The Union-Find Problem

Kruskal's algorithm for finding an MST presented us with a problem in data-structure design. As we looked at each edge, cheapest first, we had to determine whether its two endpoints were connected by the edges we had added to the tree so far.

Remember that on a graph with n nodes and e edges, the Kruskal algorithm took $O(e \log e)$ steps to sort the edges by cost, $O(e)$ steps to decide whether each edge should be added ($O(1)$ steps for each edge) and $O(n^2)$ time to maintain a table telling us which nodes were connected to which. The total time was $O(n^2 + e \log e)$, which for sparse graphs is $O(n^2)$.

Thus keeping track of the components is the bottleneck step in the algorithm in the general case, as it is what is stopping us from running the algorithm in $O(e \log e)$ steps.

The problem of maintaining the connected components of the growing forest in Kruskal's algorithm is a special case of a more general problem called the **union-find problem**. In this lecture we'll look at a series of increasingly efficient implementations of data structures that solve the union-find problem. We'll fairly quickly see how to improve the performance to make the Kruskal running time $O(e \log e)$, but the later solutions are of interest for other applications, as well as theoretically.

Unlike our "improved" algorithms for integer multiplication and matrix multiplication, these improved data structures are simple and universally practical. The mathematical challenge is *analyzing* them to see exactly how fast they are.

A **union-find data structure** maintains a family of disjoint nonempty sets. Each set in the family has an element called its **label**. The data structure supports the following three operations:

- MAKESET(x) creates a new set $\{x\}$ and adds it to the family. The element x must not be an element of any existing set in the family.
- UNION(x, y) changes the family by replacing two sets, the one containing x and the one containing y, by a single set that is the union of these two sets. It is a no-op if x and y are already in the same set.
- FIND(x) returns the label of the set containing x.

Note that the changes in the family of sets are monotone. Once an element appears it remains in some set in the family, and once two elements are in the same set they stay in the same set.

It is easy to see how to use such a data structure to implement Kruskal's algorithm. We first perform a MAKESET for each vertex of the graph. Then for each edge (u, v) we do a FIND for both u and v, followed by UNION(u, v) if they are not already in the same set.

Review of Our First Solution

Our earlier implementation of Kruskal's algorithm used what was in effect a simple implementation of a union-find data structure. Let's put it into our new terminology:

- We maintain a table with each element and the label of its set.
- A MAKESET is performed by adding a new entry to the table, whose label is itself. (We are not required to check whether the element is already there.)
- The operation UNION(x, y) operation is performed by sweeping the table, changing the label of x, and of all entries with the same label as x, to the label of y. (Or we could change the y labels to the x labels, as long as we are consistent.)
- For a FIND(x) operation we simply return the label of the entry x. (We are not charged for any search for x in the table – we assume that we are given a pointer to it.)

A First Improvement

This method takes $O(n)$ time for each UNION operation, where n is the number of elements in all the sets, and $O(1)$ for each MAKESET or FIND. In the Kruskal setting we needed $n-1$ UNION operations and $O(e)$ FIND operations, costing us $O(n^2+e)$ for the data structure operations or $O(n^2+e\log e)$ when we include the initial sort of the edges.

How can we improve this? It seems wasteful to conduct a linear search through the table to update the labels for the UNION. What if we keep each set in a linked list rather than keeping all of them in a table? We make a list node for each element and give the node two pointers – one to the label (at the head of the list) and one to the next element of the set.

Now a MAKESET is still $O(1)$ (create a new node with a label pointer to itself and a null next pointer) and a FIND is $O(1)$ as well (just return the label pointer of the desired node). How can we implement a UNION?

To implement UNION(x, y) we must:

- locate the heads of the two lists and the tail of x's list
- update the next pointer of x's tail to y's head
- change the label pointer of each node in y's list to x's head

How long does this take? The list heads are given to us by pointers from x and y, and we can keep a single pointer from the head to the tail of each list. The appending is then $O(1)$ time, but we have to change a separate pointer for each node in y's list. This could certainly be $O(n)$ in the worst case.

