Rightarrow \)
  if \( \text{find}(u) \neq \text{find}(v) \Rightarrow \)
    \( \text{link}(\text{find}(u), \text{find}(v)); \)
    \( \text{blue} := \text{blue} \cup \{u, v\}; \)
  fi;
end;
```

Note that there are at most \( n - 1 \) \textit{link} operations and at most most \( 4m \) \textit{find} operations. We’ll see that these can be done in \( O(m \log n) \) time and since the edges can also be sorted in \( O(m \log n) \) time, this gives us an overall run time bound of \( O(m \log n) \). Here is a C++ version of the algorithm.

```c
void kruskal(Wgraph& wg, list<edge>& mstree) {
  edge e, e1; vertex u,v,cu,cv; weight w; int i = 0;
  Partition vsets(wg.n());
  edge *elist = new edge[wg.m()+1];
  for (e = wg.first(); e != 0; e = wg.next(e))
    elist[i++] = e;
  sortEdges(elist,wg);
  for (e1 = 1; e1 <= wg.m(); e1++) {
    e = elist[e1];
```
u = wg.left(e); v = wg.right(e); w = wg.weight(e);
cu = vsets.find(u); cv = vsets.find(v);
if (cu != cv) {
    vsets.link(cu,cv);
    mstree.push_back(e);
}
}

In this implementation, the `sortEdges` permutes the array `elist`, so that `elist[e] ≤ elist[e+1]`. The `partition` data structure is defined on the set of vertex numbers \{1, \ldots, n\}. 