A Lecture on Disjoint-Set

Moeez Muhammad, Parth Gupta

March 20, 2023

This is the augmented transcript of a lecture given by Luc Devroye on the 14th of March 2023 for a Data Structures and Algorithms class (COMP 252). The subject was disjoint sets.

Introduction

Definition 1. A **Disjoint-Set** (also called union-find and sometimes abbreviated DSU) is an abstract data type that maintains a partition of n points. It has the following operations:

1. MAKESET(x): places x in a set by itself.
2. FIND(x) or FINDSET(x): find set to which x belongs (returns **representative** of the set).
3. UNION(A, B) or UNION(x, y): join sets A and B. Note that in the other notation, x is the representative of set A and y is the representative of set B.

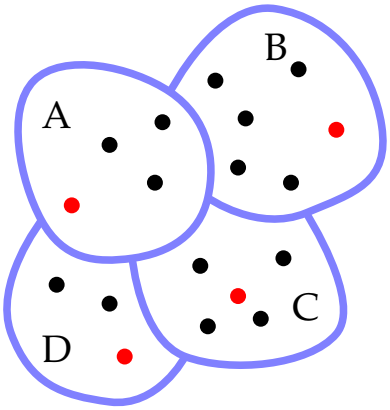


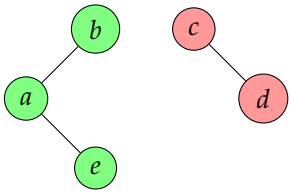
Figure 1: Example of a Disjoint-Set structure where the red coloured element is the representative of the set.

Definition 2. A **representative** of a set is some member of the set that is used to identify it. In some cases, it does not matter what element we pick to be the representative whereas in other cases there are rules (such as picking the smallest number in the set) for picking the representative. Regardless, the important thing is that FINDSET always returns the same representative for any element in the set between changes to the data structure (i.e. FINDSET(x) = FINDSET(y) $\iff$ x and y in the same set).

Applications

One of the many applications of disjoint-sets is to construct equivalence classes.

Example 3. Suppose we are given a set of equivalences: $S = \{a \equiv b, c \equiv d, a \equiv e\}$. We can partition S into into equivalence classes as follows where the colour denotes the equivalence class.



Note that this example is the same as determining connected components in a graph as an edge denotes an equivalence relation and a connected component denotes an equivalence class. Depth-first search can be used to find connected components faster, but disjoint-set is faster when you want to maintain the equivalence classes/connected components dynamically under edge additions.

We present the following algorithm that can be used to construct equivalence classes where S is the set of equivalences:

EQUIVALENCE CLASSES(x)

1 $\forall x$ do : MAKESET(x)
2 $\forall$ equivalence relations $(x \equiv y) \in S$ do:
3 $\quad a =$ FIND(x)
4 $\quad b =$ FIND(y)
5 $\quad a \neq b$ then UNION(a, b)

Example 4. The FORTRAN programming language has an instruction called "EQUIVALENCE" which takes variables (or arrays) separated by commas as an argument and specified that these variables shared the same block of memory.

Example 5. A screen can be thought of as a grid of pixels and each pixel has a grey level which indicates the brightness of a pixel. We can define an equivalence relation on pixels as follows: if two adjacent pixels have the same grey level, they are equivalent. So by forming equivalence classes of pixels, programmers can easily manipulate the grey scale of the movie frames with respect to time by doing a level order traversal. See Figure 2.

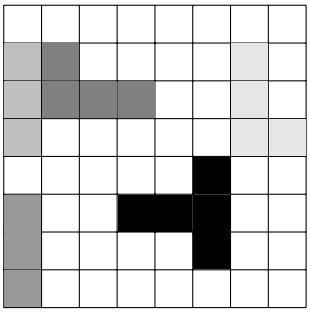


Figure 2: Example of an old movie frame shown on TV with adjacent pixels that have the same colour represent equivalence relation.

Implementations

Array Implementation

We can store $A[1], ..., A[n]$ where $A[i]$ stores the set number of element i (usually, this is the index of the representative of the set). The time complexity of FIND is $O(1)$ while UNION is $\Omega(n)$ in the worst-case.

Remark 6. If we store the set sizes with the representatives, and change the labels of the smallest set, then each x can change labels at most $\log_2(n)$ times. Hence, the complexity over m operations is bounded by $m + n \log_2 n$.

Linked List Implementation

Each set can be encoded as a linked list with the first node being the representative of the set. Each node is augmented with a pointer to

the representative which gives FIND $O(1)$ complexity. See Figure 3.

To implement the UNION operation, it is ideal to merge the smaller list into the bigger one so the time will be equivalent to length of the smallest of the two lists and each operation involves setting the augmented pointer to the representative of the bigger list. Hence giving $\Omega(n)$ complexity.

