Lecture 10 Union-Find

The union-find data structure is motivated by Kruskal's minimum spanning tree algorithm (Algorithm 2.6), in which we needed two operations on disjoint sets of vertices:

- determine whether vertices u and v are in the same set;
- form the union of disjoint sets A and B.

The data structure provides two operations from which the above two operations can be implemented:

- **find**(v), which returns a canonical element of the set containing v. We ask if u and v are in the same set by asking if **find**$(u) =$ **find**(v).
- **union**(u, v), which merges the sets containing the canonical elements u and v.

To implement these operations efficiently, we represent each set as a tree with data elements at the vertices. Each element u has a pointer parent (u) to its parent in the tree. The root serves as the canonical element of the set.

To effect a **union**(u, v), we combine the two trees with roots u and v by making u a child of v or vice-versa. To do a **find**(u), we start at u and follow parent pointers, traversing the path up to the root of the tree containing u, which gives the canonical element.

To improve performance, we will use two heuristics:

- When merging two trees in a **union**, always make the root of the smaller tree a child of the root of the larger. We maintain with each vertex u the size of the subtree rooted at u, and update whenever we do a **union**.
- After finding the root v of the tree containing u in a **find**(u), we traverse the path from u to v one more time and change the parent pointers of all vertices along the path to point directly to v. This process is called *path compression*. It will pay off in subsequent **find** operations, since we will be traversing shorter paths.

Let us start with some basic observations about these heuristics. Let σ be a sequence of m **union** and **find** operations starting with n singleton sets. Consider the execution of σ both with and without path compression. In either case we combine two smaller sets to form a larger in each **union** operation. Observe that the collection of sets at time t is the same with or without path compression, and the trees have the same roots, although the trees will in general be shorter and bushier with path compression. Observe also that u becomes a descendant of v at time t with path compression if and only if u becomes a descendant of v at time t without path compression. However, without path compression, once u becomes a descendant of v, it remains a descendant of v forever, but with path compression, it might later become a non-descendant of v.

10.1 Ackermann's Function

The two heuristics will allow a sequence of **union** and **find** operations to be performed in $O((m+n)\alpha(n))$ time, where $\alpha(n)$ is the inverse of *Ackermann's function*. Ackermann's function is a famous function that is known for its extremely rapid growth. Its inverse $\alpha(n)$ grows extremely slowly. The texts [3, 100] give inequivalent definitions of Ackermann's function, and in fact there does not seem to be any general agreement on the definition of "the" Ackermann's function; but all these functions grow at roughly the same rate. Here is yet another definition that grows at roughly the same rate:

$$\begin{aligned} A_0(x) &= x+1 \\ A_{k+1}(x) &= A_k^x(x) \end{aligned}$$

where A_k^i is the i-fold composition of A_k with itself:

$$A_k^i = \underbrace{A_k \circ \cdots \circ A_k}_{i}$$

or more accurately,

$$A_k^0 = \text{the identity function}$$

$$A_k^{i+1} = A_k \circ A_k^i .$$

In other words, to compute $A_{k+1}(x)$, start with x and apply A_k x times. It is not hard to show by induction that A_k is monotone in the sense that

$$x \leq y \rightarrow A_k(x) \leq A_k(y)$$

and that for all x, $x \leq A_k(x)$.

As k grows, these functions get extremely huge extremely fast. For $x = 0$ or 1, the numbers $A_k(x)$ are small. For $x \geq 2$,

$$\begin{aligned}
A_0(x) &= x+1 \\
A_1(x) &= A_0^x(x) = 2x \\
A_2(x) &= A_1^x(x) = x2^x \geq 2^x \\
A_3(x) &= A_2^x(x) \geq \underbrace{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}_{x} = 2 \uparrow x \\
A_4(x) &= A_3^x(x) \geq \underbrace{2 \uparrow (2 \uparrow \cdots \uparrow (2 \uparrow 2) \cdots)}_{x} = 2 \uparrow\uparrow x \\
&\vdots
\end{aligned}$$

For $x = 2$, the growth of $A_k(2)$ as a function of k is beyond comprehension. Already for $k = 4$, the value of $A_4(2)$ is larger than the number of atomic particles in the known universe or the number of nanoseconds since the Big Bang.

$$\begin{aligned}
A_0(2) &= 3 \\
A_1(2) &= 4 \\
A_2(2) &= 8 \\
A_3(2) &= 2^{11} = 2048 \\
A_4(2) &\geq 2 \uparrow 2048 = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}_{2048}
\end{aligned}$$

We define a unary function that majorizes all the A_k (*i.e.*, grows asymptotically faster than all of them):

$$A(k) = A_k(2)$$

and call it Ackermann's function. This function grows asymptotically faster than any primitive recursive function, since it can be shown that all primitive recursive functions are bounded almost everywhere by one of the functions A_k. The primitive recursive functions are those computed by a simple PASCAL-like programming language over the natural numbers with **for** loops but no **while**

loops. The level k corresponds roughly to the depth of nesting of the **for** loops [79].

