

# Union Find - Recitation 11

January 11, 2023

## 1 Union Find.

We have mentioned that for finding efficiently the minimal spanning tree using kruskal, one has to answer quickly about whether a pair of vertices  $v, u$  share the same connectivity component. In this recitation we will present a data structure that will allow us both querying the belonging of given item and merging groups at efficient time cost.

The problem is defined as follows. Given  $n$  items  $x_1 \dots x_n$  we would like to maintain the partition of them into disjoint sets by supporting the following operations:

1.  $\text{Make-Set}(x)$  create an empty set whose only member is  $x$ . We could assume that this operation can be called over  $x$  only once.
2.  $\text{Union}(x, y)$  merge the set which contains  $x$  with the one which contains  $y$ .
3.  $\text{Find-Set}(x)$  returns a pointer to the set holding  $x$ .

Notice that the naive implementation using pointers array,  $A$ , defined to store at place  $i$  a pointer to the set containing  $x$  can perform the  $\text{Find-Set}$  operation at  $O(1)$ . The bottleneck of that implementation is that the merging will require from us to run over the whole items and change their corresponding pointer at  $A$  one by one. Namely, a running time cost of  $\Theta(n)$  time. Let's review a different approach:

**Linked Lists Implementation.** One way to have a non-trivial improvement is to associate for each set a linked list storing all the elements belonging to the set. Each node of those linked lists contains, additionally to its value and its sibling pointer, also a pointer for the list itself (the set). Consider again the merging operation. It's clear that having those lists allow us to union sets by iterating and updating only the elements belong to them. Still one more trick is needed for achieving a good running cost.

**Union( $x, y$ )**

```
1 if size A[x] > size A[y] then
2   size A[x] ← size A[x] + size A[y]
3   for z ∈ A[y] do
4     A[z] ← A[x]
5   A[x] ← A[x] ∪ A[y] // O(1) concatenation of linked lists.
6 else
7   Union(y, x)
```

Clearly, executing the above over sets of linear size requires at least linear time. Let's analyze what happens when merging  $n$  times. As we have already seen at graphs, runtime can be measured by counting the total number of operations that each item/vertex does along the whole running. So we can ask ourselves how many times does an item change his location and his set pointer. Assume that at the time when  $x$  were

changed  $A[x]$  contained (before the merging)  $t$  elements then immediately after that  $A[x]$  will store at least  $2t$  elements:

$$\text{size } A^{(t+1)}[x] \leftarrow \text{size } A^{(t)}[x] + \text{size } A^{(t)}[y] \geq 2A^{(t)}[x]$$

Hence, if we will list down the sizes of the  $x$ 's set at the moments merging had occurred we could write only  $\log n$  numbers before the exceeding the maximal size ( $n$ ). That proves that the number of times vertex changed his pointer is bounded by  $\log n$ , and the total number of actions costs at most  $\Theta(n \log n)$ .

Notice that in the case in which  $m = O(1)$  we will still pay much more than needed. Anyhow, the next implementation is going to give us (eventually) much faster algorithm.



Figure 1: Representing  $G$  by array of adjacency lists.

**Forest Implementation.** Instead of associating each set with a linked list, one might attach a forest. The vertices hold the values of the items, and we could think about the root of each tree as the representative of the tree. If two vertices  $x, y$  share the same root then it's clear that they belong to the same set. At the initialization stage Make-Set defines the vertices as roots of trivial trees (single root without any descendants). Then the find method is:

**Find( $x$ )**

```

1 while  $\pi(x) \neq \text{None}$  do
2    $x \leftarrow \pi(x)$ 
3 Return  $x$ 

```

We will see that for having the last improvement one slightly change should be set. But before that let's try to mimic the decision rule above. Even though we could define a size field for each root and get the same algorithm as above, instead we will define another field which from first sight looks identical. Let the  $\text{rank}(v)$  of the node  $v$  be height of the  $v$ . Recall that tree's height is defined to be the longest path from the root to one of the vertices.

**Union By Rank Heuristic<sup>1</sup>** . So as we said, first we are going to ensure how to mimic the  $\log n$  complexity proof under the forest implementation.

