CS 157: Assignment 2

Douglas R. Lanman
27 February 2006

Problem 1: Union-Find with Deletions

The union-find data structure, utilizing the "union-by-rank" and "path compression" heuristics, is one of the most efficient disjoint-set data structures known [1]. In general, union-find can be applied to any problem in which one must determine membership within a set. For instance, in class we studied its application within Kruskal's algorithm, where it was used to efficiently detect connected component of a graph. In general, union-find supports three primary operations: insertion, union, and find. As discussed by Kaplan *et al.*, in many applications it is also desirable to support deletion [3].

In this write-up we present "lazy-deletion": a straightforward modification of the classic union-find data structure which will allow deletions. In particular, lazy deletion will not alter the asymptotic worst-case running times of either the union or find procedures. Recall that the union-find data structure maintains a number of rooted trees. If a node has no children, then we can delete it without incident (i.e., free any memory or other resources associated with it). If a node has children then we cannot eliminate it immediately – otherwise its children would become isolated from the root node. As a result, we adopt the "lazy deletion" policy, in which nodes with children are marked for deletion and, as their children are removed over time, eventually deleted.

This write-up is organized into four sections. In Section 1 we will discuss the basic modifications that must be made to MAKE-SET and UNION to implement lazy deletion. In Section 2, the pseudocode for insertion and deletion will be presented. In Section 3, the necessary modifications for FIND-SET will be presented. Finally, in Section 4, the relative merits of lazy deletion will be discussed in the context of alternate implementations.

1 Modifying MAKE-SET and UNION

We can implement union-find with lazy deletion in a manner similar to [1]. As previously discussed, in order to implement lazy deletion we will need to keep track of the number of children for each node. In addition, we will need to add an attribute to each node that indicates whether it has been marked for deletion. As a result, MAKE-SET (from CLRS page 508) must be modified to initialize these new attributes as follows.

```
MAKE-SET(x)
1  p[x] ← x
2  rank[x] ← 0
3  children[x] ← 0
4  marked[x] ← 0
```

In this case, *children* is used to store the number of children for each node and *marked* is a binary attribute indicating whether the node has been marked for deletion. Clearly, lines 3 and 4 can execute in constant time. As a result, the asymptotic worst-case running time of MAKE-SET remains $\Theta(1)$, as shown in [2].

At this point, we turn our attention to the UNION procedure – used to join two disjoint-sets given pointers to a member in each. For this write-up we utilize the UNION procedure presented on CLRS page 508 (repeated below).

```
UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))
```

In this case UNION simply calls FIND-SET to determine the representatives for each set. Since the problem statement requires that our modifications do not alter the runtimes of either union or insertion, it is important to note that we cannot alter the runtime of FIND-SET either – since this would alter UNION. Notice that the LINK procedure called by UNION could potentially change the number of children for a given node (in addition to its rank). As a result, we introduce the following modifications.

```
LINK(x, y)
1  if rank[x] > rank[y]
2     then p[y] ← x
3          children[x] ← children[x] + children[y] + 1
4     else p[x] ← y
5          children[y] ← children[y] + children[x] + 1
6          if rank[x] = rank[y]
7             then rank[y] ← rank[y] + 1
```

The modifications on lines 3 and 5 ensure that the children counters are incremented correctly. Since the number of children will only increase for the new "parent" tree, we must increment by the size of its new "child" tree plus one (for the tree's root). In the absence of additional operations, such as INSERT, DELETE, or FIND-SET, these modifications correctly increment the number of children whenever two trees are joined.

For completeness, we briefly review the asymptotic running times of UNION and its LINK subroutine. First, notice that the modifications of LINK on lines 3 and 5 are $O(1)$. As a result, LINK continues to have $O(1)$ running time [1]. Examining the pseudocode for UNION, we find that it is composed of two calls to FIND-SET and a single call to LINK. As a result, its running time is only $O(1)$ in addition to the two calls to FIND-SET [2]. In conclusion, the asymptotic worst-case running time of UNION is identical to that of FIND-SET, which will be discussed in Section 3.

2 Implementing INSERT and DELETE

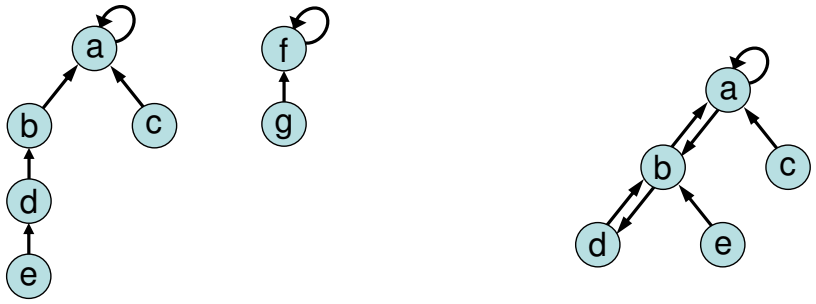
2.1 High-level Description

At this point, we have a basic union-find data structure that can keep track of its children. We now turn our attention to implementing insertion and deletion. As presented in [3], INSERT can be implemented trivially as follows.

```
INSERT(x, y)
1  MAKE-SET(x)
2  UNION(x, y)
```

First, we call MAKE-SET to form a singleton set for x. Afterwards, we insert this set into the tree containing y using UNION. The correctness of INSERT is immediate. Similarly, since MAKE-SET has $O(1)$ running time, the asymptotic worst-case running time of INSERT is identical to FIND-SET.

