Union Find - Recitation 11

January 11, 2023

1 Union Find.

We have mentioned that to find 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 to query the belonging of a given item and merge groups at an efficient time cost.

The problem defines as follows. Given n items $x_1...x_n$, we would like to maintain the partition of them into disjoints sets by supporting the following operations:

1. 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. Union(x, y) merge the set which contains x with the one which contains y.
3. Find-Set(x) returns a pointer to the set holding x.

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

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

Uinon(x, y)

```
if $size\ A[x] \geq sizeA[y]$ then
    size $A[x] \leftarrow$ size $A[x]$ + size $A[y]$
    for $z \in A[y]$ do
        $A[z] \leftarrow A[x]$
    $A[x] \leftarrow A[x] \cup A[y]$ // $O(1)$ concatenation of linked lists.
else
    Union $(y, x)$
```

Executing the above over sets at linear size requires at least linear time. Let's analyze what happens when merging n times. As we have seen in graphs, the runtime can be measured by counting the total number of operations each item/vertex does along the whole running. So we can ask ourselves how many times an item change its location and its set pointer. Assume that at the time when x were changed $A[x]$ contains (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 list down the sizes of the x's set at the moments merging occurred, we could write only $\log n$ numbers before exceeding the maximal size (n). That proves that the number of times the 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) a much faster algorithm.

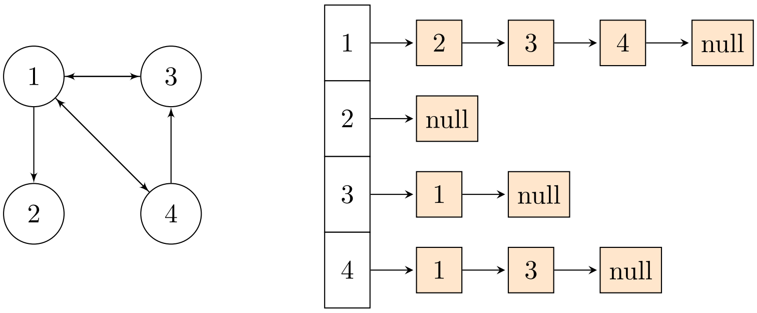

Figure 1: Presenting G by an array of adjacency lists.

Forest Implementation. Instead of associating each set with a linked list, one might attach a first. The vertices hold the values of the items, and we could think about the root of each tree as the represtive of the tree. If two vertices x, y share the same root, then it's clear 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)

```
while $\pi(x) \neq None$ do
    $x \leftarrow \pi(x)$
Return $x$
```

We will see that a slight change should be set for the last improvement. But before that, let's try to mimic the decision rule above. Even those, we could define a size field for each root and get the same algorithm as above. Instead, we will define another field that, from first sight, looks identical. Let the rank(v) of the node v be the height of the v. Recall that tree's height defines to be the longest path from the root to one of the vertices.

Union By Rank Huristic[1] . So as we said, first, we will ensure how to mimic the $\log n$ complexity proof under the first implementation.

[1] Corman 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 Corman terminology.

Uinon(x, y)

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

The decision rule in lines (4-8) preserves the correctness of the following claim: Claim, let $S(r)$ a lower bound over the size of the tree at rank r. Then $M(r+1) \geq 2M(r)$. The proof is left as an exercise. Assuming the correctness of the claim, it holds that $M(\log n) \geq n$. So it immediately follows that the running time takes at most $n \log n$. We could get even tighter bound by noticing that the rank bounds a single query. 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 brunch on which we have already passed and reduce the number of duplicated transitions.

Find(x)

```
if  $\pi(x) \neq None$ then
    $\pi(x) \leftarrow$Find$(\pi(x))$
Return $\pi(x)$
```

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

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

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:

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

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 stores the ith $n/\log\log n$ numbers. Then the sum above will become:

$$\begin{aligned}
&\sum_{b\in\mathcal{B}}\sum_{rank(u)\in B}|B| \le \sum_{b\in\{B\in\mathcal{B}\}} \frac{n}{2^{\frac{i\log n}{\log\log n}}}\cdot\log\log n \le 2\cdot n \\
&\le n\sum_{b\in\{B\in\mathcal{B}\}} 1\cdot\cdot\log\log n \le n\left(\log\log n\right)^2 \\
&\Rightarrow m + m\cdot\log\log n + n\left(\log\log n\right)^2
\end{aligned}$$

Could we do even better? Yes, Consider a nonunifom parttion $B_r = \{r, r+1 \ldots 2^r\}$. So first question one should ask is, what is $|\mathcal{B}|$? ($\log^*(n)$). On the other hand, the number of vertices in which their rank belongs to the ith bucket is at most:

$$\begin{aligned}
&\text{maximal number of nodes at rank } r + \text{ maximal number of nodes at rank } (r+1) + \\
&\text{maximal number of nodes at rank } (r+2) + \ldots + \text{ maximal number of nodes at rank } 2^r \\
&\frac{n}{2^r} + \frac{n}{2^{r+1}} + \frac{n}{2^{r+2}} + \ldots + \frac{n}{2^{2^r}} \Rightarrow |\{v\in B_r\}| \le 2\cdot\frac{n}{2^r}
\end{aligned}$$

$$\sum_{b\in\mathcal{B}}\sum_{rank(u)\in B}|B| \le \sum_{b\in\mathcal{B}}\sum_{rank(u)\in B}\frac{2n}{2^r}|B| \le \sum_{b\in\mathcal{B}}\sum_{rank(u)\in B} 2n \le \log^{(*)}(n)\cdot 2n$$