CS2040S: Data Structures and Algorithms

Tutorial Problems for Week 8: Union-Find

For: 4 Oct 2021, Tutorial 6

Solution: Secret! Shhhh... This is the solutions sheet.

Problem 1. Largest set

Given a UFDS initialised with n disjoint sets, what is the maximum possible rank h that can be obtained from calling any combination of `unionSet(i, j)` and/or `findSet(i)` operations? Assume that both the path-compression and union-by-rank heuristics are used. Justify your answer.

Solution: For the operation `unionSet(i, j)`, the rank of the resulting set be higher than that of set i and set j if and only if the size of both sets are the same. The resulting set will have 1 rank higher than that of the two sets. For the operation `findSet(i)`, there is no way of increasing the rank of the set. In fact, the rank of the set may decrease due to path-compression.

Thus, the maximum height is obtained when we by calling `unionSet(i, j)` on $\lfloor \frac{n}{2} \rfloor$ distinct pairs from the n disjoint sets of rank 0, to obtain $\lfloor \frac{n}{2} \rfloor$ disjoint sets of rank 1. Similarly, we call `unionSet(i, j)` again on the remaining sets that have $\lfloor \frac{n}{4} \rfloor$ distinct pairs, to obtain $\lfloor \frac{n}{4} \rfloor$ disjoint sets of rank 2. Repeatedly applying this set of operations, we will eventually obtain one set of rank $\lfloor \log_2 n \rfloor$ (with some leftovers if the number of sets are not even at any point).

Problem 2. Intergalatic wars

n intergalactic warlords scattered throughout the galaxy have been vying to conquer each other and become the intergalactic overlord. Each of them start out being only the overlord of his/her own world. A conquered warlord will have all the worlds under him/her added to the worlds of the victorious warlord. Each starting world may be represented by a positive integer from 0 to $n - 1$. Each warlord is also represented by an integer from 0 to $n - 1$. At the beginning warlord x will only be the conqueror of world x.

Problem 2.a. Since a warlord x can have many worlds under him/her, describe the data structure(s) and associated operations you need so that the warlord can

i) Add worlds of a conquered warlord y to his/her domain in $< O(\log n)$ time.

ii) Answer the query of whether a world is under him/her in $< O(\log n)$ time.

Solution: One way to solve this is to use a UFDS + an integer array (used as a DAT). At the start each world belonging to a warlord is a disjoint set represented by its number. An integer array W of size n will map warlord (index) to his/her "dominant" world (value stored at index). At the beginning the starting world of the warlord is the dominant world.

For when a warlord x conquers another warlord y, perform `unionSet(W[x],W[y])` in $O(1)$ time. If `W[x]` is added under `W[y]` set dominant world of x to be `W[y]` and then set `W[y]` to be -1 in $O(1)$ time. Thus total time is $O(1)$.

To answer query of whether a world z is under x, simply call `isSameSet(W[x],z)`. Time is $O(1)$.

Problem 2.b. Make modifications/additional operations to the data structure you described in 2(a) so that a warlord can answer in $< O(\log n)$ time the most important question - Am I now the intergalactic overlord (meaning have I conquered all the N worlds)? Justify your answer.

Solution: Keep a count `numWarlord`. At the start, `numWarlord = n`. Whenever a warlord y is conquered, i.e `W[y] == -1,` decrement `numWarlord`. Can be done in $O(1)$ without affecting the other operations in 2(a). To answer a query of whether a warlord x is now the galactic overlord, check if `W[x] >= 0` and `numWarlord == 1`.

Problem 3. Number candies (Adapted from AY19/20 Sem 1 Final Exam)
The current rage among children are number candies. As the name suggest, these candies are given a number from 1 to n, where n is an extremely huge number. The ultimate goal is to collect all n candies. This is very difficult as the candies carried by any store is random (and can have a lot of repeats). To make it easier, the manufacturer has a competition to gather all sequential candies from 1 to m, where m is much smaller than n. The winner will be given all the n candies.

One particularly rich kid has hired h helpers, where $m < h < n$. Each helper will buy one candy from each of the h available stores. He hopes that doing so will allow him to get all m candies. However, he does not know how to figure out if he has all candies from 1 to m.

To help him solve the problem, his butler suggests him to answer the following query every time one of the helpers return with a candy: **What is the largest numbered candy he has that is in an unbroken sequence from candy 1?** (If greater than m, just return m. If he does not have candy 1, return -1.)

Design an algorithm that makes use of UFDS such that each of the h queries can be answered in $O(\alpha(h))$ time.

