Alan Hahn
8650 Project

# MST USING KRUSKAL'S ALGORITHM WITH UNION FIND AND PATH COMPRESSION

Given an undirected graph G with a real valued weight on each edge, it is of interest to find the minimum spanning tree T of G, defined to be the spanning tree of G such that the sum of the edge weights in T is minimum with respect to all spanning trees of G. This may be achieved by using any one of a number of greedy algorithms which constructs a minimum spanning tree edge by edge, including appropriate small weight edges and excluding appropriate large weight edges. In this paper one such algorithm is explored, Kruskal's algorithm. First, a discussion of a simple version of a data structure which is useful in the implementation of Kruskal's algorithm, called a Partition, is included. Then after introducing Kruskal's algorithm, two heuristics are discussed which are used to improve the runtime for the algorithm. Finally, some results of a test between the naive implementation and the implementation with path compression and union by rank are discussed.

0.1. **The Partition Class.** The Partition class is a data structure which maintains a set $S$ as a collection of parts of a partition. Throughout a part will refer to a set, which is a subset of $S$, within the partition. To achieve this the Partition class is comprised of three methods which maintain and update the parts of the partition of a given set. To identify a part of the partition, an arbitrary but unique element will be maintained within each part as a representative element of the part.

**Definition 1.** The following maintain and update the parts of the partition of a given set $S$.

initialize_partition($S$): Initialize the partition as a collection of one element parts, one for each $s$ in $S$.

find($s$): Return the representative element of the part containing the element $s$.

link($x, y$): Union the two parts whose representative elements are $x$ and $y$, and choose a new representative element to represent the one part.

To implement this data structure a map may be used, where for each element $s$ in $S$, the value for $s$ in the map may be thought of as the parent of $s$ in a rooted tree, where the tree is rooted at the representative element of a part of the partition. Each part in the partition may then be thought of as a tree rooted at its representative element. The initialization function above may then be implemented by setting the parent node of each element $s$ to itself. Then to union two parts with representatives $x$ and $y$, the link method may be implemented by updating the parent node of one of $x$ or $y$ to be $y$ or $x$, respectively. To find the representative element a given element $s$, the find method may be implemented by traversing the parent node path of $s$ until an element is reached which has itself as its parent, which is the representative element of the part which contains $s$.

An example may be illustrative:

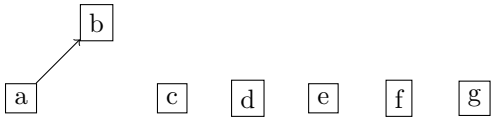
**Example 2.** Consider a set $S = \{a, b, c, d, e, f, g\}$
Initializing the Partition class makes a partition with a part for each vertex in $S$:

initialize_partition($S$)

$$[a : a,\ b : b,\ c : c,\ d : d,\ e : e,\ f : f,\ g : g]$$

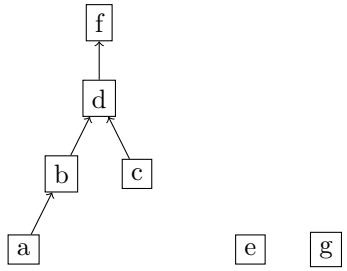

Linking $a$ and $b$ unions the part which contains $a$ and the part which contains $b$, and sets the representative element of the now single part which contains $a$ and $b$ to be $b$:
link($a, b$):

$$[a : b,\ b : b,\ c : c,\ d : d,\ e : e,\ f : f,\ g : g]$$




After inking $c$ to $d$, $b$ to $d$, then $d$ to $f$, there are three parts with representative elements $f$, $e$, and $g$:
link($c, d$)
link($b, d$)
link($d, f$)

$$[a : b,\ b : d,\ c : d,\ d : f,\ e : e,\ f : f,\ g : g]$$



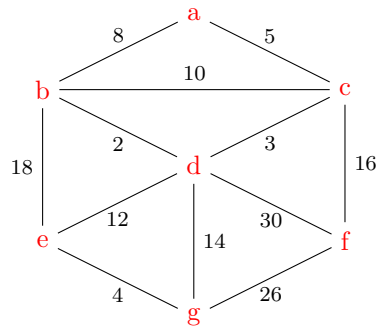

Now, find($a$) traverses through the parent path of $a$ and returns $f$; the parent of $a$ is $b$, the parent of $b$ is $d$, the parent of $d$ is $f$, and the parent of $f$ is $f$, and thus $f$ is the representative element of the part which contains $a$.