But we noted before that we could either merge x's set into y's or *vice versa*. Does it make a difference?

The preponderence of the time taken by our UNION operation is needed to change the labels of the elements of y's set. What if we keep track of the size of each set, and always merge the *smaller* set into the *larger*? It seems at though this would save at least some time.

This doesn't help us in the worst case, because if our last merge happens to deal with sets of roughly equal size, we will need $\Theta(n)$ time to do it. But what we really care about is the *total* time to perform a *series* of operations.

Determining this total time is our first example of **amortized analysis**. By breaking up the set of operations in a different way, we can prove a bound on its total size. Let's consider the total time to perform $n-1$ UNION operations. The bulk of the time spent is proportional to the numer of labels changed. Rather than divide the label changes into phases for each UNION, we group together all the times that the label of a *given element* is changed.

Look at one particular element x. It begins life in a set of size 1. When its label is changed, it must go into a set of size at least 2. The next time it is changed, this set of size at least 2 is merged with another set that is *at least as big*. By induction, we can see that after i label changes, x is in a set of size at least 2^i.

Hence each element undergoes at most $\log n$ label changes. Every label change is assigned to an element, so there are at most $n \log n$, hence $O(n \log n)$, in all. The other operation in the n UNIONs take only $O(n)$ total time. This gives us the result:

Theorem: The linked-list implementation, always merging the smaller set into the larger, takes $O(n \log n + m)$ time to carry out any sequence of m union-find operations on a set of size n.

This is enough to remove the bottleneck in Kruskal's algorithm, so that it takes $O(e \log e)$ time. There are faster ways to find an MST. Prim's algorithm (cf. HW#2) takes $O(e + n \log n)$ if Fibonacci heaps are used to implement the priority queues needed. [CLRS] has more if you are interested.

Representing Sets as Trees

Can we do even better? One idea is to avoid updating every element of the smaller set each time, by making the structure for each set a *tree* instead of a linked list. Now the node for each element has just one pointer, to its parent:

- MAKESET: Create a node whose parent is itself, a **root**.
- UNION: Change the parent of the root of the smaller tree to be the root of the larger.
- FIND(x) : Follow the path from x to the root of its component, and return that root as the label.

There's a problem here – the time for a FIND is no longer $O(1)$ but $O(h)$, where h is the height of the tree. A UNION takes only $O(1)$ once we have found the two roots, but finding them also takes $O(h)$. We'll thus reclassify the UNION as two FIND operations plus the pointer changes to merge the trees, which we'll call LINK.

With the smaller-into-larger rule, it is easy to prove by induction that a tree of height h always has at least 2^h nodes, so that a tree with k nodes always has height $O(\log k)$. Actually, we get this property from a simpler rule – we keep track of the **rank** (height) of each node and always merge the tree of lower rank into that of higher rank.

Improvements to the Tree Method

When we carry out the operation FIND(x), we compute the label not only of x but of all the nodes on the path from x to its root. If we save this information, we can carry out a future FIND on x, or on one of its ancestors, more quickly. Let's try investing some time after each FIND to change the parent pointers of all the nodes on the path to the newly found root. This is called **tree compression** and at most doubles the time for a FIND. ([CLRS] gives a nice recursive implementation of this that avoids the need for any new pointers.)

What does this buy us? Remember that each UNION includes two FIND operations, and these will tend to flatten the trees for the components they involve. The FIND operations still take $O(h)$ time, where h is the height of the tree, but can we hope that h will be smaller than its former $O(\log n)$?

Ackermann's Function

The analysis of our last (and fastest) implementation of a union-find data structure turns out to involve a function first defined as a mathematical curiosity, **Ackermann's function**.

In CMPSCI 601 we define the **primitive recursive functions** from **N** (the natural numbers) to **N** as the possible results of a particular kind of recursive definition. An equivalent definition is the set of function that can be computer in an imperative language with **bounded loops** but no unbounded loops.

Ackermann devised his function to be computable (by a Turing machine, for example) but faster-growing than *any* primitive recursive function.

