1 The Cut Property

A *tree* is an undirected graph which is connected and acyclic. A tree with n nodes has $n-1$ edges.[1] (*Proof:* Induction on n.) In fact, it is easy to see that if a graph $G = (V, E)$ that satisfies any two of the following four properties then it also satisfies the third, and is therefore a tree:

- $G = (V, E)$ is connected
- $G(V, E)$ is acyclic
- $|E| = |V| - 1$

Also, if a graph $G = (V, E)$ satisfies any one of the following properties, it is a tree:

- There is a unique path between any two nodes.
- If we add to G any edge not already in G, then a unique cycle results.

A *spanning tree* in a graph $G = (V, E)$ is a subset of edges $T \subseteq E$ such that (V, T) is a tree. That is, a spanning tree of G is a tree that is a subgraph of G and whose vertex set contains *(spans)* all nodes of V. It follows from the above conditions that a spanning tree must consist of exactly $|V| - 1$ edges. Now suppose that each edge has an integer weight associated with it: $w : E \to Z$. Say that the weight of a spanning tree T is the sum of the weights of its edges; $w(T) = \sum_{e \in T} w(e)$. The minimum spanning tree in a weighted graph G is one which has the smallest weight among all spanning trees in G.

It general, the number of spanning trees in G grows exponentially in V. Therefore it is infeasible to search through all possible spanning trees to find the lightest one. Luckily it is not necessary to examine all possible spanning trees; minimum spanning trees satisfy a very important property which makes it possible to efficiently zoom in on the answer.

We shall construct the minimum spanning tree of a graph by successively selecting edges to include in the tree. We will guarantee after the inclusion of each new edge that the set of edges selected so far, call it X, is contained in the set of edges of some minimum spanning tree, T. But how can we guarantee this if we don't yet know any minimum spanning tree in the graph? The following property provides this guarantee:

Cut property: Let $G = (V, E)$ be a graph with weights on its edges, and let X be a subset of edges of G such that $X \subseteq T$ where T is a minimum spanning tree of G. That is, X is a *forest* of G, it consists of many small disjoint trees, but it has the property that *it can be extended to a minimum spanning tree.* This means that, intuitively, in choosing the edges of X we have made no mistake that has made it impossible to eventually end up with a minimum spanning tree. Let S be a connected component of (V, X). Among all edges crossing between S and $V - S$, let e be an edge of minimum weight. Then $X \cup \{e\} \subseteq T'$ where T' is a MST of G. In other words, adding e to X is a correct step, it does not prevent us from eventually constructing a minimum spanning tree.

[1] It is usually very confusing that the term *tree* has two distinct meanings in computer science: If it is used in the context of undirected graphs, as in the present case, a tree is a graph that is connected and has no cycles. A *rooted* tree is a directed acyclic graph all nodes of which have outdegree one —except for a single node, the *root* that has outdegree zero. Undirected trees are interesting because they are the smallest possible connected subgraphs of a graph. Rooted trees are useful for modeling data structures, and for capturing the process of exploring a graph, as by depth-first search. If we disregard the directions of a rooted tree we get an undirected tree. And, given any undirected tree, we can select any node as its root, and direct all edges towards the root; a rooted tree results.

Proof: Suppose $e \notin T$. Adding e into T creates a unique cycle. We will remove a single edge e' from this unique cycle, thus getting $T' = T \cup \{e\} - \{e'\}$. It is easy to see that T' must be a tree — it is connected and has $n-1$ edges. Furthermore, as we shall show below, it is always possible to select an edge e' in the cycle such that it crosses between S and $V-S$. Now, since e is a minimum weight edge crossing between S and $V-S$, $w(e') \geq w(e)$. Therefore $w(T') = w(T) + w(e) - w(e') \leq w(T)$. However since T is a MST, it follows that T' is also a MST and $w(e) = w(e')$. Furthermore, since X has no edge crossing between S and $V-S$, it follows that $X \subseteq T'$ and thus $X \cup \{e\} \subseteq T'$.

How do we know that there is an edge $e' \neq e$ in the unique cycle created by adding e into T, such that e' crosses between S and $V-S$? This is easy to see, because as we trace the cycle, e crosses between S and $V-S$, and we must cross back along some other edge to return to the starting point.