Remark 7. The total time spent on UNION is $O(n \log(n))$. This is because if we fix x (an element of the set), its contribution to the complexity is the number of times it changes its pointer (which is 1 time unit). Every time x changes its pointer, it goes to a set of bigger size, so it at least doubles (ex. going from a set of size 2 to a set of size 4). Since we can only double $\log(n)$ times and there are n elements, it follows that the complexity of UNION is $O(n \log(n))$.

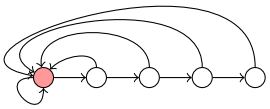


Figure 3: Implementation of a set where the red node is the representative.

Parent-pointer trees and forests

Most implementations of disjoint-set instead use parent-pointer trees[1]. Each tree in the forest represents a single set, and each element holds a pointer to its parent element (see figure 4). The root is the representative of the set and points to itself.

The operations still have simple implementations with a parent-pointer forest structure. We wish to do something similar to merging small to large like with linked lists. To do this, we introduce the notion of RANK for each node, which will be a useful loose upper bound on the height of the subtree that's easy to maintain. We'll give the code first and then prove properties about RANK.

[1] First introduced by Galler and Fisher [1964]

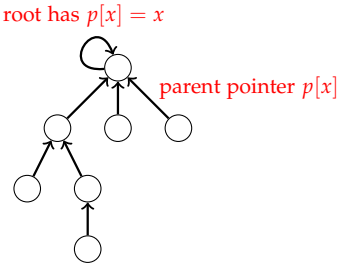


Figure 4: A set represented as a parent-pointer tree

MAKESET(x)

```
1  p[x] = x   // p[x] represents the parent of x, where x is the node
2  RANK[x] = 0
```

FIND(x)

```
1  while p[x] ≠ x :
2        x = p[x]
3  return x
```

In the following algorithms, x, y are the roots of the parent-pointer trees and we give two union algorithms: union by rank (first algorithm) and lazy-union (second algorithm).

UNION(x, y)

```
1  if x ≠ y :
2      if RANK[y] ≥ RANK[x] :
3          p[x] = y
4          if RANK[x] = RANK[y] :
5              RANK[y]+ = 1
6      else p[x] = y
```

UNION(x, y)

```
1  if x ≠ y :
2      p[x] = y
```

Properties of RANK

1. RANK$[x]$ always increases during its lifetime until x is no longer a root.
2. RANK$[p[x]] >$ RANK$[x]$ if x is not a root.
3. SIZE$[x] \geq 2^{\text{RANK}[x]}$ (where SIZE$[x]$ represents the size of the subtree with root x).

Proof. By induction on the number on UNION steps. Note that for 0 union steps, we have SIZE$[x] = 1$ since the only thing that has happened is MAKESET(x) and $2^{\text{RANK}[x]} = 1$, so we are done. Suppose now we do a UNION, we have two cases to deal with.

(a) If old RANK$[x] <$ RANK$[y]$, then

$$\text{new SIZE}[y] > \text{old SIZE}[y] \geq 2^{\text{old RANK}[y]} (\text{induction hypothesis}) = 2^{\text{new RANK}[y]}.$$

(see Figure 5)

(b) If old RANK$[x] =$ old RANK$[y]$, then

$$\begin{aligned} \text{new SIZE}[y] &= \text{old SIZE}[x] + \text{old SIZE}[y] \\ &\geq 2^{\text{old RANK}[x]} + 2^{\text{old RANK}[y]} \\ &= 2^{\text{old RANK}[y]+1} \\ &= 2^{\text{new RANK}[y]}. \end{aligned}$$

□

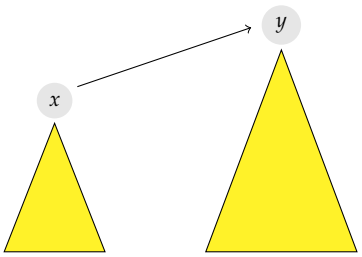


Figure 5: RANK[y] > RANK[x] so the size of the tree with root y is greater than the size of the tree with root x.

We briefly list out the worst-case complexities of the atomic operations:

1. FIND: $\Omega(n)$.
2. FIND if UNION is done by RANK: $O(\log_2(n))$

3. UNION, UNION BY RANK: $O(1)$.

Note that because the RANK increases as you go up a path, and the height increases by 1 each step up the path, the rank is always at least as much as the height. Therefore we have $\text{HEIGHT}[x] \leq \text{RANK}[x] \leq \log_2(\text{SIZE}[x])$. Therefore, FIND takes $O(\log n)$ time.

If actual size is maintained instead of rank, union-by-size would also yield $O(\log n)$ complexity for FIND if smaller sets are merged into larger ones. However, we will see in the following section on path compression that RANK is both much easier to maintain and easier to analyze.

