Lecture 19: Union-Find

built on 2019/06/26 at 13:44:56

Today we're going to apply this recipe on the so-called **Union-Find** problem. To motivate this study, we'll talk about a question from a real-life situation.

1 Are we connected?

Many of us are Facebook users. On this popular social-networking site, a user could request to be friend with another user, who will accept or decline the connection; however, once the relationship has been established, it is mutual—if A is a friend of B, then B is a friend of A.

The simple question that arises often is, given the friendship connection, *is a user X "connected" to a user Y through a chain of friendship?* This question is one that is asked and answered often as a component of other processes. To model this interaction, we'll use the following abstraction:

- *basic entity:* users
- union(x, y)—x and y are friends.
- connected(x, y)—can x and y reach each other through a chain of friendship?

This abstraction turns out to be surprisingly rich and has found applications such as:

- basic entity: computers in a network. Can machine A reach machine B?
- basic entity: variable name aliases. Are these variables the same? (Plagiarism detection)
- basic entity: pages on the Internet. Are they connected?
- etc.

In these applications, the kinds of basic entities differ but we aren't taking advantage of what they are. To us, they are different objects. So we'll hide details irrelevant to the basic problem at hand.

2 Oh Lovely Graph

Of course, the problem just stated can be couched as a graph problem. The vertices are entities in our graph. The edge set is initially empty. But as x and y become friends, we add an edge between x and y. So then, the question of whether p and q are connected, for a given pair of p and q, is simply asking *whether there is a path between p and q in this graph.*

In the past two lectures, we saw that both BFS and DFS can be used to find all the vertices reachable from a source vertex s. Therefore, if we let $s = p$ and run BFS or DFS, we can see whether or not q is reached—and that is the answer to our question.

However, this is not at all efficient. Each BFS or DFS run takes $O(m)$ time, and somehow if the graph doesn't change much between these `connected` queries, why should one carry out BFS and DFS anew?

We'll look at a way to recast the problem into something almost completely different that will allow us to tackle the problem much more efficiently.

3 The Union-Find Problem

Suppose there are n objects (entities) we're considering. We'll name them $0, 1, \ldots, n-1$. If we really need to map between real entities and their integer IDs, we could use a dictionary data structure (in a few lectures). Hence, we model the **Union-Find problem** as follows:

- *Objects:* $0, 1, \ldots, n-1$ for a collection involving n objects.
- Conceptually, we're representing disjoint sets of objects. For example,

```
   {0}  {1}  {4}  {2 5 9} {3 6} {7 8}
id: 0    1    4      2      3     7
```

- Operations: union and find.
- find(x) — returns an identifer for the set that x belongs to. This identifier is special in that x and y are connected if and only if find(x) == find(y). For example, find$(5) \hookrightarrow 2$. Hence, because find of 2 and 9 returns the same id, we know they're connected. But find(1) and find(4) return different ids, so they aren't connected.
- union(x, y) — merge together the set containing x and the set containing y. For example, merge$(5, 3)$ gives
  `{0}  {1}  {4}  {2 5 9 3 6} {7 8}`
- Most often, when we create a new instance, we start with each object being in its own set, i.e.,
  `{0}  {1} {2} ... {n-1}`
- Note: we get to design the IDs.

Our goal is to design an efficient data structure for union-find, where we note that the operations union and find may be interleaved. Furthermore, the total number of operations m is generally large, and the number of objects n is also large.

4 Eager Union For Quick Find

Since find(x) has to return a number representing the id of the set x belongs to, we'll start with the simple idea of keeping a list `id` where `id[`x`]` is the id of the object x. This means that the following invariant is readily met:

> p and q are connected if and only if `id[p]` is equal to `id[q]`. In other words, all members of a set must have the same value in `id[]`.

Here's an example of the list `id`

```
index: 0  1  2  3  4  5  6  7  8  9
  id:  0  1  9  9  9  6  6  7  8  9
```

that satisfies this invariant for the collection of sets

$$\{0\}\ \{1\}\ \{2,3,4,9\}\ \{5,6\}\ \{7\}\ \{8\}$$

The question to ask ourselves is, *how should we give an object an id?* There are two subquestions: First, what should the id's be initially? And second, when two sets merge, how should their corresponding id's change? Keep in mind that the id's must change when a union operation is called.

