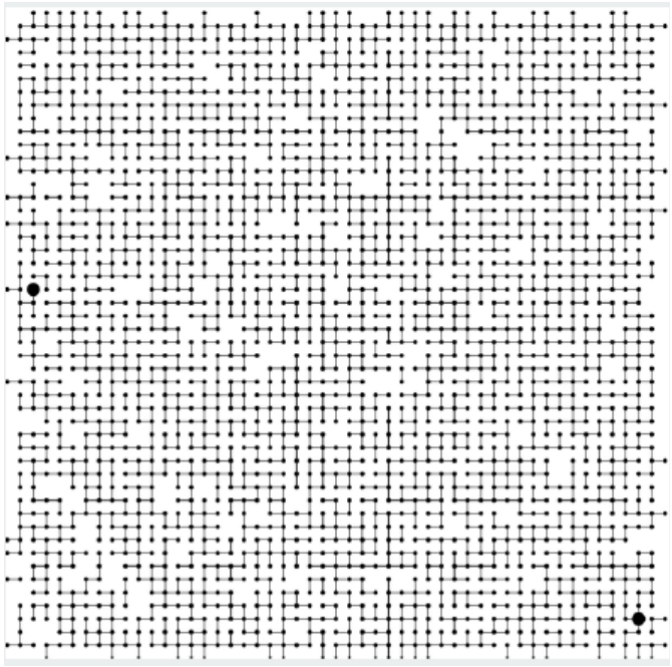
Amortized Analysis

Union-Find

A practical question: In the following network, are the two big dots connected?



Suppose there are n vertices and m edges in the graph. Doing search for each query isn't wise – would cost $O(n^2)$ time to compute a shortest path. Notice that connectivity relation is *transitive*. All the connected vertices form an equivalence class. Such query can be answered with the following index structure:

- Put the n vertices into k disjoint subsets (equivalence classes under the connectivity relation).
- Find: Given any vertex, query the subset id (or a representative) that holds the vertex.

Then, one can determine if two vertices are connected by comparing their subset ids. The index can be implemented with an array A where $A[i]$ keeps the ID of the subset that contains element i.

For example, the following array indicates six sets: {0}, {1}, {2,3,4,9}, {5,6}, {7}, {8}.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	9	9	6	6	7	8	9

Real applications of this include

- Computers in a network

- Web pages on the Internet
- Transistors in a computer chip
- Variable name aliases.
- Pixels in a digital photo
- Minimum spanning tree

Note that if the graph does not change, then computing the connected component can be done by a depth-first search. The time complexity to build the subsets takes $O(n+m)$ time. But if the graph changes dynamically, then rebuilding the index from scratch isn't wise. So we'd like to have a dynamically changing data structure that supports the adding of new edges to the graph. We don't deal with deletion.

UnionFind Data Structure: Maintains a collection of disjoint sets subject to the following two operations:

1. Union(A, B): Replace two sets A and B with one new set $A \cup B$.
2. Find(e): Return the subset that contains e.

Thus, adding an edge can be achieved by first finding the two sets containing the two vertices, and then union of the two sets.

Recall the MST algorithm:

Minimum Spanning Tree

1. $T \leftarrow \emptyset$
2. Sort edges according to weight $e_1, \ldots, e_m$
3. For i from 1 to m

3.1 if e_i connects two connected components in T

3.2 $T = T \cup \{e_i\}$

Clearly the connected components in T can be maintained with a UnionFind data structure. Then adding an edge will union the two sets that contain the new edge's two vertices, respectively. The amortized efficiency of the two operations will affect the MST algorithm's complexity. The MST algorithm takes $O(m \log m) + 2m \cdot Finds + n \cdot Unions$.

Data Structure I:

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	9	9	6	6	7	8	9

If we still use array as our data structure, then Find takes $O(1)$ time.

Union(A,B) would require rewriting the set ID for all elements in one of A and B. That would take $O(|A|)$ time. The worst case is that we keep adding a single-element set B to A, which would cost $O(n^2)$ for n union operations.

Data Structure II:

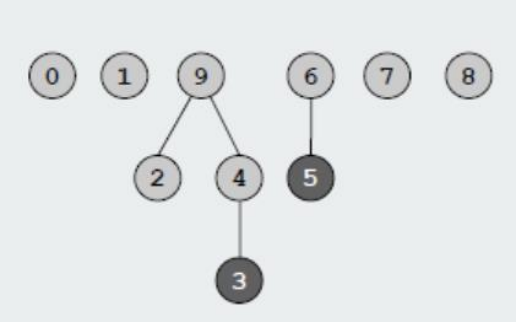
In addition to data structure I, additionally maintain a size array for each subset. During the union operation, rewrite the id for the smaller of A and B.