Path compression

Right now, parent-pointer trees are as good as linked lists, just swapping the complexity of FIND and UNION. However, parent-pointer trees enable us to do something called *path compression* which will give us much better amortized time.

Path compression is simple: when we FIND an element, we put it and each of its ancestors on the top level by attaching them directly to the root, thereby making them faster to FIND next time.

This change is only a few extra lines to FIND to do a second pass up the path:

FIND(x)

```
1  y = x
2  while y ≠ p[y] :
3        y = p[y]
4  while y ≠ p[x] :
5        z = p[x]
6        p[x] = y
7        x = z
```

Those few lines, however, give us a big speed up.

Analysis of path compression

We prove the following claim[2]: For n MAKESET and m FIND and UNION operations, the total time taken is $O((m+n)\log^*(n))$ where $\log^*(n)$ is the *iterated log* or *log star* function, defined as $\log^*(n) = \min\{i : \underbrace{\log\log\ldots\log(n)}_{i \text{ times}} \leq 1\}$. We will also define the inverse function $A(n) = 2^{2^{\cdot^{\cdot^{\cdot^2}}}}$ (n times).

[2] Hopcroft and Ullman [1973]

Notice that $\log^*$ is an *extremely* slowly growing function as $\log^*(2) = 1$, $\log^*(4) = 2$, $\log^*(16) = 3$, $\log^*(65536) = 4$, and $\log^*(2^{65536}) = 5$.

Note that are an estimated $10^{18} \leq 2^{60}$ atoms in the observable universe. This means that $O((m+n)\log^*(n))$ might as well be linear for all practical inputs[3].

[3] In fact, notice that then $A(6) = 2^{2^{65536}}$ is too big to even be written in the observable universe

Proof. We first revise an earlier statement and claim that there are still $\leq \frac{n}{2^r}$ nodes of rank r, even with path compression. Remember that the earlier claim relied on the fact that a node of rank r had $\geq 2^r$ descendants. However, with path compression, subtrees are destroyed and every node on the path now only has one node: itself. So we have to be a little more careful.

Remember that the rank of a node only increases while the node is a root. Also, while a node is a root, the size of its subtree only increases. Therefore, every node of rank r has some set of nodes (at least 2^r) that were once under it and "explain" its rank being r. So, we just have to show that for all nodes of rank r, their sets of "explanation nodes" are disjoint. This is because a node of rank r only ever merges into other roots of rank r or greater, and if it merges into a root of rank r, the new root has rank $r+1$. Therefore, once path compression happens and the node loses its children, it will only ever lose them to nodes of higher rank. So there can be no overlap between nodes of the same rank.

Next, we divide the nodes by rank into intervals of $[A(i), A(i+1))$. We will then compute the total time taken by m finds by computing the number of edges traversed over all finds, since we pass over each traversed edge twice, so we process each edge in $O(1)$ time. To do this, we divide the edges into three different categories:

1. Those that end at the root
2. Those that cross bucket boundaries
3. Those that stay within a bucket

Each FIND only has one edge that ends at the root and so Type 1 edges only contribute $O(m)$ over m FINDS. In a given find, the path has length $O(\log n)$, and each edge that cross bucket boundaries increases the rank by 2^r but that can only happen $\log^*(\log_2(n)) = \log^*(n) - 1$ times[4]. Therefore, in a FIND there are $O(\log^*(n))$ edges of Type 2. So Type 2 edges contribute $O(m\log^*(n))$ over m FINDS.

[4] $\log^*(n)$ takes one additional log than $\log^*(\log_2(n))$ would

The analysis of Type 3 edges is a bit more involved. We'll count this by considering a single node, and counting the number of edges that stay within the same bucket of that node. So fix a node u with rank $r \in [A(i), A(i+1))$. Each edge that is traversed that starts at u ends at a parent of u. So consider the sequence $v_1 \dots v_k$ of parents of u within the same bucket over time. Remember that we are not considering edges ending at the root since we already counted those in

Type 1, so all v_i are not roots and have fixed rank at the time they are u's parent. Also, because ranks increase as you go up a path, and v_i are not roots, when path compression is performed, u is attached to an ancestor of v_i, and thus the rank of u's parent is strictly increasing over time. Therefore, all v_i must be distinct.