The inverse of Ackermann's function is

$$\alpha(n) \quad = \quad \text{the least } k \text{ such that } A(k) \geq n$$

which for all practical purposes is 4. We will show next time that with our heuristics, any sequence of m **union** and **find** operations take at most $O((m+n)\alpha(n))$ time, which is not quite linear but might as well be for all practical purposes. This result is due to Tarjan (see [100]). A corresponding lower bound for pointer machines with no random access has also been established [99, 87].

Lecture 11 Analysis of Union-Find

Recall from last time the heuristics:

- In a **union**, always merge the smaller tree into the larger.
- In a **find**, use path compression.

We made several elementary observations about these heuristics:

- the contents of the trees are the same with or without path compression;
- the roots of the trees are the same with or without path compression;
- a vertex u becomes a descendant of v at time t with path compression if and only if it does so without path compression. With path compression, however, u may at some later point become a non-descendant of v.

Recall also the definitions of the functions A_k and α:

$$\begin{aligned} A_0(x) &= x+1 \\ A_{k+1}(x) &= A_k^x(x) \\ \alpha(n) &= \text{least } k \text{ such that } A_k(2) \geq n \end{aligned} \tag{15}$$

and that $\alpha(n) \leq 4$ for all practical values of n.

11.1 Rank of a Node

As in the last lecture, let σ be a sequence of m **union** and **find** instructions starting with n singleton sets. Let $T_t(u)$ denote the subtree rooted at u at time t in the execution of σ *without* path compression, and define the *rank* of u to be

$$\text{rank}\,(u) \;=\; 2 + \text{height}\,(T_m(u))\;, \tag{16}$$

where height (T) is the *height* of T or length of the longest path in T. In other words, we execute σ without path compression, then find the longest path in the resulting tree below u. The rank of u is defined to be two more than the length of this path. (Beware that our rank is two more than the rank as defined in [3, 100]. This is for technical reasons; the 2's in (15) and (16) are related.)

As long as u has no parent, the height of $T_t(u)$ can still increase, since other trees can be merged into it; but once u becomes a child of another vertex, then the tree rooted at u becomes fixed, since no trees will ever again be merged into it. Also, without path compression, the height of a tree can never decrease. It follows that if u ever becomes a descendant of v (with or without path compression), say at time t, then for all $s > t$ the height of $T_s(u)$ is less than the height of $T_s(v)$, therefore

$$\text{rank}\,(u) \;<\; \text{rank}\,(v)\;. \tag{17}$$

The following lemma captures the intuition that if we always merge smaller trees into larger, the trees will be relatively balanced.

Lemma 11.1

$$|T_t(u)| \;\geq\; 2^{\text{height}\,(T_t(u))}\;. \tag{18}$$

Proof. The proof is by induction on t, using the fact that we always merge smaller trees into larger. For the basis, we have $T_0(u) = \{u\}$, thus height $(T_0(u)) = 0$ and $|T_0(u)| = 1$, so (18) holds at time 0. If (18) holds at time t and the height of the tree does not increase in the next step, *i.e.* if height $(T_{t+1}(u))$ = height $(T_t(u))$, then (18) still holds at time $t+1$, since $|T_{t+1}(u)| \geq |T_t(u)|$. Finally, if height $(T_{t+1}(u))$ > height $(T_t(u))$, then the instruction executed at time t must be a **union** instruction that merges a tree $T_t(v)$ into $T_t(u)$, making v a child of u in $T_{t+1}(u)$. Then

$$\text{height}\,(T_t(v)) \;=\; \text{height}\,(T_{t+1}(v)) \;=\; \text{height}\,(T_{t+1}(u)) - 1\;.$$

By the induction hypothesis,

$$|T_t(v)| \;\geq\; 2^{\text{height}\,(T_t(v))}\;.$$

Since we always merge smaller trees into larger,

$$|T_t(u)| \geq |T_t(v)| .$$

Therefore

$$\begin{aligned} |T_{t+1}(u)| &= |T_t(u)| + |T_t(v)| \\ &\geq 2^{\text{height}\,(T_t(v))} + 2^{\text{height}\,(T_t(v))} \\ &= 2^{\text{height}\,(T_t(v))+1} \\ &= 2^{\text{height}\,(T_{t+1}(u))} . \end{aligned}$$

□

Lemma 11.2 *The maximum rank after executing σ is at most $\lfloor \log n \rfloor + 2$.*

Proof. By Lemma 11.1,

$$n \geq |T_m(u)| \geq 2^{\text{height}\,(T_m(u))} \geq 2^{\text{rank}\,(u)-2} ,$$

so

$$\lfloor \log n \rfloor \geq \text{rank}\,(u) - 2 .$$

□

Lemma 11.3

$$|\{u \mid \text{rank}\,(u) = r\}| \leq \frac{n}{2^{r-2}} .$$

Proof. If rank (u) = rank (v), then by (17) $T_m(u)$ and $T_m(v)$ are disjoint. Thus

$$\begin{aligned} n &\geq \Big| \bigcup_{\text{rank}\,(u)=r} T_m(u) \Big| \\ &= \sum_{\text{rank}\,(u)=r} |T_m(u)| \\ &\geq \sum_{\text{rank}\,(u)=r} 2^{r-2} \quad \text{by Lemma 11.1} \\ &= |\{u \mid \text{rank}\,(u) = r\}| \cdot 2^{r-2} . \end{aligned}$$

□

Now consider the execution of σ with path compression. We will focus on the distance between u and parent (u) as measured by the difference in their ranks, and how this distance increases due to path compression. Recall that rank (u) is fixed and independent of time; however, rank (parent (u)) can

change with time because the parent of u can change due to path compression. By (17), this value can only increase.