<sup>1</sup>Cormen calls that rule a heuristic, but please notice that heuristics usually refers to methods that seem to be efficient empirically, yet it doesn't clear how to prove their advantage mathematically. Still, in that course we stick to Cormen terminology.

**Union( $x, y$ )**

```

1  $x \leftarrow \text{Find}(x)$ 
2  $y \leftarrow \text{Find}(y)$ 
3 if  $x \neq y$  then
4   if  $\text{rank}(y) < \text{rank}(x)$  then
5      $\pi(y) \leftarrow x$ 
6   else if  $\text{rank}(y) = \text{rank}(x)$  then
7      $\pi(y) \leftarrow x$ 
8      $\text{rank}(x) \leftarrow \text{rank}(x) + 1$ 
9   else
10     $\pi(x) \leftarrow y$ 

```

The decision rule at line (4-8) preserves the correctness of the following claim: Claim, let  $S(r)$  a lower bound over the size of tree at rank  $r$ . Then  $M(r+1) \geq 2M(r)$ . The proof left as exercise. Assuming the correctness of the claim it holds that  $M(\log n) \geq n$ . So it immediately follows that the running time take at most  $n \log n$ . Actually we could get even a tighter bound by notice that a single query is bounded by the rank. and therefore, the total cost is at most  $m \cdot \log n$ .

**Path Compression Heuristic.** The final trick to yield a sub-logarithmic time algorithm is to compress the path on which we have already passed and by that reduce the number of duplicated transitions.

**Find( $x$ )**

```

1 if  $\pi(x) \neq \text{None}$  then
2    $\pi(x) \leftarrow \text{Find}(\pi(x))$ 
3 Return  $\pi(x)$ 

```

Let's analyse the query cost by counting the edges on which the algorithm went over. Denote by  $\text{find}(v^{(t)})$  the query which was requested at time  $t$  and let  $P^{(t)} = v, v_2, \dots, v_k$  be the vertices path on which the algorithm climb from  $v$  up to his root. Now, observe that by compressing the path the ranks of the vertices in  $P$  must be distinct. Now consider any partition of the line into set of buckets (segments)  $\mathcal{B} = \{B_i | B_i = [b_i, b_{i+1}]\}$ .

$$\begin{aligned}
T(n, m) &= \text{direct parent move} + \text{climbing moves} = \\
&= \text{direct parent move} + \text{stage exchange} + \text{inner stage} = \\
&\leq m + m \cdot |\mathcal{B}| + \sum_{B \in \mathcal{B}} \text{steps inside } B \\
&\leq m + m \cdot |\mathcal{B}| + \sum_{B \in \mathcal{B}} \sum_{\text{rank}(u) \in B} \text{steps inside } B \text{ started at } u \\
&\leq m + m \cdot |\mathcal{B}| + \sum_{B \in \mathcal{B}} \sum_{\text{rank}(u) \in B} |B|
\end{aligned}$$

For example consider our last calculation, In which we divided the ranges of ranks into  $\log n$  buckets at length 1,  $B_r = \{r\}$ , then as the size of the subtrees at rank  $r$  is at least  $2^r$  we have that the size of  $|B_r|$  is at most  $\frac{n}{2^r}$  and that's why:

$$\sum_{B \in \mathcal{B}} \sum_{\text{rank}(u) \in B} |B| \leq \sum_{b \in \{[i] | i \in [\log n]\}} \frac{n}{2^r} \cdot 1 \leq 2 \cdot n$$

So the total time is at most  $m + m \log n + 2n = \Theta(n)$ . And if we would take the  $\log \log n$  buckets such that  $B_i$  store the  $i$ th  $n / \log \log n$  numbers. Then the sum above will become:

$$m + m \cdot \log \log n + \sum_{b \in \mathcal{B}} \sum_{\text{rank}(u) \in B} |B| \leq \sum_{b \in \{B \in \mathcal{B}\}} \frac{n}{2^{\frac{i \log n}{\log \log n}}} \cdot \log \log n \leq 2 \cdot n$$

$$m + m \cdot \frac{\log n}{\log \log n}$$