0.2. **Kruskal's Algorithm.** The problem of including appropriate edges in a construction of the minimum spanning tree is made concrete by considering this problem as an edge coloring problem; edges to be included in the minimum spanning tree are to be colored blue, and a set of edges of G which have been colored blue and are connected is called a blue subtree. Kruskal's algorithm consists of applying the following step to the edges of G in non-decreasing order by edge weight:

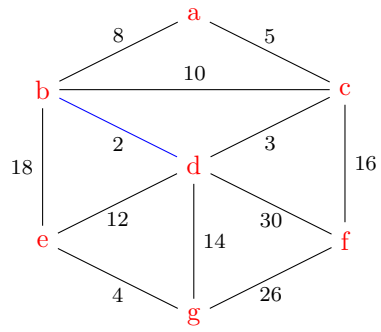
**Definition 3.** (Inclusion Step) If the edge $e = (u, v)$ is such that $u$ and $v$ are in the same blue subtree, leave $e$ uncolored. Else, color $e$ blue.

An example may be illustrative:

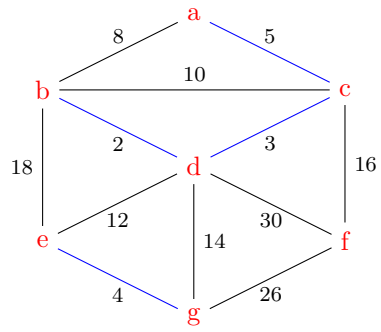
**Example 4.** Consider the following weighted graph $G$ with vertex set $V$:



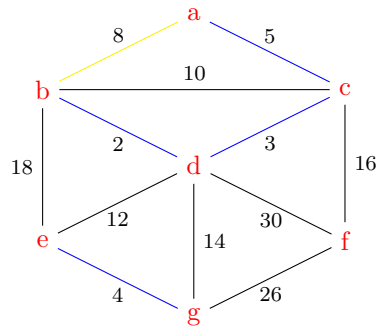

By ordering the vertices by edge weight in non-decreasing order, the first edge considered is edge $(b, d)$ with weight 2. $b$ and $d$ are not in the same blue subtree (as there are no blue subtrees as of yet), and the edge is thus colored blue:



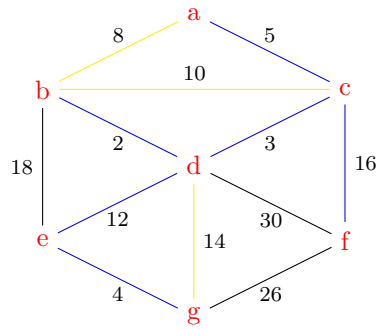

After four steps the graph has two blue subtrees, $\{(b, d), (c, d), (a, c)\}$ and $\{(e, g)\}$:




The next edge to be considered is $(a, b)$, both of whose vertices are in the same blue subtree and thus this vertex is skipped; for clarity the edge is colored yellow to show that it has been skipped:




Proceed in this manner until the number of edges amongst the blue subtrees is $|V| - 1$ or the end of the edge list is reached:




The blue subtree is the minimum spanning tree of $G$.

For a discussion of the proof that the algorithm returns a minimum spanning tree which is the blue subtree (or forest of blue subtrees if $G$ is not connected), see Ch. 6 of (Tarjan, 1983)

In order to implement Kruskal's algorithm, the edges must be sorted and there must be a way to tell whether two vertices are in the same blue subtree. Sorting the edges is relatively easy and may be done in many ways; for the simple algorithm the details of the implementation of sorting the edges are left to the reader. For a way to tell whether two vertices share a blue subtree, the Partition data structure from section 0.1 is utilized.

