Lecture 15 Minimum Spanning Trees

Input: A connected undirected graph $G = (V, E)$ with edge weights.

What we seek: a subset $T \subset E$ such that
(1) $G' = (V, T)$ is acyclic and connected
and (2) $\sum_{e \in T} w(e)$ is minimum subject to condition (1).

[We assume edge weights are given by the function $w : E \to \mathbb{R}$.]

Such a set T is called a minimum spanning tree. (MST)

We will see a simple greedy algorithm for this problem.
- our algorithm has to make a choice in each step and a greedy algorithm makes the choice that looks best at the moment.

In general, a greedy strategy is not guaranteed to find an optimal solution. In MST algorithms, certain greedy strategies work.

Greedy algorithm

Invariant: maintain a subset $A \subseteq E$ such that $A \subseteq$ some MST

1. $A = \emptyset$
2. while $|A| < n-1$ do
   {
   - find an edge (u, v) that is safe for A.
   - $A = A \cup \{(u,v)\}$
   }
3. Return A.

An edge (u, v) is safe for A if $A \cup \{(u,v)\} \subseteq$ some MST.

Question: How do we find a safe edge?

Since $|A| < n-1$, there is a cut $(S, V-S)$ such that no edge of A crosses this cut. Let (u, v) be a minimum weight edge crossing this cut.

Claim. $A \cup \{(u,v)\} \subset$ some MST.

Proof. Before we added the edge (u, v) to A, we had $A \subseteq$ some MST. Call this tree T_1. Suppose $(u,v) \in T_1$. Then $A \cup \{(u,v)\} \subseteq T_1$.

So let us assume that $(u, v) \notin T_1$. Since T_1 is a connected graph, T_1 has to contain some edge (x, y) crossing this cut $(S, V-S)$.

Let $T_2 = T_1 - (x, y) + (u, v)$.
We claim T_2 is an MST. Observe that T_2 has no cycle - this is because T_1 has a unique $u-v$ path and deleting (x, y) from T_1 puts u and v in different components. Adding the edge (u, v) joins these 2 components again.

Since (u, v) is a minimum weight edge crossing $(S, V-S)$, we have $w(u,v) \leq w(x,y)$. So $w(T_2) \leq w(T_1)$. This means $w(T_2) = w(T_1)$ since T_1 is an MST.

Thus (u, v) is safe for A: we have $A \cup \{(u,v)\} \subset T_2$, which is an MST. $\square$

We will now see Prim's algorithm which is the above greedy algorithm where a safe edge added to A is described on the next page.

In Prim's algorithm, the edges in A span a single component. The safe edge added to A is a minimum weight edge joining some vertex in V−A to a vertex in A.

Prim's algorithm operates much like Dijkstra's algorithm.

- The tree starts from an arbitrary root vertex (call it r) and grows until the tree spans all the vertices in V.
- Let S be the set of vertices already connected by edges in A to r. Add the light weight edge crossing (S, V−S) to A.

this is a min weight edge crossing this cut

An efficient way to determine the light weight edge crossing (S, V−S): use an F-heap.

MST - Prim

1. Select any vertex as the root r.
2. Set $key[r] = 0$ and $key[u] = \infty \ \forall u \in V - \{r\}$.
3. Set $\pi[u] = nil \ \forall u$.
4. $Q = V$
5. while $Q \neq \phi$ do

   {

   - u = extract-min(Q)
   - for all $v \in Q$ that are adjacent to u do:
     - if $key[v] > key[u]$ then
       * set $key[v] = w(u, v)$
       * $\pi[v] = u$

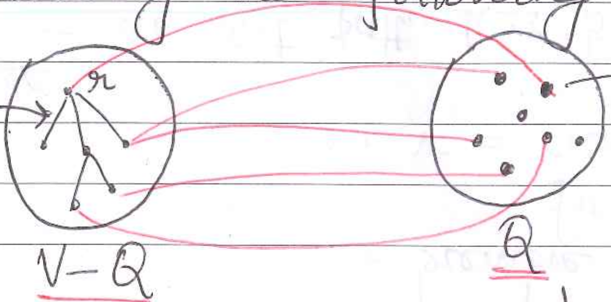
   }

6. Return the array π.

Is the correctness of the above algorithm easy to show?