In light of this, the basic outline of all our minimum spanning tree algorithms is going to be the following:

```
Algorithm general-mst($G=(V,E,w)$: edge-weighted graph);
$X$ set of edges, initially , $S$ set of nodes;
repeat
  {pick a connected component $S$ of $(V,X)$
  let $e$ be a lightest edge in $E$ that crosses
    between $S$ and $V-S$
  add $e$ to $X$}
until $|X|=|V|-1$
```

The several minimum spanning tree algorithms we shall see come from different choices of S in the fourth line: We can consistently choose the connected component containing some vertex s we have chosen arbitrarily (this is *Prim's algorithm*); or we can choose S so that the length of e is minimum (this is *Kruskal's algorithm*); or we could add to X all eligible edges at the same time (we call this is *the anything goes algorithm*).

2 Prim's algorithm:

Prim's algorithm "grows" the minimum spanning tree from a particular node s that we have chosen arbitrarily. We denote by S the set of nodes that has already been connected to s. In order to find the lightest edge crossing between S and $V-S$, Prim's algorithm maintains a heap containing all those vertices in $V-S$ which are adjacent to some vertex in S. The priority of a vertex v, according to which the heap is ordered, is the weight of its lightest edge to a vertex in S. This is reminiscent of Dijkstra's algorithm. As in Dijkstra's algorithm, each vertex v will also have a parent pointer prev(v) which is the other endpoint of the lightest edge from v to a vertex in S. The pseudocode for Prim's algorithm is identical to that for Dijkstra's algorithm, except for the definition particulars of the update operation:

```
Algorithm prim($G=(V,E,$length$)$: edge-weighted graph, $s \in V$);
$S$: set of nodes, initially $\emptyset$
dist: array[$V$] of integer, initially $\infty$
prev: array[$V$] of $V$, intilally nil.
$H$: heap of edges, prioritized by dist, initially $\{s\}$;
dist[$s$]:=0
while $H \neq \emptyset$ do
  {$u$:= deletemin($H$), add $u$ to $S$
  for each $v \in V-S$ do
    {if $w(u,v) <$ dist[$v$] then
    {dist[$w$]:=length($v, w$), prev[$w$]:=$v$, insert($v, H$)}}}
```

Note that each vertex can be inserted on the heap at most once, since as soon as it is removed from the heap it is inserted in S (intuitively, the set of nodes already connected in the minimum spanning tree under construction). The set S also plays the role of the set with the same name in the cut property stated above. The set X of edges chosen to be included in the minimum spanning tree is given by the `prev` pointers of the vertices in the set S. Since the smallest `dist` in the heap at any time gives the lighest edge crossing between S and $V - S$, Prim's algorithm follows the generic outline presented above, and therefore its correctness follows from the cut property.

Notice the similarity between Prim's algorithm and Dijkstra's algorithm. The only difference is that in Dijkstra's algorithm we prioritize nodes according to their distance from the *single node* s, while in Prim's algorithm we prioritize them according to their distance from the *set of nodes* S. For this reason, the running time of Prim's algorithm is clearly the same as Dijkstra's algorithm, since the only change is the definition of the key under which the heap is ordered. Thus, if we use d-heaps, the running time of Prim's algorithm is $O(m \log_{2+m/n} n)$.

3 Kruskal's algorithm:

Prim's algorithm treats the node s preferentially, and "grows the tree from it." Kruskal's algorithm has no such bias. It starts with the edges sorted in increasing order by weight. Initially $X = \emptyset$, and each vertex in the graph regarded as a trivial tree (with no edges). Each edge in the sorted list is examined in order, and if its endpoints are in the same tree, then the edge is discarded; otherwise it is included in X and this causes the two trees containing the endpoints of this edge to merge into a single tree.

To implement Kruskal's algorithm, given a forest of trees, we must decide given two vertices whether they belong to the same tree. For the purposes of this test, each tree in the forest can be represented by a set consisting of the vertices in that tree. We also need to be able to update our data structure to reflect the merging of two trees into a single tree. Thus our data structure will maintain a collection of disjoint sets (disjoint since each vertex is in exactly one tree), and support the following operations:

MAKESET(x): Create a new set, whose only element is x.