**Pseudo Code 5.** The following function takes as inputs the set of vertices V and the set of edges $E$ of an undirected, real-value weighted graph $G$, and produces the minimum spanning tree of $G$.

```
def krsklminspantree(V, E):
   blue = []
   E = E sorted by edge weight
   P = initialize_partition(V)
   for {x,y} in E:
        u = P.find(x)
        v = P.find(y)
        if u != v:
            P.link(u,v)
            blue.append({x,y})
   return(blue)
```

0.3. **Heuristics to improve runtime.** There are two heuristics which change the Partition class and which may be used to improve the runtime of Kruskal's algorithm. The first is called path compression which is a change in the find method; given an input element $v$ and representative element $s$ of the part which contains $v$, the find method updates to $s$ the parent node of each vertex on the parent node path from $v$ to $s$. This increases the runtime of a single find for a given input element $v$ but greatly decreases the runtime of subsequent executionss of find on vertices in the parent node path from $v$ to $s$, including $v$.

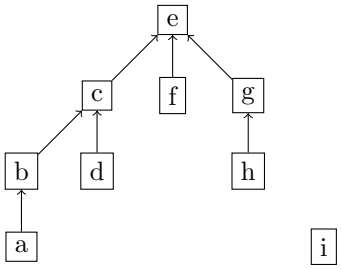
The second heuristic is a change to both the map and the link method in the Partition class, called union by rank. Along with the parent node, the map value of a given element $s$ returns a non-negative integer function called the rank of $s$, which is described further below. Given two representatives $x$ and $y$, if without loss of generality $\text{rank}(x) > \text{rank}(y)$, link chooses as the representative of the union of the parts which contain $x$ and $y$ to be $x$. If $\text{rank}(x) = \text{rank}(y)$, one of $\text{rank}(x)$, $\text{rank}(y)$ is increased by one and the element chosen to have higher rank is then the representative of the union of the parts which contain $x$ and $y$.

**Code 6.** The following is python code which implements the Partition class with the two heuristics mentioned above

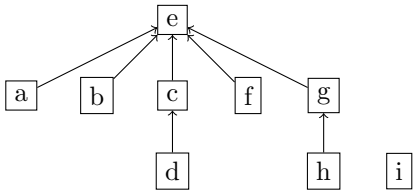
```
class Partition():

    def __init__(self,V):
        self._position = {}
        for v in V:
            self._position[v] = [v,0]

    def find(self,v):
        if self._position[v][0] != v:
            self._position[v][0] = self.find(self._position[v][0])
        return(self._position[v][0])

    def link(self,u,v):
        if self._position[u][1] > self._position[v][1]:
            u,v = v,u
        if self._position[u][1] == self._position[v][1]:
             self._position[v][1]+=1
        self._position[u][0] = v
```

Some visuals may be helpful:

**Example 7.** A visualization of path compression.
Suppose for example that the partition of the set $S = \{a, b, c, d, e, f, g, h, i\}$ has the following representation:



The find method applied to $a$ then returns $e$, but also updates the parent node path of $a$ so that the partition has the following representation:

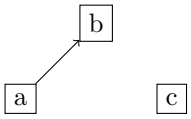



**Example 8.** Now, for a visualization of union by rank, consider a set S = {a,b,c} initialized to a partition:

$$[a : [a, 0],\ b : [b, 0],\ c : [c, 0]]$$

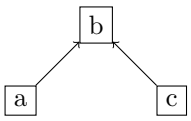

As $a$ and $b$ both have rank 0, link$(a, b)$ using the above python code sets the rank of $b$ to be $0 + 1 = 1$, and sets the parent of $a$ to be $b$:

