CS125	Lecture 2	Fall 2015

2.1 Greedy Algorithms

We will start talking about methods – high-level plans – for constructing algorithms. One of the simplest is just to have your algorithm "be greedy". Being greedy, unsurprisingly, doesn't always work, but when it does, it can lead to very intuitive, natural, and fast algorithms. Here we'll look at the greedy paradigm in the context of building minimum spanning trees.

2.2 Minimum Spanning Trees

A tree is an undirected graph which is connected and acyclic. It is easy to show 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
- $|E| = |V| - 1$

(**Exercise:** Show that any two of the above properties implies the third (use induction).)

A *spanning tree* in an undirected 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 $G(V,E)$.

As an example of why one might want to find a minimum spanning tree, consider someone who has to install the wiring to network together a large computer system. The requirement is that all machines be able to reach each other via some sequence of intermediate connections. By representing each machine as a vertex and the cost of wiring two machines together by a weighted edge, the problem of finding the minimum cost wiring scheme reduces to the minimum spanning tree problem.

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.

Exercise: Try to determine the number of different spanning trees for a complete graph on n vertices.)

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:

Cut property: 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 $G(V,E)$.

The cut property says that we can construct our tree *greedily*. Our greedy algorithms can simply take the minimum weight edge across two regions not yet connected. Eventually, if we keep acting in this greedy manner, we will arrive at the point where we have a minimum spanning tree. Although the idea of acting greedily at each point may seem quite intuitive, it is very unusual for such a strategy to actually lead to an optimal solution, as we will see when we examine other problems!

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 our minimum spanning tree algorithms is going to be the following:

$X := \{\ \}$.

```
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 $V-S$.
    $X := X \cup \{e\}$.
```

The difference between minimum spanning tree algorithms lies in how we pick the set S at each step.

2.3 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. One way to think of the algorithm is that it grows a single tree, adding a new vertex at each step, until it has the minimum spanning tree. In order to find the lightest edge crossing between S and $V-S$, Prim's algorithm maintains a heap (described below) 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. 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.

```
Procedure Prim($G(V,E)$, $s$)
    $v, w$: vertices
    dist: array$[V]$ of integer
    prev: array$[V]$ of vertices
    S: set of vertices, initially empty
    $H$: priority heap of $V$
    $H := \{s : 0\}$
    for $v \in V$ do
        dist$[v] := \infty$, prev$[v]$ :=nil
    rof
    dist$[s] := 0$
    while $H \neq \emptyset$
        $v :=$ deletemin$(h)$
        $S := S \cup \{v\}$
        for $(v,w) \in E$ and $w \in V-S$ do
            if dist$[w] >$ length$(v,w)$
                dist$[w] :=$ length$(v,w)$, prev$[w] := v$, insert($w$,dist$[w]$,$H$)
            fi
        rof
    end while end Prim
```

Note that each vertex is "inserted" on the heap at most once; other insert operations simply change the value on the heap. The vertices that are removed from the heap form the set S for the cut property. The set X of edges chosen to be included in the MST are given by the parent pointers 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.

The running time of Prim's algorithm depends on how one implements the priority queue. A standard approach – heaps – will be gone over in section, and notes will be posted.

2.4 Kruskal's algorithm:

Kruskal's algorithm uses a different strategy from Prim's algorithm. Instead of growing a single tree, Kruskal's algorithm attempts to put the lightest edge possible in the tree at each step. 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. Note that, by this process, we are *implicitly* choosing a set $S \subseteq V$ with no edge in X crossing between S and $V-S$, so this fits in our basic outline of a minimum spanning tree algorithm.

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 three operations:

- MAKESET(x): Create a new x containing only the element 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:

```
Function Kruskal(graph $G(V,E)$)
  set $X$
  $X = \{\ \}$
  $E$:= sort $E$ by weight
  for $u \in V$
      MAKESET(u)
  rof
```

```
    for $(u,v) \in E$ (in increasing order) do
        if $FIND(u) \neq FIND(v)$ do
            $X = X \cup \{(u,v)\}$
            UNION($u$,$v$)
    rof
    return($X$)
end Kruskal
```

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, assuming the edges are given in sorted order, is dominated by the set operations: UNION and FIND. 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 $O(m \times FIND + n \times UNION)$. We will soon show that this is $O(m \log^* n)$. Note that, if the edges are *not* initially given in sorted order, then to sort them in the obvious way takes $O(m \log m)$ time, and this would be the dominant part of the running time of the algorithm.

