CSE341T/CSE549T **10/27/2014**

Minimum Spanning Tree and Maximal Independent Set

Lecture 16

1 Minimum Spanning Tree

The minimum (weight) spanning tree (MST) problem is given an *connected* undirected graph $G = (V, E)$, find a spanning tree of minimum weight (i.e. sum of the weights of the edges). That is, find a spanning tree T that minimizes $w(T) = \sum_{e \in E(T)} w_e$.

In previous classes you have seem Kruskal's algorithm:

```
sort edges by weight
put each vertex in its own component
for each edge e = (u,v) in order or weight
   if u and v are in the same component skip
   else join the components for u and v and add e to the MST
   if(|T| = |V|) break;
```

You used a union-find data structure to detect when two vertices are in the same component and join them if not. With an efficient implementation of union-find, the work for the algorithm is dominated by the need to sort the edges. The algorithm therefore runs in $O(|E| \lg |V|)$ work. It is fully sequential.

You may have also seen Prim's algorithm:

```
Start with an arbitrary source vertex in the graph, and maintain
    a set of visited vertices X.
In each step, grow the tree maintained by X:
   add to X the vertex that is adjacent to the tree maintained
      by X that:
   1. has the smallest weight, and
   2. would maintain the tree property.
Repeat until all vertices have been added.
```

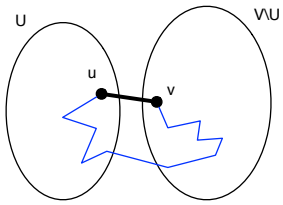
The Prim's algorithm grows a tree (maintained by set X) starting from some arbitrarily chosen vertex, and only adds an edge e if e has the smallest weight leaving the tree maintained by X and if adding e does not break the tree property of set X. To select the minimum weight edge leaving X on each step, it stores all edges leaving X in a priority queue and adds the edge only if the other

end point is not in X already. The algorithm is almost identical to Dijkstra's algorithm but instead of storing distances in the priority queue, it stores edge weights. Assuming you use a binary heap, it's asymptotically the same as Kruskal's (but you can do a little better if you use a Fibonacci heap), and it is also sequential.

At first glance, these two algorithms are quite different, but they both reply on the same underlying principles about "cuts" in a graph, which we'll refer to as the ***light edge property***. Here we will assume without any loss of generality that all edges have distinct weights. This is easy to do since we can break ties in a consistent way. For a graph $G = (V, E)$, a ***cut*** is defined in terms of a subset $U \subsetneq V$. This set U partitions the graph into $(U, V \setminus U)$, and we refer to the edges between the two parts as the cut edges $E(U, \overline{U})$, where as is typical in literature, we write $\overline{U} = V \setminus U$. The subset U might include a single vertex v, in which case the cut edges would be all edges incident on v. But the subset U must be a proper subset of V (i.e., $U \neq \emptyset$ and $U \neq V$).

Theorem 1 (The Light Edge Property) *Let $G = (V, E, w)$ be a connected undirected weighted graph with distinct edge weights. For any nonempty $U \subsetneq V$, the minimum weight edge e between U and $V \setminus U$ is in the minimum spanning tree MST(G) of G.*

Proof. The proof is by contradiction. Assume the minimum-weighted edge $e = (u, v)$ is not in the MST. Since the MST spans the graph, there must be some simple path P connecting u and v in the MST (i.e., consisting of just edges in the MST). The path must cross the cut between U and $V \setminus U$ at least once since u and v are on opposite sides. By attaching P to e, we form a cycle (recall that by assumption $e \notin MST$). If we remove the maximum weight edge from P and replace it with e we will still have a spanning tree, but it will be have less weight. This is a contradiction. □



Note that the last step in the proof uses the facts that (1) adding an edge to a spanning tree creates a cycle, and (2) removing any edge from this cycle creates a tree again.

If you think about it, both Kruskal's and Prim's algorithms work under this principle. For Kruskal's, every edge that we add to the MST is a lightest weight edge remaining that would connect two distinct components. This edge must be in the MST by the Light Edge Property. For Prim's, the algorithm maintains a visited set X, which also corresponds to the set U in the cut. At each step, it selects the minimum weigh edge $e = (u, v), u \in X, v \in V \setminus X$. Again, this edge must be in the MST by the Light Edge Property.

