CS330HW8

ras70

November 2017

Problem 3

In this problem we are asked to implement 3 functions, *Merge(A,B)*, *Count(A)*, and *Enter(A)*. To solve this problem I will use disjoint sets, implemented with trees with union-by-rank and path compression.

Quick Note: In the following algorithm descriptions below I will try to distinguish between my merge Method and the merge method on the disjoint sets. For clarity I will use a capital letter when referencing my Merge method and a lowercase for the disjoint set merge. In some cases I am more explicit than this an describe the disjoint set merge with a leading adjective.

Initialization

The problem will be initialized such that each person starts off in their own disjoint set. In the union-by-rank implementation of disjoint sets each set head has a stored rank. Let the algorithm also store a **count** value (integer) with each set head. This count value can be stored in an array of size n. Each of these count values would be initialized to 1 for each of the sets. The implementation, given this setup, of each of the 3 functions is described in detail below.

Algorithms

Merge(A,B): The algorithm will first call the *merge(A,B)* method built into the functionality of disjoint sets. The disjoint set implementation will call find twice, on A and B, to determine out which trees these nodes belong. If A and B are in the same tree the algorithm would commence. Let C and D be the heads of the sets to which A and B below, respectively. With the union-by-rank method the merge call will point D to C if C has a higher rank than D and point C to D if D has a higher rank than C.

My algorithm will do additional work when the above steps are completed. My algorithm will take the sum from the **count** of the two prior heads C and D and associate it with the new head chosen after the merge.

Simple pseudo-code is given below:

```
counts[]
```

```
Merge(A,B) {
    // Assume disjointSetMerge merges A and B
    //and return the # of the new root
    new_root = disjointSetMerge(A,B)  // new_root = new root after merge
    counts[new_root] = Count(A) +  Count(B)
}
```

Note: The pseudo-code above assumes the counts of the roots are implemented with an array called *counts*. Additionally, the code can be optimized by having $disjointSetMerge$ return the root of A and B, since it must calculate them anyways. This would prevent calling $find$ two additional times in the *Count* method.

Count(A): The count method will first find which tree/set A belongs by calling the disjoint set's built in $find$ method. Let E be the node returned by $find(A)$. Since E is a root/head it will have an associated **count** value. This value will be returned by the algorithm. Simple pseudo-code is given below:

```
counts[]

count(A) {
    root = find(A)
    return counts[root]
}
```

Enter(A): This method will find the root of the tree to which A belongs and decrease its count by one. The method will not remove A from the tree (the rational behind this is described in the Tombstoning section below). Simple pseudo-code is given below:

```
counts[]

Enter(A) {
    root = find(A)
    counts[root] = counts[root]--;
}
```

Comment about the Tombstoning Method

In the *Enter* method my algorithm never removes the entering student from the line. However, this implementation is correct because the problem states that *Merge* and *Count* will never be called on a student in the stadium. Given this constraint there is no need to delete the student from the tree. The count associated from the tree is never calculated using the tree's elements so it does not matter if the tree contains additional elements.

An alternative approach would to fully implement the tombstoning method by having each student have a flag, indicating whether or not they were in the

stadium. The *Enter* method would then alter this flag accordingly. However, again since *Merge* and *Count* will never be called on students in the stadium this tombstone flag is not needed.

Running time

Since I implemented this algorithm using disjoint sets with union-by-rank and path compression the running time for merge and find are bounded by $\alpha(n)$. All of my modifications include updating a count value, which if implemented with an array can be done in constant time. Therefore, the overall running time of the algorithm is $O(\alpha(n))$