It turns out the initial id's don't matter much, and for simplicity, we could use i for the set containing just i. Then, we could use the following approach to handle union and merge:

- find(p): return `id[`p`]`; and
- union(p, q) to change all entries with `id[`p`]` to `id[`q`]`.

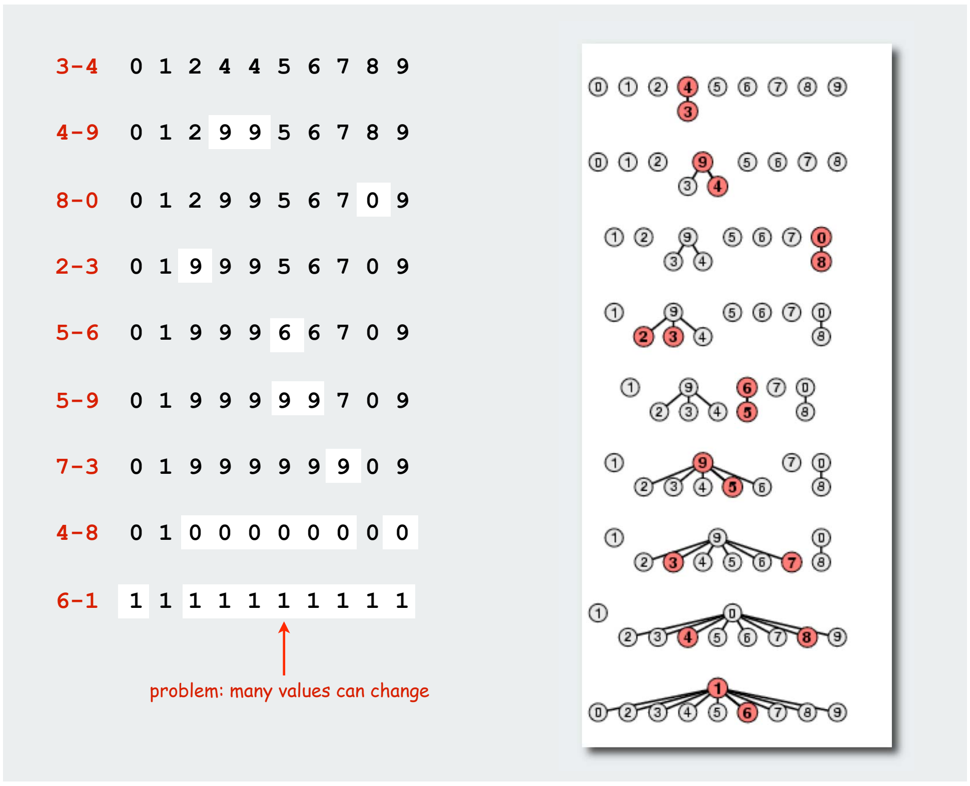
In code, we have:

```java
public class EagerUnion implements UnionFind {
  private int[] id;

  public EagerUnion(int n) {
    id = new int[n];
    for (int i=0;i<n;i++) id[i]=i;
  }

  public int find(int p) { return id[p]; }

  public void union(int p, int q) {
    int pid = id[p];
    for (int i=0;i<id.length;i++)
```



```
3-4  0 1 2 4 4 5 6 7 8 9
4-9  0 1 2 9 9 5 6 7 8 9
8-0  0 1 2 9 9 5 6 7 0 9
2-3  0 1 9 9 9 5 6 7 0 9
5-6  0 1 9 9 9 6 6 7 0 9
5-9  0 1 9 9 9 9 9 7 0 9
7-3  0 1 9 9 9 9 9 9 0 9
4-8  0 1 0 0 0 0 0 0 0 0
6-1  1 1 1 1 1 1 1 1 1 1
```

Figure 1: Eager union in action

```
      if (id[i]==pid) id[i]=id[q];
  }
}
```

As stated, the running time of this union-find implementation is $O(1)$ for find (indeed it's fast) and $O(n)$ for union, where the space consumption is $O(n)$.

Let's look at a worked-out example to understand how this works in action (example from Sedgewick's Algorithms in Java). See Figure 1.

We should note that the $O(n)$ bound for union isn't all that pessimistic because there are cases where about n objects need to take on a new id.