So k is bounded by the number of ranks in the bucket $[A(i), A(i+1))$ which is just $A(i+1) - A(i) \leq A(i+1)$ (notice that $A(i+1) \gg A(i)$). Therefore the number of Type 3 edges traversed in path compression per node is $\leq A(i+1)$. Therefore the total number of Type 3 edges seen is:

$$\sum_{\text{Bucket } i} \sum_{\text{node } n \in \text{Bucket } i} A(i+1).$$

Notice that $A(i+1)$ only depends on the bucket and so can be taken out of the sum over nodes. But we can express the number of nodes in a given bucket in a simpler form. Remember that for each r there are $\leq \frac{n}{2^r}$ nodes of rank r. Therefore, the number of nodes in bucket i is not more than

$$\sum_{r=A(i)}^{A(i+1)-1} \frac{n}{2^r} = \frac{n}{2^{A(i)}} + \frac{n}{2^{A(i)+1}} + \cdots + \frac{n}{2^{A(i+1)-1}},$$

which is a geometric series of ratio $1/2$. We take out the common factor $\frac{n}{2^{A(i)}} = \frac{n}{A(i+1)}$ (remember the definition for A) and obtain:

$$\frac{n}{A(i+1)} \sum_{r=0}^{A(i+1)-A(i)-1} \frac{1}{2^r}.$$

We bound the finite sum by the infinite series and obtain

$$\leq \frac{n}{A(i+1)} \sum_{r=0}^{\infty} \frac{1}{2^r} = \frac{2n}{A(i+1)}.$$

Therefore, our original sum becomes

$$\sum_{\text{Bucket } i} \frac{2n}{A(i+1)} A(i+1) = \sum_{\text{Bucket } i} 2n = O(2n \log^*(n))$$

because there are $O(\log^*(n))$ buckets. Therefore the sum over all types of edges is $O(\log^*(n))$. □

Ackermann's function

$O((m+n)\log^*(n))$ is not actually a tight upper bound. It was shown by Tarjan [1975] that the running time is actually $\Theta((m+n)\alpha(n))$[5], where $\alpha(n)$ is an even slower growing function than $\log^*$, and is defined as the inverse of the *Ackermann function*, which is similar to, but different from our $A(n)$. There exist multiple variations of the

[5] Later Tarjan [1979] and Fredman and Saks [1989] showed that this is the theoretical lower bound for all general disjoint-set structures

Ackermann function and its inverse, but one is the following from Cormen et al. [2009], actually defined as a sequence of functions. For integers $k \geq 0$ and $j \geq 1$, let $A_k(j)$ be:

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1 \end{cases}$$

where $f^{(j)}(x)$ is the function f applied j times to x. We then define the inverse $\alpha(n) := \min\{k : A_k(1) \geq n\}$. We can do some calculations to show how slowly growing $\alpha(n)$ is[6]. Using the definition, it's quick to show that $A_1(j) = 2j+1$ and $A_2(j) = 2^{j+1}(j+1) - 1$. Then, we can compute $A_3(1) = A_2(A_2(1)) = A_2(7) = 2^8 \cdot 8 - 1 = 2047 > A(3) = 16$. Therefore, we can see that $\log^*(2047) = 4$ but $\alpha(2047) = 3$. After this point, the discrepancy becomes much larger.

$A_3(j)$ is repeated application of $A_2(j)$, and so corresponds roughly to repeated exponentiation (since in each application of A_2 you exponentiate by 2), and you can show that $A_3(j)$ is much bigger than $A(j+2)$. Then, we can compute $A_4(1) = A_3(A_3(1)) = A_3(2047) \gg A(2049)$. $A(2049)$ is a mind-bogglingly large number, and of course $\log^*(A(2049)) = 2049$ but $\alpha(A(2049)) = 4$.

The full proof of the $\alpha(n)$ bound is much more intricate, but can be found in Cormen et al. [2009].

[6] $\alpha(n)$ is actually $o(\log^*(n))$

References

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844.

M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, STOC '89, page 345–354, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913078. DOI: 10.1145/73007.73040. URL https://doi.org/10.1145/73007.73040.

Bernard A. Galler and Michael J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, may 1964. ISSN 0001-0782. DOI: 10.1145/364099.364331. URL https://doi.org/10.1145/364099.364331.

J. E. Hopcroft and J. D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973. DOI: 10.1137/0202024. URL https://doi.org/10.1137/0202024.

Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, mar 1984. ISSN 0004-5411. DOI: 10.1145/62.2160. URL https://doi.org/10.1145/62.2160.

Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, apr 1975. ISSN 0004-5411. DOI: 10.1145/321879.321884. URL https://doi.org/10.1145/321879.321884.

Robert Endre Tarjan. A class of algorithms which require non-linear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–127, 1979. ISSN 0022-0000. DOI: https://doi.org/10.1016/0022-0000(79)90042-4. URL https://www.sciencedirect.com/science/article/pii/0022000079900424.

Wikipedia. Disjoint-set data strucuture — Wikipedia, the free encyclopedia, 2023. URL https://en.wikipedia.org/wiki/Disjoint-set_data_structure. [Online; accessed 20-March-2023].