Kruskal Algorithm.¹

Overview

In this lecture we describe the union-find problem. This is a problem that captures the key task we had to solve in order to efficiently implement Kruskal’s algorithm. We then give a data structure for it with good amortized running time.

Motivation

To motivate the union-find problem, let’s recall Kruskal’s minimum spanning tree algorithm.

Kruskal’s Algorithm (recap):
Sort edges by length and examine them from shortest to longest. Put each edge into the current forest if it doesn’t form a cycle with the edges chosen so far.

We argued correctness last time. Today, our concern is running time. The initial step takes time $O(|E| \log |E|)$ to sort. Then, for each edge, we need to test if it connects two different components. If it does, we will insert the edge, merging the two components into one; if it doesn’t (the two endpoints are in the same component), then we will skip this edge and go on to the next edge. So, to do this efficiently we need a data structure that can support the basic operations of (a) determining if two nodes are in the same component, and (b) merging two components together. This is the union-find problem.

The Union-Find Problem

The general setting for the union-find problem is that we are maintaining a collection of disjoint sets $\{S_1, S_2, \ldots, S_k\}$ over some universe, with the following operations:

$\text{MakeSet}(x)$: create the set $\{x\}$.

¹These lecture notes are due to Avrim Blum.
**Union**$(x, y)$: replace the set $x$ is in (let’s call it $S$) and the set $y$ is in (let’s call it $S'$) with the single set $S \cup S'$.

**Find**$(x)$: return the unique ID for the set containing $x$ (this is just some representative element of this set).

Given these operations, we can implement Kruskal’s algorithm as follows. The sets $S_i$ will be the sets of vertices in the different trees in our forest. We begin with MakeSet$(v)$ for all vertices $v$ (every vertex is in its own tree). When we consider some edge $(v, w)$ in the algorithm, we just test whether Find$(v)$ equals Find$(w)$. If they are equal, it means that $v$ and $w$ are already in the same tree so we skip over the edge. If they are not equal, we insert the edge into our forest and perform a Union$(v, w)$ operation. All together we will do $|V|$ MakeSet operations, $|V| - 1$ Unions, and $2|E|$ Find operations.

**Notation and Preliminaries:** in the discussion below, it will be convenient to define $n$ as the number of MakeSet operations and $m$ as the total number of operations (this matches the number of vertices and edges in the graph up to constant factors, and so is a reasonable use of $n$ and $m$). Also, it is easiest to think conceptually of these data structures as adding fields to the items themselves, so there is never an issue of “how do I locate a given element $v$ in the structure?”.

### Data Structure 1 (list-based)

The data structure that we discussed in class is a simple one with a very cute analysis. The total cost for the operations will be $O(m + n \log n)$.

In this data structure, the sets will be just represented as linked lists: each element has a pointer to the next element in its list. However, we will augment the list so that each element also has a pointer directly to head of its list. The head of the list is the representative element.

We can now implement the operations as follows:

**MakeSet**$(x)$: just set $x->\text{head}=x$. This takes constant time.

**Find**$(x)$: just return $x->\text{head}$. Also takes constant time.

**Union**$(x, y)$: To perform a union operation we merge the two lists together, and reset the head pointers on one of the lists to point to the head of the other.

Let $A$ be the list containing $x$ and $B$ be the list containing $y$, with lengths $L_A$ and $L_B$ respectively. Then we can do this in time $O(L_A + L_B)$ by appending $B$ onto the end of $A$ as follows. We first walk down $A$ to the end, and set the final next pointer to point to $y->\text{head}$. This takes time $O(L_A)$. Next we go to $y->\text{head}$ and walk down $B$, resetting head pointers of elements in $B$ to point to $x->\text{head}$. This takes time $O(L_B)$.
Can we reduce this to just $O(L_B)$? Yes. Instead of appending $B$ onto the end of $A$, we can just splice $B$ into the middle of $A$, at $x$. I.e., let $z=x\rightarrow\text{next}$, set $x\rightarrow\text{next}=y\rightarrow\text{head}$, then walk down $B$ as above, and finally set the final $\text{next}$ pointer of $B$ to $z$.

Can we reduce this to $O(\min(L_A, L_B))$? Yes. Just store the length of each list in the head. Then compare and insert the shorter list into the middle of the longer one. Then update the length count to $L_A + L_B$.

We now prove this simple data structure has the running time we wanted.

**Theorem 1** The above algorithm has total running time $O(m + n \log n)$.

**Proof:** The Find and MakeSet operations are constant time so they are covered by the $O(m)$ term. Each Union operation has cost proportional to the length of the list whose head pointers get updated. So, we need to find some way of analyzing the total cost of the Union operations.

Here is the key idea: we can pay for the union operation by charging $O(1)$ to each element whose head pointer is updated. So, all we need to do is sum up the costs charged to all the elements over the entire course of the algorithm. Let’s do this by looking from the point of view of some lowly element $x$. Over time, how many times does $x$ get walked on and have its head pointer updated? The answer is that its head pointer is updated at most $\log n$ times. The reason is that we only update head pointers on the smaller of the two lists being joined, so every time $x$ gets updated, the size of the list it is in at least doubles, and this can happen at most $\log n$ times. So, we were able to pay for unions by charging the elements whose head pointers are updated, and no element gets charged more than $O(\log n)$ total, so the total cost for unions is $O(n \log n)$, or $O(m + n \log n)$ for all the operations together.

Recall that this is already low-order compared to the $O(m \log m)$ sorting time for Kruskal’s algorithm.

We can do even better via a more sophisticated data-structure.