$$[a : [b, 0],\ b : [b, 1],\ c : [c, 0]]$$




Then as rank$(b) = 1 > 0 =$ rank$(c)$, link$(b, c)$ sets the parent of $c$ to be $b$ and leaves the ranks unchanged:

$$[a : [b, 0],\ b : [b, 1],\ c : [b, 0]]$$




The runtime for a given find method execution without the path compression heuristic has worst case time $O(|V|)$; on an initial find method execution on a vertex $v$, the path compression heuristic will increase the runtime of find by a constant factor as the execution must make two passes along the parent node path - one to find the representative element $s$ and one to change the parents along the parent node path to $s$. However, subsequent executions to find the representative element $s$ for an element along the parent node path from $v$ to $s$ will run in constant time, and the runtime will decrease for elements in the subtrees rooted at an element in the parent node path from $v$ to $s$ as well.

The time savings for implementing union by rank are less straightforward to see, but one consequence of implementing union by rank and path compression is that there are bounds on the height of a tree rooted at a given element $x$ with rank$(x)$, and similarly there are bounds on the number of elements in a tree rooted at $x$ with rank$(x)$. These results may be found in Ch. 2 of (Tarjan, 1983) and are summarized as follows. Throughout, a partition of a set $V$ is considered.

**Lemma 9.** If $x$ is any node, rank($x$) $\leq$ rank(parent of $x$), with the inequality strict if the parent of $x$ is not $x$ itself. The value of rank($x$) is initially zero and increases as time passes until the parent of $x$ is assigned a value other than $x$; subsequently rank($x$) does not change. The value of rank(parent of $x$) is a nondecreasing function of time.

**Lemma 10.** The number of nodes in a tree with root $x$ is at least $2^{\text{rank}(x)}$.

**Lemma 11.** For any integer $k \geq 0$, the number of nodes of rank $k$ is at most $|V|/2^k$. In particular, every node has rank at most $\log(|V|)$.

We then have the following bounds on the time: The initialization function takes $O(|V|)$ time, and a single link method execution takes $O(1)$ time. A single find execution takes $O(\log(|V|)$. Thus for a sequence of $m$ Partition class operations, there is a bound on the runtime of $O(m\log(|V|))$. A careful analysis of the amortized runtime gives the following result, due to Tarjan (Tarjan, 1983), slightly reworded to better fit the context of this paper:

**Theorem 12.** Finding a minimum spanning tree using the Partition class with path compression and union by rank, given a sorted edge list $E$, runs in $O(|E|\alpha(|E|, |V|))$ time, where $\alpha(|E|, |V|)$ is a functional inverse of Ackerman's function.

The functional inverse of Ackerman's function cited, $\alpha(|E|, |V|)$, is an integer not greater than 5 for all practical purposes, and thus the dominating term in the runtime for Kruskal's algorithm is the time to sort the edges with time $O(|E|\log(|V|))$; thus Kruskal's algorithm runs in time $O(|E|\log(|V|))$.

0.4. **Results from a test in python.** In order to test the runtime between the naive implementation and the implementation with union by rank and path compression, I took a data set with 35,592 edges from the Stanford Network Analysis Project (Kumar, Spezzano, Subrahmanian, & Faloutsos, 2016), took the first 7000 edges, and computed a minimum spanning tree using both implementations. For the implementation with path compression and union by rank, the runtime was usually less than .3 seconds to find a minimum spanning tree. For the naive implementation, the runtime was regularly over 3 seconds. One other interesting thing to note is that for the whole data set of 35,592 eges, the implementation with path compression and union by rank terminated and output a minimum spanning tree, while the naive implementation would stop executing and return runtime errors - specifically, the recursion depth limit would be reached while executing the find method, thus highlighting the utility of path compression and union by rank.

## References

Kumar, S., Spezzano, F., Subrahmanian, V., & Faloutsos, C. (2016). Edge weight prediction in weighted signed networks. In *Data mining (icdm), 2016 ieee 16th international conference on* (pp. 221–230).

Tarjan, R. E. (1983). *Data structures and network algorithms.* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.