Following the recipe stated earlier, after testing for correctness, we ask ourselves if the solution is fast enough, and a back-of-the-envelope calculation indicates that it probably is not going to cut it for large datasets. Specifically, eager union requires upto $O(mn)$ time to handle m unions on n objects. Say we can read and write about 10^9 integers per second. If we have 10^{10} users (not far-fetched, hi, Facebook) and about 10^9 connections (again, not far-fetched), this solution will need to read and write about 10^{19} integers—hence about $10^{19}/10^9$ seconds, which is 2.78 million hours, or roughly 317 years.

5 Lazy Union

We could summarize the problem with the previous solution in a short sentence: union was too eager. We could easily fix this at the expense of a more complex find operation. We propose to retain the same `id[·]` array but this time, we don't require that `id[`p`]` $==$ `id[`q`]` if and only if p and q are connected.

We reinterpret `id[`p`]` as the *parent* of p. This naturally forms a family tree. Thus, we can define the concept of the *root* of an object p—that is, we keep asking for the parent of p successively until we find an ancestor whose parent turns out to be itself. Specifically,

$root(p)$ is `id[id[id[`$\ldots$`id[`p`]`$\ldots$`]]]` that keeps going until the parent of that is itself. In code:

```
int root(int p) {
  int pid = id[p];
  while (pid!=id[pid])
    pid = id[pid];
  return pid;
}
```

For example, the id's list `[0, 1, 9, 4, 9, 6, 6, 7, 8, 9]` corresponds to the following shape:

```
0   1   9    6  7  8
       / \   |
      2   4  5
          |
          3
```

Here, we seek to maintain the invariant that p and q are connected if and only if $root(p) == root(q)$. Indeed, this simplifies union because all it has to do for union(p, q) is setting the id of q's root to the id of p's root—i.e., $\texttt{id}[root(q)] \leftarrow root(p)$.

Let's watch this in action:

```
union(1, 0)    0 ->1   2   3   4   5   6   7   8
union(2, 0)    0 ->1 ->2   3   4   5   6   7   8
union(3, 0)    0 ->1 ->2 ->3   4   5   6   7   8
union(4, 0)    0 ->1 ->2 ->3 ->4   5   6   7   8
 ...
union(8, 0)    0 -> 1 -> 2 -> 3 -> ... -> 8
```

In this case, find(0) will take $O(n)$ time, which is awful. So because root can take up to linear time, both union and find are potentially $O(n)$ operations. Ouch! We haven't made any progress.

The problem at a glance:

Problems with Eager-Union:

- Union takes too long.
- Flat structure but keeping it flat is expensive.

Problems with Lazy-Union:

- The structure can be very deep
- Root becomes expensive.
- Root is needed for both union and find (!!).

6 Improvement: Depth Control

Lazy union turns out to be a basis for an efficient algorithm. The challenge? We need to avoid a deep structure. This can be accomplished via small modifications.

The basic idea is to keep track of the size of each root so that we could make an informed decision when joining two roots together during a union. The trick here is to **link the smaller root into the larger one**.

Let's look at an example (figure taken from Sedgewick's Algorithms). See Figure 2.

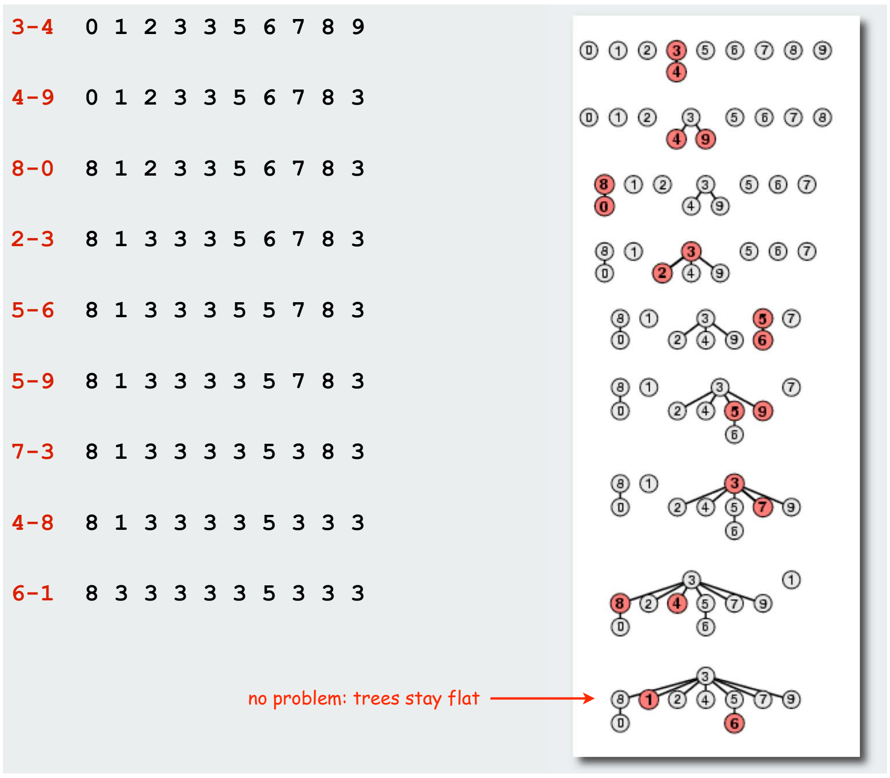
We indeed see that the structure is kept rather flat, so finding the root of any node takes little time. But just how flat the structure is needs to be quantified mathematically.