Solution: Suppose that currently, we have the following candies in our collection: $[1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 25, 26, 27, 114, 115, 116]$. Let every consecutive sequence form a set of candies. So in our collection above, there are four sets of candies: $[1, 2, 3, 4, 5, 6, 7]$, $[10, 11, 12, 13, 14]$, $[25, 26, 27]$, $[114, 115, 116]$.

At each query, a new candy is added to our collection. This may cause two sets to be merged together, if this new candy links up two initially disjoint consecutive sequences. For example, if we have the following candies in our collection initially: $[3, 4, 5, 6, 8, 9, 10, 11]$, with two sets $[3, 4, 5, 6$ and $8, 9, 10, 11$, and if candy 7 is added, then these two sets will be merged into one combined set $[3, 4, 5, 6, 7, 8, 9, 10, 11]$.

After every query, we want to report the length of the longest consecutive sequence of candies starting from candy 1. Observe that this is effectively that largest candy in the same set as candy 1. Note that after two sets are merged together (as described in the paragraph above), the length of the longest consecutive sequence of candies starting from candy 1 cannot decrease. So, we will try to maintain the maximum candy number for every set, and update this maximum when two sets are merged.

Initially, we place every candy in a separate set in the UFDS of size h. When we receive a new candy, we try to merge it with the set containing the candy on its left and right, if the sets exist, and update the value of the maximum candy in the set. After every query, we output the value of the largest candy in the same set as candy 1. The pseudocode (with modifications for the UFDS) is shown below. Here, the solution uses a UFDS of size h, but it is sufficient to only use a UFDS of size m.

```
Let par[1..h] be an array of integers  //parent array for UFDS
Let rank[1..h] be an array of integers // rank array for UFDS
Let max_candy[1..h] be an array of integers // we maintain
   the candy with the largest value in each set here
Let has_candy[1..h] be an array of booleans // this is to
   track if we already have a certain candy
Let A[i] denote the candy added in the ith query

function findSet(x)
    if par[x] == x
        return x
    return par[x] = findset(par[x])

function sameSet(x, y)
    return findSet(x) == findSet(y)

function mergeSet(x, y)
    x = findSet(x)
    y = findSet(y)
    if x == y
```

```
        return
    if rank[x] > rank[y]
        par[y] = x
        max_candy[x] = max(max_candy[x], max_candy[y])
    else
        par[x] = y
        if rank[x] == rank[y]
        rank[y]++
        max_candy[y] = max(max_candy[x], max_candy[y])

// UFDS initialisation
for i = 1 to h
    par[i] = i
    rank[i] = 0
    max_candy[i] = i
    has_candy[i] = false

for i = 1 to h
    if A[i] > h
        // if a candy has value higher than h, then we just
           ignore it
        // since it is impossible to get a consecutive
           sequence of candies
        // higher than h
        continue
    if has_candy[A[i]]
        // duplicate candy
        continue
    has_candy[A[i]] = true
    // some array out of bounds checking here omitted
    // since they are not essential to the solution
    if has_candy[A[i]-1]
        mergeSet(A[i], A[i]-1)
    if has_candy[A[i]+1]
        mergeSet(A[i], A[i]+1)
    if has_candy[1]
        print max_candy[findSet(1)]
    else
        print -1
```

Problem 4. Funfair Ride

You are operating a ride at a funfair. The ride contains n seats numbered from 1 to n arranged **clockwise** in a circle (i.e. seat i is followed (clockwise) by seat $i+1$ for $1 \leq i < n$, and seat n is followed (clockwise) by seat 1).

m children ($m \leq n$) numbered 1 to m arrive at the ride in sequence (from 1 to m). Each child wants a certain seat in the ride: the child numbered i wants seat s_i. However, there are some seats that are wanted by more than one child, so you have decided to allocate the seats in the following way:

When the ith child arrives:

- If seat s_i is not taken, allocate seat s_i to child i.
- Otherwise, get child i to walk clockwise starting from seat s_i, and allocate the first empty seat encountered to child i.

There are many children coming for the ride, so this process of getting each child to manually find another seat if the seat he wants is taken is slow. Fortunately, you have been given the seat preferences for all the children (all the s_i). Using a UFDS, give an algorithm to figure out the seat that a child i will eventually occupy. State the time complexity of your algorithm.

Solution: Observe that if seat s_i is occupied, then we want to allocate the next seat that is vacant down the line from seat s_i. If we group this block of occupied seats together, then we would be allocating the seat with the maximum seat number in this block + 1 to this child, since that would be the next empty seat. We can use a UFDS to keep all adjancent seats that are occupied together in the same set, as well as the highest numbered seat in each set. We can then easily find the next empty seat to allocate to each child.