# Notes 8 for CS 170

## 1 Minimum Spanning Trees

A tree is an undirected graph that is connected and acyclic. It is easy to show (see Theorem 5.2 in CLR; B.2 of CLRS) that if graph $G(V, E)$ that satisfies any two of the following properties also satisfies the third, and is therefore a tree:

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

A spanning tree in a graph $G(V, E)$ is a subset of edges $T \subseteq E$ that are acyclic and connect all the vertices in $V$. It follows from the above conditions that a spanning tree must consist of exactly $n - 1$ edges. Now suppose that each edge has a weight associated with it: $w : E \to Z$. Say that the weight of a 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 $G(V, E)$ is one which has the smallest weight among all spanning trees in $G(V, E)$.

In general, the number of spanning trees in $G(V, E)$ grows exponentially in the number of vertices in $G(V, E)$. 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 by successively selecting edges to include in the tree. We will guarantee after the inclusion of each new edge that the selected edges, $X$, form a subset of some minimum spanning tree, $T$. How can we guarantee this if we don't yet know any minimum spanning tree in the graph? The following property provides this guarantee:

Lemma 1 (Cut Lemma)
*Let $X \subseteq T$ where $T$ is a MST in $G(V, E)$. Let $S \subset V$ such that no edge in $X$ crosses between $S$ and $V - S$; i.e., no edge in $X$ has one endpoint in $S$ and one endpoint in $V - S$. Among 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 in $G(V, E)$.*

Before proving the Cut Lemma, we need to make some observations about trees. Let $G = (V, E)$ be a tree, let $v, w$ be vertices such that $(v, w) \notin E$, and consider the graph obtained from $G$ by adding the edge $(v, w)$, that is, $G' = (V, E')$ where $E' = E \cup \{(u, v)\}$. Then

1. $G'$ has exactly one simple cycle
2. If we remove from $G'$ any of the edges on the unique cycle, we obtain a tree.

To prove part (1), notice that, since $G$ is connected, there is a path from $v$ to $w$, and this path, together with the edge $(v, w)$ gives a cycle. Now, every simple cycle in $G'$ must contain the edge $(v, w)$, and then a simple path in $G$ from $v$ to $w$, but in a tree there is only one simple path between two fixed vertices (if there were two simple paths with the same endpoints, their union would contain a cycle).

To prove part (2), let $v, v_1, \ldots, v_k, w, v$ be the unique cycle in $G'$, and let us call $v = v_0$ and $w = v_{k+1}$. Suppose we remove from $G'$ the edge $(v_i, v_{i+1})$, for some $i \in \{0, \ldots, k\}$, and let us call $G''$ the resulting graph. Then $G''$ is acyclic, because we have broken the unique cycle in $G'$, and it is connected, since $G'$ is connected, and any path in $G'$ that used the edge $(v_i, v_{i+1})$ can be re-routed via the path from $v_i$ to $v_{i+1}$ that still exists.

We can now prove the Cut Lemma.

PROOF: [Of Lemma 1] If $e \in T$, then we can set $T' = T$ and we are done.

Let us now consider the case $e \notin T$. Adding $e$ into $T$ creates a unique cycle. We will remove a single edge $e'$ from this unique cycle, thus getting $T' = (T \cup \{e\}) - \{e'\}$ which, by the above analysis is again a tree.

We will now show that 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 our minimum spanning tree algorithms is going to be the following:

$X := \{\ \}$ ... $X$ contains the edges of the MST
Repeat until $|X| = n - 1$
  Pick a set $S \subseteq V$ such that no edge in $X$ crosses between $S$ and $V - S$
  Let $e$ be a lightest edge in $G(V, E)$ that crosses between $S$ and $V - S$
  $X := X \cup \{e\}$

We will now describe two implementations of the above general procedure.

## 2 Prim's algorithm:

In the case of Prim's algorithm, $X$ consists of a single tree, and the set $S$ is the set of vertices of that tree. 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 key 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 $S$. Notice that the pseudocode for Prim's algorithm is identical to that for Dijkstra's algorithm, except for the definition of the key under which the heap is ordered:

```
algorithm Prim(weighted graph G=(V, E))
initialize empty priority queue H
for all v ∈ V do
  key[v] = ∞; prev[v] =nil
pick an arbitrary vertex s
H={s}; key[s] =0; mark(s)
while H is not empty do
  v := deletemin(H)
  mark(v)
  for each edge (v,w) in E out of v do
    if w unmarked and key[w] > weight[v,w] then
      key[w] := weight[v,w]; prev[w] = v; insert(w,H)
```

As in Dijkstra's algorithm, `insert(w,H)` really means to insert only if $w \notin H$, and to update $w$'s priority `key` in $H$ otherwise.

The complexity analysis of Prim's algorithm is identical to Dijkstra: each vertex and each edge is processed once, so the cost is $|V|\cdot$Cost(deletemin) + $|E|\cdot$Cost(insert).

The vertices that are removed from the heap form the set $S$ in the cut property stated above. The set $X$ of edges chosen to be included in the MST are given by the parent pointers `prev` of the vertices in the set $S$. Since the smallest key in the heap at any time gives the lightest edge crossing between $S$ and $V - S$, Prim's algorithm follows the generic outline for a MST algorithm presented above, and therefore its correctness follows from the cut property.

## 3 Kruskal's algorithm

Kruskal's algorithm starts with the edges sorted in increasing order by weight. Initially $X = \{\ \}$, 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. Thus, $X$ consists of a forest of trees, and edges are added until it consists of exactly one tree, a MST. At each step $S$ consists of the endpoints of vertices of one tree in $X$, the tree which contains one endpoint of the chosen edge.

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 two operations:

- `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 data structure is constructed with the operation `makeset(x)`, that adds to the data structure a set that contains the only element $x$.

We will discuss the implementation of `find` and `union` later. The pseudocode for Kruskal's algorithm follows:

```
algorithm Kruskal(weighted graph G(V,E))
  X = { }
  sort E by weight
  for u ∈ V
      makeset(u)
  for (u,v) ∈ E in increasing order by weight
      if find(u) ≠ find(v) do
         X = X ∪ {(u,v)}
         union(u,v)
  return(X)
end
```

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 trees; let $S$ be the set of vertices in one of these two trees. 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 crosses between $S$ and $V - S$. Thus the cut property of MST implies the correctness of the algorithm.

The running time of the algorithm is dominated by the set operations `union` and `find` and by the time to sort the edge weights. There are $n - 1$ `union` operations (one corresponding to each edge in the spanning tree), and $2m$ `find` operations (2 for each edge). Thus the total time of Kruskal's algorithm is $O(m \times FIND + n \times UNION + m \log m)$. This will be seen to be $O(m \log n)$.

## 4 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.

Lemma 2 (Exchange Lemma)
*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.*

PROOF: [Sketch] The proof is quite similar to that of the Cut Lemma. 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 $n-1$. $\square$

To see how one may use this exchange property to "walk" from any spanning tree to a MST: let $T$ be any spanning tree and let $\hat{T}$ be a MST 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 at least as light as the old one. Since $\hat{T}$ is already a MST, 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 $\hat{T}$).