Definition: We define an infinite sequence of functions $A_0, A_1, A_2, \ldots$ as follows:

- $A_0(x) = 1 + x$ for any $x \in \mathbf{N}$
- for $k > 0$, $A_k(x)$ is A_{k-1} applied to x, x times

We can compute the first few A_i's easily:

- $A_1(x)$ is the result of adding 1 to x, x times, or $2x$
- $A_2(x)$ is the result of doubling x, x times, or $x2^x$
- $A_3(x)$ is *greater than* the result of raising 2 to the power x, x times. The latter number is called $\exp^*(x)$, a tower of twos x high
- $A_4(x)$ is a lot bigger than that…

Definition: The **Ackermann function** A is defined by $A(k) = A_k(2)$. Thus $A(0) = A_0(2) = 1 + 3 = 3$, $A(1) = A_1(2) = 2(2) = 4$, $A(2) = A_2(2) = 2 \cdot 2^2 = 8$, and $A(3) = A_3(2) = A_2(A_2(2)) = A_2(8) = 8 \cdot 2^8 = 2048$. The next value, $A(4)$, is greater than $\exp^*(2048)$, an inconceivably huge number.

Definition: The **inverse Ackermann function** $\alpha(n)$ is defined to be the smallest number k such that $A(k) \geq n$. Thus $\alpha(n) \leq 4$ for any conceivable number.

Theorem: The running time needed to perform m union-find operations on a set of size n is $\Theta(m\alpha(n))$, and the upper bound is achieved by our last algorithm.

We'll sketch the proof of the upper bound as time permits.

A Potential Function

The technique used to prove the $O(m\alpha(n))$ upper bound is an interesting one. In mechanics the total energy of a system consists of both kinetic and potential energy. Here we will define a **potential function** on states of the data structure, and measure the number of steps taken *plus* the increase in potential (or minus the decrease in potential) for each operation. We'll prove that each operation costs $O(\alpha(n))$ in this **amortized cost** measure. Even though a single operation might take more steps than that, it must reduce the potential at the same time. Since the potential will start at zero and finish non-negative, the $O(m\alpha(n))$ bound on the total amortized cost implies an $O(m\alpha(n))$ bound on the actual cost for all m steps.

The potential of the whole system will be the sum of a non-negative potential for each node.

The cost of a root node, or of any node of rank zero, will be $\alpha(n)$ times its rank. Other nodes have a more complicated definition of potential, for which we'll omit the details (see section 21.4 of [CLRS]). There are two parameters of such a node called the **level** $\ell(x)$ and **iteration number** $i(x)$, such that $0 \leq \ell(x) < \alpha(n)$ and $1 \leq i(x) \leq r(x)$. The exact potential of a node x is

$$r(x)[\alpha(n) - \ell(x)] - i(x).$$

Effect of Moves on the Potential

The MAKESET operation creates a new node with zero rank and hence zero potential, so its only cost is $O(1)$ for the actual operations it uses.

Recall that we broke the UNION operation into two FIND's and a LINK. The latter changes only one pointer, from the root of one tree to the root of another. If these two roots were at the same rank, the LINK increases the rank of one root by one, and thus increases its potential by $\alpha(n)$. It turns out that no other node's potential is changed, and since there are only $O(1)$ actual operations the total amortized cost is $O(\alpha(n))$.

Finally, a FIND operation involves all the nodes from the node found to its root – the number of these nodes, s, may be greater than α. But it turns out that with this definition of potential, a FIND operation with its tree compression *decreases* the potential of all but possibly $\alpha(n) + 2$ of these s nodes. With a careful choice of the units with which to measure actual steps, this makes the amortized cost of the FIND operation only $O(\alpha(n))$.

So any sequence of m MAKESET, FIND, and LINK operations takes $O(m\alpha(n))$ time. The same holds, with a different constant, for any sequence of m MAKESET, FIND, and UNION operations.

There is a matching **lower bound** argument that we omit here. It shows that any data structure for union-find, under certain general assumptions, must take $\Omega(m\alpha(n))$ time. Detailed references may be found at the end of Chapter 21 of [CLRS].