2.5 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.

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 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 lighter than 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 $\hat{T}$).

2.6 Basics of Heaps

A heap is an data structure that keeps a set of objects, where each object has an associated value. The operations a heap H implements include the following:

deletemin(H)	return the object with the smallest value
insert(x, y, H)	insert a new object x/value y pair in the structure
change(x, y, H)	if y is smaller than x's current value, change the value of object x to y

We will not distinguish between insert and change, since for our purposes, they are essentially equivalent; changing the value of a vertex will be like re-inserting it. [1]

What is the running time of Prim's algorithm? The algorithm involves $|E|$ insert operations and $|V|$ deletemin operations on H, and so the running time depends on the implementation of the heap H. There are many ways to implement a heap. Even an unsophisticated implementation as a linked list of node/priority pairs yields an interesting time bound, $O(|V|^2)$ (see first line of the table below). A binary heap would give $O(|E|\log|V|)$.

Which of the two should we prefer? The answer depends on how *dense* or *sparse* our graphs are. In all graphs, $|E|$ is between $|V|$ and $|V|^2$. If it is $\Omega(|V|^2)$, then we should use the linked list version. If it is anywhere below $\frac{|V|^2}{\log|V|}$, we should use binary heaps.

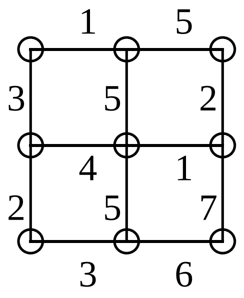
heap implementation	deletemin	insert	$\lvert V\rvert\times$deletemin$+\lvert E\rvert\times$insert
linked list	$O(\lvert V\rvert)$	$O(1)$	$O(\lvert V\rvert^2)$
binary heap	$O(\log\lvert V\rvert)$	$O(\log\lvert V\rvert)$	$O(\lvert E\rvert\log\lvert V\rvert)$
d-ary heap	$O(\frac{d\log\lvert V\rvert}{\log d})$	$O(\frac{\log\lvert V\rvert}{\log d})$	$O((\lvert V\rvert\cdot d+\lvert E\rvert)\frac{\log\lvert V\rvert}{\log d})$
Fibonacci heap	$O(\log\lvert V\rvert)$	$O(1)$ amortized	$O(\lvert V\rvert\log\lvert V\rvert+\lvert E\rvert)$

[1] In all heap implementations we assume that we have an array of pointers that gives, for each vertex, its position in the heap, if any. This allows us to always have at most one copy of each vertex in the heap. Furthermore, it makes changes and inserts essentially equivalent operations.

A more sophisticated data structure, the d-ary heap, performs even better. A d-ary heap is just like a binary heap, except that the fan-out of the tree is d, instead of 2. (Here d should be at least 2, however!) Since the depth of any such tree with $|V|$ nodes is $\frac{\log|V|}{\log d}$, it is easy to see that inserts take this amount of time. Deletemins take d times that, because deletemins go down the tree, and must look at the children of all vertices visited.

The complexity of this algorithm is a function of d. We must choose d to minimize it. A natural choice is $d = \frac{|E|}{|V|}$, which is the the average degree! (Note that this is the natural choice because it equalizes the two terms of $|E| + |V| \cdot d$. Alternatively, the "exact" value can be found using calculus.) This yields an algorithm that is good for both sparse and dense graphs. For dense graphs, its running time is $O(|V|^2)$. For graphs with $|E| = O(|V|)$, it is $|V|\log|V|$. Finally, for graphs with intermediate density, such as $|E| = |V|^{1+\delta}$, where δ is the *density* of the graph, the algorithm is *linear!*

There are other heap variations that we will not cover here. For example, there is a data structure known as a Fibonacci heap, which has the property the bounds for the insert operation for Fibonacci heaps are amortized bounds: certain operations may be expensive, but the average cost over a sequence of operations is constant. We will be looking at such amortized bounds for the data structure for Kruskal's algorithm more closely.


1
5
3
5
2
4
1
2
5
7
3
6