Examine the $N-1$ union operations that are responsible to a set with size N. Whenever an element's set ID is rewritten, its set's size is at least doubled. So, an element's set ID is rewritten by at most $\log_2 N$ times. The total time spent for all the N elements is then $N \log_2 N$. Average it on each union operation, and repeat the argument on each set. The amortized time complexity for each union operation is $O(\log n)$ if there are n elements in total.

Note that this data structure II is good enough for the MST algorithm. The sorting of edges start to dominate the time complexity. However, when the graph is unweighted or when the weights can be sorted with a linear time sorting algorithm, it still makes sense to improve the data structure.

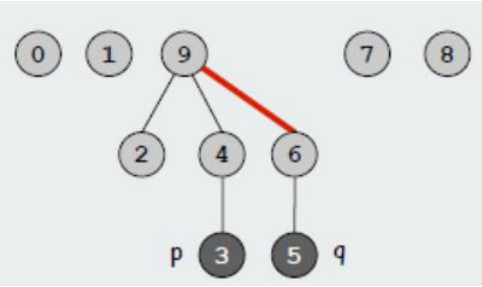
Data Structure III: (Disjoint set forest)

Use a tree to represent a set.



Find: Starting from an element, keep checking parent until find the root; return the root.

Union(A,B): Put B's root as a child of A's root.



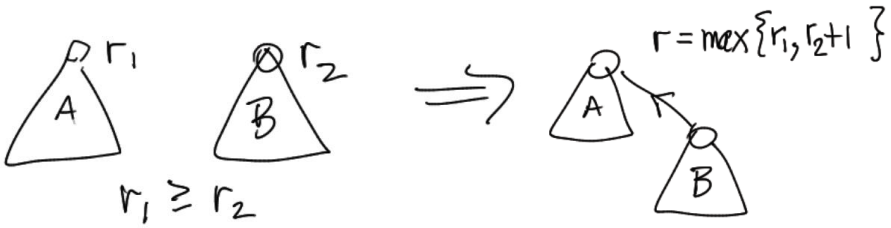
Example: Union(9,6) will merge the two trees as one.

Time complexity: Union takes $O(1)$ time. But Find may take $O(n)$ because the tree depth can be very large.

Data Structure IV: (Disjoint forest with union by rank heuristics)

In addition to data structure III, maintain a height value at the root of each tree. When union needs to merge two trees, add the lower tree as the taller tree's root's child.

Instead of using "height", let us call this value as the "rank". The rank of a single element tree is initialized as 0. Then rank is updated as follows during union operations.



Clearly in this data structure, rank is equal to the height of a vertex. But in the data structure we will learn later, rank and height may differ.

Lemma. A rank r vertex has at least 2^r descendants.

Proof: Lemma is true for $r = 0$. Prove $r > 0$ by induction. Getting the first rank r tree will have to be the union of two rank $r - 1$ trees. By induction a height $r - 1$ tree has at least 2^{r-1} descendants, thus a rank r tree must have at least 2^r descendants.

QED.

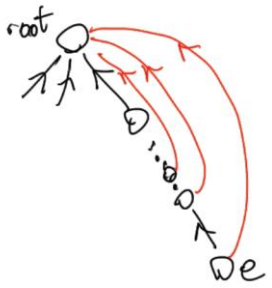
By the lemma, the rank of a tree is at most $\log_2 n$. Thus, the Find takes $O(\log_2 n)$ in this new data structure.

Compare data structures II and IV. II is fast at find and slow at union, but IV is fast at union but slow at find.

Data Structure V: (Disjoint-set forest with union by rank and path compression heuristics)

This is a further improvement over IV.

During a find operation, we link each node encountered on the parenting path to the root. This heuristics, named path compression, takes little additional time but makes future find operations easier.



this doubles work of Find
– same $O(\cdot)$

Note that this data structure is very easy to implement. But the analysis of its performance is exceedingly difficult. The data structure was first described in (Galler, Bernard A.; Fischer, Michael J., An improved equivalence algorithm, Communications of the ACM 7 (5): 301–303, 1964). But the proof of the first precise analysis appeared in (Tarjan, Robert Endre. Efficiency of a Good But Not Linear Set Union Algorithm. Journal of the ACM 22 (2): 215–225. 1975). Here we give the main theorem without proof.