Theorem 6.1 *For any object p, the operation $root(p)$ visits at most $O(\log n)$ objects before reaching the root.*

Proof: Let p be an object that we're going to run $root(\cdot)$ on. Suppose the function visits a total of $t + 1$ objects, where $p = o_t$ is the starting point and o_0 is the root (i.e., $\texttt{id}[o_0] = o_0$). That is, for $i \geqslant 1$, $o_{i-1} = \texttt{id}[o_i]$. To prove this theorem, we'll define $n(a)$ to be the number of descendents of a (i.e., all objects where asking for their roots will touch a). So it follows that $n(o_0) \leqslant n$ because there are only n objects in all.

The crux of the proof is the following observation: $n(o_{i+1}) \leqslant n(o_i)/2$. This holds because:

- at one point, o_{i+1} joined into o_i.



```
3-4   0 1 2 3 3 5 6 7 8 9
4-9   0 1 2 3 3 5 6 7 8 3
8-0   8 1 2 3 3 5 6 7 8 3
2-3   8 1 3 3 3 5 6 7 8 3
5-6   8 1 3 3 3 5 5 7 8 3
5-9   8 1 3 3 3 3 5 7 8 3
7-3   8 1 3 3 3 3 5 3 8 3
4-8   8 1 3 3 3 3 5 3 3 3
6-1   8 3 3 3 3 3 5 3 3 3
```

Figure 2: Lazy union with depth control: small into large

- back then, right before the join took place:

$$\underbrace{\text{the size of } o_{i+1}}_{=n(o_{i+1})} \leqslant \underbrace{\text{the size of } o_i}_{=\beta} \quad \text{and} \quad n(o_{i+1}) + \beta \leqslant n(o_i)$$

  as a few other objects may have joined into o_i (but not o_{i+1} because it's no longer a root) since then.
- this means $n(o_i) \geqslant \beta + n(o_{i+1}) \geqslant 2n(o_{i+1})$, so $n(o_{i+1}) \leqslant n(o_i)/2$.

Through this chain of reasoning, $n(o_t) \leqslant n/2^t$. But $n(o_t)$ (recall $o_t = p$) is at least 1. Hence,

$$1 \leqslant n(o_t) \leqslant \frac{n}{2^t} \implies 2^t \leqslant n \implies t \leqslant \log_2 n,$$

showing that *root*(p) visits at most $1 + \log_2 n \in O(\log n)$ objects before stopping at the root. ■

In code, we can maintain the sizes as follows:

```java
public class UnionBySizeUF implements UnionFind {
  private int[] id;
  private int[] sz;

  public UnionBySizeUF(int n) {
    id = new int[n];
    sz = new int[n];
    for (int i=0;i<n;i++) {
      id[i] = i;
      sz[i] = 1;
    }
  }

  int root(int p) {
    int pid = id[p];
    while (pid!=id[pid])
      pid = id[pid];
    return pid;
  }

  public int find(int p) { return root(p); }

  public void union(int p, int q) {
    int rp = root(p), rq = root(q);
    if (sz[rp] > sz[rq]) {
      // make rp the parent of rq
      id[rq] = rp; sz[rp] += sz[rq];
    }
    else {
      // symmetrically, make rq the parent of rp
      id[rp] = rq; sz[rq] += sz[rp];
    }
  }
}
```

Using this observation, we have $O(\log n)$ for both union and find. We can actually do much better (we'll revisit this topic in Algorithms).