FIND(x): Given an element x, which set does it belong to?

UNION(x, y): replace the set containing x and the set containing y by their union.

The pseudocode for Kruskal's algorithm follows:

```
Algorithm kruskal($G = (V, E, w)$: edge-weighted graph);
$X$: set of edges, initially $\emptyset$
$E$:= sort $E$ in increasing length
for each $v \in V$ do {MAKESET($v$)}
for $[u, v] \in E$ (in inccreasing order) do
  {if FIND($u$) $\neq$ FIND($v$) then
    {add edge $[u, v]$ to $X$, UNION($u, v$)}}
```

The correctness of Kruskal's algorithm follows from the following argument: Kruskal's algorithm adds an edge e into X only if it connects two different connected components of (V, X) —otherwise, `FIND`(u) =`FIND`(v); let S be one of these two components. Then e must be the first edge in the sorted edge list that has one endpoint in S and the other endpoint in $V - S$, and is therefore the lightest edge that leaves S. Thus the cut property of minimum spanning tree implies the correctness of the algorithm.

The running time of the algorithm is dominated by the work needed to sort the edges, and by the set operations UNION and FIND. There are $V-1$ UNION operations (one corresponding to each edge in the spanning tree), and $2|E|$ FIND operations (two for each edge). Thus the total time of Kruskal's algorithm is $O(|E|\log|V| + |E| \times \texttt{FIND}) + |V| \times \texttt{UNION})$.[2] By the analysis in the next lecture, this is $O(|E|\log|V|)$.

4 The "anything goes" algorithm

Prim's algorithm implements the cut property by growing the minimum spanning tree from a distinguished node s. At each stage it selects the shortest edge that leaves the connected component of X —the tree so far— containing s. Kruskal's algorithm always selects the shortest edge leaving *any* of the components of X. Why can't we, at each stage, add the shortest edges out of *all* components of X, all at the same time?

As it turns out,

- This algorithm can be implemented in time $O(|E|\log|V|)$. Can you see how?
- This algorithm runs into trouble if many edges have the same weight. (Can you make up an example?) So, it is advisable that, before running this algorithm, the edge weights be *perturbed* a little so that each is unique.

5 The Exchange Property

Actually spanning trees satisfy an even stronger property than the cut property — the exchange property. The exchange property is quite remarkable since it implies that we can "walk" from any spanning tree T to a minimum spanning tree $\hat{T}$ by a sequence of exchange moves — each such move consists of throwing an edge out of the current tree that is not in $\hat{T}$, and adding a new edge into the current tree that is in $\hat{T}$. Moreover, each successive tree in the "walk" is guaranteed to weigh no more than its predecessor.

The exchange property: Let T and T' be spanning trees in $G(V,E)$. Given any $e' \in T' - T$, there exists an edge $e \in T - T'$ such that $(T - \{e\}) \cup \{e'\}$ is also a spanning tree.

The proof is quite similar to that of the cut property. Adding e' into T results in a unique cycle. There must be some edge in this cycle that is not in T' (since otherwise T' must have a cycle). Call this edge e. Then deleting e restores a spanning tree, since connectivity is not affected, and the number of edges is restored to $n-1$.

To see how one may use this exchange property to "walk" from any spanning tree to a minimum spanning tree: let T be any spanning tree and let $\hat{T}$ be a minimum spanning tree in $G(V,E)$. Let e' be the lightest edge that is not in both trees. Perform an exchange using this edge. Since the exchange was done with the lightest such edge, the new tree must be lighter than the old one. Since $\hat{T}$ is already a minimum spanning tree, it follows that the exchange must have been performed upon T and results in a lighter spanning tree which has more edges in common with $\hat{T}$ (if there are several edges of the same weight, then the new tree might not be lighter, but it still has more edges in common with $\hat{T}$).

[2] Recall that, since we assume that $|E|$ is between $|V|$ and $|V|^2$, there is no asymptotic difference between $\log|E|$ and $\log|V|$.