Union Find - Recitation 11

January 11, 2023

1 Union Find.

We have mentioned that for finding efficently the minimal spanning tree using kruskal, one has to answer quickly about wheter a pair of vertices v, u share the same connactivity commponent. In this recitation we will presnt a datastructure that will allow us both quering the belonging of given item and mergging groups at efficent time cost.

The problem define as follow. Given n items $x_1 ... x_n$ we would like to maintain the parttion of them into disjointss 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 navie immpleamntion using pointres 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 bottle neck of that immplemntion is that the mergging will require from us to run over the whole itmes and changes their corresponding pointer at A one by one. Namly, a running time cost of $\Theta(n)$ time. Let's review a diffrent approch:

Linked Lists Immplemntation. One way to have a non-trival improvmeant is to associate for each set a linked list storing all the elemnents belonging to the set. Each node of those linked lists contains, additilly to it's value and it's sibling pointer, also a pointer for the list itself (the set). Consider again the mergging operation. it's clear that having those lists allow us to uinfind sets by iterating and updating only the elememnts belong to them. Still one more trick is needed for achiving a good running cost.

Uinon(x, y)

```
1 if size A[x] ≥ sizeA[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 at linear size require at least linear time. Let's anlayze what happens when mergging n times. As we have allredy seen at graphs, runtime can be measured by counting the total number of operations that each item/vertex do along the whole running. So we can ask our selfes how many times does an item change his location and his set pointer. Assume that at the time when x were

changed $A[x]$ containd (before the mergging) t elemenets then immdetly 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 momments mergging had occured we could write only $\log n$ nmbers before the excciding 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 $\Theta(n \log n)$.

Noitce that that in the case in which $m = O(1)$ we will still pay much more than nedded. Any how, the next implemmntion is going to give us (enventuly) much faster algorithm.

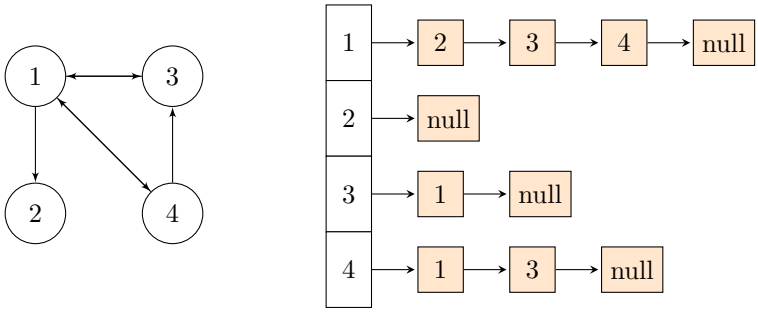

Figure 1: Persenting G by array of adjacency lists.

Forest Immplemntation. Insted of associating each set with a linked list, one might attach a forst. The vertices holds 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 than it's clear than they are belong to the same set. At the initialliztion stage Make-Set defines the vertices as roots of trival trees (single root without any descendants), Then the find method is:

Find(x)

```
1 while π(x) ≠ None do
2     x ← π(x)
3 Return x
```

We will see that for having the last improvment one slightly change should be setted. But before that lets try to mimic the decsion rule above. Even those we could define a size field for each root and get the same algortihm as above, Insted we will define another field which from first sight looks indentical. Let the rank(v) of the node v be hight of the v. Recall that tree's height define to be the logest path from the root to one of the vertices.

Union By Rank Huristic[1] . So as we said, first we are going to ensure how to mimic the $\log n$ complexitiy proof under the forst immplemntion.

[1] Corman calls that rule an hursitic, but please noitce that huristics are usually refers to methods that seemes to be efficent ampiricly, yet it doesn't clear how to prove their advantage mathmatclly. Still, in that course we stick to Corman triminalogy.

Uinon(x, y)

```
1  x ← Find(x)
2  y ← Find(y)
3  if x ≠ y then
4  |  if rank(y) < rank(x) then
5  |  |  π(y) ← x
6  |  else if rank(y) = rank(x) then
7  |  |  π(y) ← x
8  |  |  rank(x) ← rank(x) + 1
9  |  else
10 |  |  π(x) ← y
```

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

Path Compression Huristic. The final trick to yield a sub-logaritmic time algorithm is to compress the bruch on which we have allready passed and by that reduce the number of duplicated transitions.

Find(x)

```
1  if π(x) ≠ None then
2  |  π(x) ←Find(π(x))
3  Return π(x)
```

Let's analyses the queris cost by counting the edges on which the algorithm went over. Denote by find$(v^{(t)})$ 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 climbe from v up to his root. Now, observes that by comperssing the path the ranks of the verticis in P must be distincit. Now consdier any parttion of the line into set of buckets (segements) $\mathcal{B} = \{B_i | B_i = [b_i, b_{i+1}]\}$.

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

For example conisder 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 store the ith $n/\log \log n$ numbers. Then the sum above will become:

$$\sum_{b \in \mathcal{B}} \sum_{rank(u) \in B} |B| \leq \sum_{b \in \{B \in \mathcal{B}\}} \frac{n}{2^{\frac{i \log n}{\log \log n}}} \cdot \log \log n \leq 2 \cdot n$$
$$\leq n \sum_{b \in \{B \in \mathcal{B}\}} 1 \cdot \cdot \log \log n \leq n (\log \log n)^2$$
$$\Rightarrow m + m \cdot \log \log n + n (\log \log n)^2$$