Specifically, we consider the following conditions, one for each k:

$$\text{rank}(\text{parent}(u)) \geq A_k(\text{rank}(u)) . \tag{19}$$

Define

$$\delta(u) = \text{the greatest } k \text{ for which (19) holds.}$$

The value of $\delta(u)$ is time-dependent and can increase with time due to path compression. Note that $\delta(u)$ exists if u has a parent, since by (17),

$$\text{rank}(\text{parent}(u)) \geq \text{rank}(u) + 1 = A_0(\text{rank}(u))$$

at the very least.

For $n \geq 5$, the maximum value $\delta(u)$ can take on is $\alpha(n) - 1$, since if $\delta(u) = k$,

$$\begin{aligned} n &> \lfloor \log n \rfloor + 2 \\ &\geq \text{rank}(\text{parent}(u)) \quad \text{by Lemma 11.2} \\ &\geq A_k(\text{rank}(u)) \\ &\geq A_k(2) , \end{aligned}$$

therefore

$$\alpha(n) > k .$$

11.2 Analysis

Each **union** operation requires constant time, thus the time for all **union** instructions is $O(m)$.

Each instruction **find**(u) takes time proportional to the length of the path from u to v, where v is the root of the tree containing u. The path is traversed twice, once to find v and then once again to change all the parent pointers along the path to point to v. This amounts to constant time (say one time unit) per vertex along the path. We charge the time unit associated such a vertex x as follows:

- If x has an ancestor y on the path such that $\delta(y) = \delta(x)$, then charge x's time unit to x itself.
- If x has no such ancestor, then charge x's time unit to the **find** instruction.

Let us now tally separately the total number of time units apportioned to the vertices and to the **find** instructions and show that in each case the total is $O((m+n)\alpha(n))$.

There are at most $\alpha(n)$ time units charged to each **find** instruction, at most one for each of the $\alpha(n)$ possible values of δ, since for each such value k only the last vertex x on the path with $\delta(x) = k$ gets its time unit charged to the **find** instruction. Since there are at most m **find** instructions in all, the total time charged to **find** instructions is $O(m\alpha(n))$.

Let us now count all the charges to a particular vertex x over the course of the entire computation. For such a charge occurring at time t, x must have an ancestor y such that $\delta(y) = \delta(x) = k$ for some k. Then at time t,

$$\begin{aligned} \text{rank (parent } (x)) &\geq A_k(\text{rank } (x)) \\ \text{rank (parent } (y)) &\geq A_k(\text{rank } (y)) \ . \end{aligned}$$

Suppose that in fact

$$\text{rank (parent } (x)) \geq A_k^i(\text{rank } (x)) \ , \quad i \geq 1 \ .$$

Let v be the last vertex on the path. Then at time t,

$$\begin{aligned} \text{rank } (v) &\geq \text{rank (parent } (y)) \\ &\geq A_k(\text{rank } (y)) \\ &\geq A_k(\text{rank (parent } (x))) \\ &\geq A_k(A_k^i(\text{rank } (x))) \\ &\geq A_k^{i+1}(\text{rank } (x)) \ , \end{aligned}$$

and since v is the new parent of x at time $t+1$, we have at time $t+1$ that

$$\text{rank (parent } (x)) \geq A_k^{i+1}(\text{rank } (x)) \ .$$

Thus at most rank (x) such charges can be made against x before

$$\begin{aligned} \text{rank (parent } (x)) &\geq A_k^{\text{rank } (x)}(\text{rank } (x)) \\ &= A_{k+1}(\text{rank } (x)) \ , \end{aligned}$$

and at that point

$$\delta(x) \geq k+1 \ .$$

Thus after at most rank (x) such charges against x, $\delta(x)$ increases by at least one. Since $\delta(x)$ can increase only $\alpha(n) - 1$ times, there can be at most rank $(x)\alpha(n)$ such charges against x in all. By Lemma 11.3, there are at most

$$r\alpha(n)\frac{n}{2^{r-2}} = n\alpha(n)\frac{r}{2^{r-2}}$$

charges against vertices of rank r. Summing over all values of r, we obtain the following bound on all charges to all vertices:

$$\begin{aligned}\sum_{r=0}^{\infty} n\alpha(n)\frac{r}{2^{r-2}} &= n\alpha(n)\cdot\sum_{r=0}^{\infty}\frac{r}{2^{r-2}} \\ &= 8n\alpha(n)\ .\end{aligned}$$

We have shown

Theorem 11.4 *A sequence of m* **union** *and* **find** *operations starting with n singleton sets takes time at most $O((m+n)\alpha(n))$.*