There is yet another MST algorithm that you may not have heard of, called Borůvka, which actually predates both Kruskal's and Prim's (in 1926, before computers are even invented!). Again, it exploits the Light Edge Property, and the main observation is that, **the minimum weight edge out of every vertex of a weighted graph G belongs to its MST**. This observation follows directly from the Light Edge Property, if you consider each vertex as its own set U.

The following example illustrates this situation, where we have highlighted the minimum weight edges. Note that some edges are minimum for both of their endpoints (i.e., the ones weighted 1 and 4).



Borůvka's algorithm throws all the minimum weight edges coming out of each vertex into the MST, contracts these minimum weight edges, and repeats until no more edges left. Here, we show a parallel based on an approach similar to Borůvka's algorithm.

We can modify the contract routine to only consider contracting the set of edges that are a subset of E that are the minimum edge weight coming out of each vertex, as follows:

MINSTARCONTRACT(V, E, T)

1 Let $minE$ be the set of all minimum-weight edge coming out of each attached vertex in G.
2 Flip a coin to label each vertex, $L(u) \in \{head, tail\}$.
3 Select vertex u to contract into v such that $L(v) = head$ and $L(u) = tail$, and $(u, v) \in minE$. Mark this edge (u, v) as being part of MST T.
4 Remove vertices that got contracted in step 3.
5 Create E' by relabeling E (i.e., rewrite vertex u that got contracted into v in step 3 as v).
6 Mark the contracted edges in T using its unique label as being in MST.
7 Relabel edges to create E': for every vertex u contracted into v, change u to v in the adjacency list.
8 Remove self loops in E'.
9 **return** (V', E', T)

There is a little bit of trickiness since as the graph contracts, the endpoints of each edge changes. Therefore, if we want to return the edges of the minimum spanning tree, they might not correspond to the original endpoints. Also, we need to somehow avoid races on T in step 3. To deal with the first issue, we can simply associate a unique label with every edge. The end point of edges can change during contractions, but the edge labels will remain the same. To deal with the second issue, we simply store T as an edge array, mapping edge labels (the array index) to the edges in the original graph G. To indicate that an edge should be included in T, we simply look up in the array using the edge label and mark the edge as included.

Analysis of this algorithm is very similar to the SCC algorithm. Note that even though we only consider a subset of edges, the expected number of vertices removed is still $n/4$, because at each step, every attached vertex v has an edge in $minE$, and the probability that v gets contracted is still $1/4$.

2 Maximal Independent Set

Independent set of vertices is a set of vertices such that the set does not contain both a vertex and its neighbor. A maximum independent set is the largest such set. A maximal independent set is an independent set that can not be further increased in size.

Finding the maximum independent set is NP-complete. Can you tell me how to find a maximal independent set?

```
Pick an arbitrary vertex u and put it in IS
remove u and all its neighbors from consideration and remove all edges
that are incident on these vertices.
repeat until all vertices are gone.
```

How do we parallelize it? Instead of just removing a single vertex and its neighbors on each iteration, we can try to remove set S of nodes and all the neighbors $N(S)$.

We want to use graph contraction, and at every step we want to reduce the size of the graph by a constant factor. So in principle, we want to pick S such that $S \cup N(S)$ is large. However, this turns out to be difficult. So instead, we will try to pick nodes S such that a large number of edges are incident on $N(S)$ and we get to remove all those edges.

- For each vertex u label the vertex u such that $L(u) = T$ with probability $1/(2d(u))$ where $d(u)$ is the degree of u and H otherwise. If the degree of a vertex is 0, definitely mark it as T.
- For each edge (u, v) if both u and v are marked T then unmark (mark as H) the vertex with the lower degree.
- If $L(u) = T$ then put u in MIS and remove all such u and their neighbors from v and remove all edges incident on these vertices.
- Repeat.

Each round takes $O(m + n)$ work and $O(\lg^2 n)$ span. Removing marked vertices and edges requires parallel prefix. If we assume that each round removes a constant fraction of the edges, then the total work is $O(m + n)$ and the total span is $O(\lg^3 n)$. In the next lecture, we will prove that each round removes a constant fraction of the edges.