Prim's algorithm is a particular implementation of the greedy algorithm where in every iteration, the algorithm selects a min-weight edge crossing the following cut:

Edges added to our MST



Each vertex here has a key value - the weight of the lightest edge incident to this vertex with the other endpoint in V−Q. This is the red edge in the figure.

The correctness of Prim's algo. follows from the correctness of the generic greedy algorithm.

Running time of Prim's algorithm: This involves n extract-min operations and $\leq m$ decrease-key operations. These operations can be implemented in $O(m + n\log n)$ time using an F-heap.

Kruskal's algorithm: This is another classical MST algorithm — here we grow a forest. We find a safe edge to add to the growing forest by finding among all edges joining 2 distinct components, an edge (u, v) of least weight.

MST - Kruskal (G)

1. Initialize $A = \phi$.
2. Sort the edges in increasing order of weight.
   - call the edges $\underbrace{e_1, \ldots, e_m}_{\text{this is the sorted order.}}$
   - let $i = 1$.

3. while $|A| < n-1$ do

{

- let u and v be the endpoints of e_i.
- if $FIND(u) \neq FIND(v)$ then
  - $A = A \cup \{(u,v)\}$
  - Union $COMP(u)$ and $COMP(v)$.
- $i = i + 1$

4. Return A.

In the above algorithm, each vertex has an identity called set number which is distinct for each component.

- if $FIND(u) \neq FIND(v)$ then

u and v are in different components.

Once we add (u,v) to A, we will change the set number of all vertices in $COMP(u)$ to the set number of v or vice-versa.

Simple Approach: Keep an array of vertices. Each vertex stores its set number in the array. FIND takes $O(1)$ time and Union takes $O(\log n)$ amortized time.

Union: Change the set number of the smaller set. So if a set number is changed i times, then this vertex is in a set of size $\geq 2^i$.

So any element's set number is changed at most $\log n$ times. Give each vertex $\log n$ credit points to begin with and these $n \log n$ credit points are enough to pay for n union operations.

The ~~entire~~ running time of Step 3 is $O(m + n\log n)$.

Let us forget the sorting time for now (this is Step 2) and focus on the data structure problem (this is Step 3).

- there are n elements and the set of elements is partitioned into components: given a pair of elements (u, v), we want to determine if u and v are in the same component or in different components. If they are in different components, then we want to merge these 2 components into the same component.

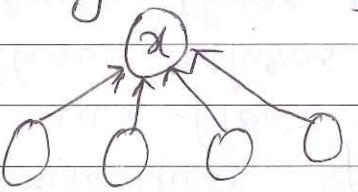
The 2 operations that we have here are:

$\underbrace{\text{FIND}(x)}_{m \text{ such operations}}$ and $\underbrace{\text{Union}(C, C')}_{n-1 \text{ such operations}}$

We just saw a solution that performs FIND in $O(1)$ time and Union in $O(\log n)$ amortized time.

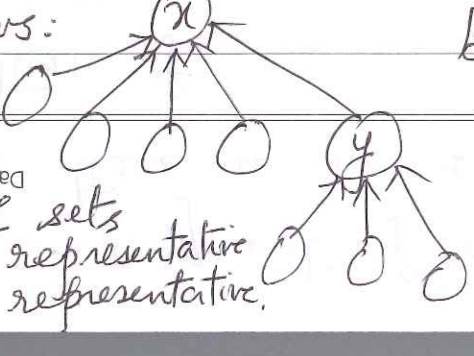
- Can we perform Union more efficiently?

Nodes in a set point to a common location containing the representative set element.



When we merge 2 sets, we can do this in $O(1)$ time

as follows:



This is the Union of sets with x as representative and y as representative.

But this method leads to trees with increased depth, so the time for FIND goes up.