The Ackermann function is defined on $m \geq 0$ and $n \geq 1$.

$$A(m,n) = \begin{cases} n+1, \text{ if } m = 0 \\ A(m-1,1), \text{ if } m > 0 \text{ and } n = 0 \\ A\big(m-1, A(m,n-1)\big), \text{ if } m > 0 \text{ and } n > 0 \end{cases}$$

Theorem: Using the disjoint-set forest with union by rank and path compression heuristics, m operations on n elements takes $O\big(m \cdot \alpha(m,n)\big)$ time. Here

$$\alpha(m,n) = \min\left\{k: A\left(k, \left\lfloor \frac{m}{n} \right\rfloor\right) \geq \log_2 n\right\}$$

is the inverse Ackermann function.

Essentially, $\alpha(m,n) \leq 4$ for most practical cases (when $n \leq 2^{65533}$). So, $\alpha(m,n)$ is almost a constant.

Here we prove a weaker result but the proof is simper.

Notation: Denote $t(k) = \underbrace{2^{2^{\cdots^2}}}_{k \text{ times}}$. $t(k)$ is the tower of 2 function. Let

$$\log^* n = \min\{k: \underbrace{\log \log \ldots \log n}_{k \text{ times}} \leq 1\}$$

be the number of times we need to apply logarithm to reduce n to be less than 1.

Theorem: Using the disjoint-set forest with union by rank and path compression heuristics, m operations on n elements takes $O(m \cdot \log^* n)$ time.

n	2	3..4	5..16	17..65536	$65537..2^{65536}$
log*(n)	1	2	3	4	5

So, the result is not much weaker.

Proof:

First two simple facts:

Claim 1: If u is v's parent, then $rank(v) < rank(u)$.

This is clear from the rank assignment during union operations.

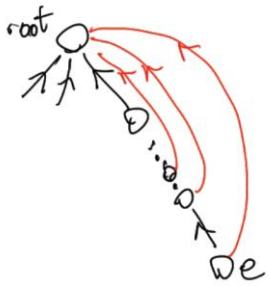
Claim 2. The number of vertices with rank r is no more than $\frac{n}{2^r}$.

This is because a vertex of rank r has $\geq 2^r$ descendants and two different vertices of rank r have disjoint descendants.

Now move back to the main proof. Without loss of generality we can assume there are $n-1$ union operations that put everything in a single tree. Otherwise we can prove the theorem on each tree separately, and add things up. A consequence of this assumption is $m \geq n$, where m is the total number of operations.

A non-root vertex v will not change its $rank(v)$. So it is meaningful to talk about $\log^*\big(rank(v)\big)$ for a non-root vertex v. Define $g(v) = \log^*\big(rank(v)\big)$ be v's group. The vertices are grouped into $\log^* n$ groups. Let $p(v)$ be v's parent.

Consider a find operation on e. The algorithm goes through the path from e to the root, and does path compression.



For each v on this path, we charge the cost $O(1)$ in two different ways:

- Case 1. v has parent and grandparent, and $g(v) = g\big(p(v)\big)$. We charge cost 1 to the node v.
- Case 2. Otherwise, we charge 1 to the find operation.

The find operation is charged at most $O(\log^* n)$. So, we only need to show that the total charge to all vertices in all Find operations is bounded by $O(n \log^* n)$. Then the average cost per operation is $O(\log^* n)$.

Each time v is charged, the path compression will connect it to a new parent with higher rank (it's grandparent or higher), until to a point when $g(v) < g\big(p(v)\big)$. Then v will never be charged again. Thus a vertex v will be charged at most $t(g(v))$ times until its parent's rank becomes greater than $t\big(g(v)\big)$.

Thus, the total charge to vertices is upper bounded by $\sum_v t\big(g(v)\big)$. Thus, to prove the theorem, we only remain to prove the following technical lemma.

Lemma:

$$\sum_v t(g(v)) \leq n \log^* n$$

Now the proof becomes purely technical. Notice that

$$\sum_v t(g(v)) = \sum_{k=1}^{\log^* n} \sum_{v \in group\ k} t(k) = \sum_{k=1}^{\log^* n} t(k) \cdot (size\ of\ group\ k),$$

Note that group k contains vertices with rank from $t(k-1)+1$ to $t(k)$. From Claim 2,

$$size\ of\ group\ k \leq \sum_{r=t(k-1)+1}^{t(k)} \frac{n}{2^r} \leq \frac{n}{2^{t(k-1)+1}} \cdot \sum_{r=0}^{\infty} \frac{1}{2^r} \leq \frac{n}{2^{t(k-1)}} = \frac{n}{t(k)}$$

Combining the above two inequalities, the Lemma is proved; and therefore the theorem.

QED