Before we implement deletion, lets consider some some characteristic examples. In Figure 1(a) notice that nodes c, e, and g could be deleted immediately, whereas nodes a, b, d, and f could only be marked for deletion. In either case, once a node has been "deleted", the number of children



(a) Example using lazy deletion (b) Example using child pointers

Figure 1: Examples considered for union-find with deletion

for each of its ancestors will decrease by one. For example, if we mark node d for deletion, the *children* attributes for nodes a and b must be decremented by one. Since both removing a node and marking a node for deletion decreases the number of children for any ancestors, we must also ensure that DELETE removes any nodes that have been marked for deletion and, as a result of a deletion, no longer have any children.

From these simple examples it is apparent that DELETE must accomplish three goals: (1) either immediately remove a node or mark it for deletion, (2) decrement all *children* attributes for the "deleted" node's ancestors, and (3) remove any nodes marked for deletion that no longer have any children. The following pseudocode for DELETE, and its CLEAN subroutine, accomplishes all of these goals.

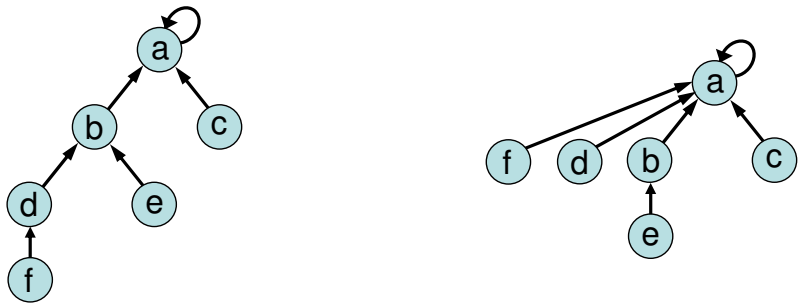
```
DELETE(x)
 1  if x ≠ p[x]
 2     then CLEAN(p[x])
 3  if children[x] ≠ 0
 4     then marked[x] ← 1
 5     else  FREE(x)

CLEAN(x)
 1  children[x] ← children[x] − 1
 2  if x ≠ p[x]
 3     then CLEAN(p[x])
 4  if marked[x] = 1
 5     then if children[x] = 0
 6             then FREE(x)
```

2.2 Proof of Correctness for DELETE

To begin our analysis, consider lines 3 through 5 of DELETE. In the absence of lines 1 and 2, this segment of pseudocode correctly implements lazy deletion – removing a node if it has no children or marking it for deletion if it does. Notice that, in this write-up, FREE is used to represent any operations which are necessary to free the memory or other resources associated with a node. *It is assumed that* FREE *has a constant running time* of $O(1)$.

As previously discussed, once a node has been deleted (or marked for deletion), we must decrement the *children* attributes of its ancestors and delete any marked nodes that no longer have children. These tasks are achieved on lines 1 and 2 of DELETE. Taking our inspiration from the



(a) Before FIND-SET(f) (b) After FIND-SET(f)

Figure 2: Result of path compression within FIND-SET

FIND-SET routine presented in CLRS on page 508, we recursively call CLEAN, starting at the "deleted" node, until we reach the root of the tree (where $p[x] = x$). Line 1 of CLEAN decreases the *children* attribute by one – ensuring that the children counters are correct for all ancestors. Notice that lines 4 through 6 of CLEAN will only be called *after* the *children* attributes have been updated. These lines check to see if any nodes marked for deletion no longer have any children and delete the node permanently if that is the case.

As an example, once again consider Figure 1(a). Initially, no nodes are marked for deletion and {a,b,c,d,e} have $\{4, 2, 0, 1, 0\}$ children, respectively. If we call DELETE(d), then nodes {a,b,c,d,e} will now have $\{3, 1, 0, 1, 0\}$ children. In addition node d will be marked for deletion. If we subsequently call DELETE(e), then nodes {a,b,c,d,e} will have $\{2, 0, 0, 0, 0\}$ children. Since node d was marked for deletion and now has no children, it will be deleted along with node e – exactly the outcome we desired.

2.3 Analysis of Running Time for DELETE

Since DELETE was derived from FIND-SET, we expect it will have a similar running time. It is important to note, however, that this implementation of DELETE does not perform path compression (i.e., the parent pointers are not set to the root after updating the *children* attributes). Since CLEAN will be called once for every node on the path between the "deleted" node and the root, the asymptotic worst-case running time will be proportional to the number of nodes on this path – similar to the analysis presented in [2] and [4]. Since path compression will not be preformed, there will be no amortized benefit for repeated calls to DELETE and the asymptotic worst-case running time is $O(\log n)$ – identical to FIND-SET without path compression [1]. Note that n is the number of nodes created by MAKE-SET.

3 Modifying FIND-SET

3.1 High-level Description

At this point, all that remains is to modify FIND-SET, without changing its runtime, to ensure: (1) that it updates the *children* attributes correctly and (2) that is removes any nodes that have been marked for deletion and no longer have any children due to path compression. As was done for deletion, we begin by considering a characteristic example. In Figure 2 we show a rooted tree and its configuration after FIND-SET(f). In general, the path compression heuristic will cause all ancestors between node f and the root to become immediate children of the root. As a result, if a node has no children (other than an ancestor of node f or node f itself) and is included on the

path between node f and the root, then it will become a leaf node after path compression. This property is important because it can cause marked nodes to become leaf nodes (and therefore able to be deleted). It is also important to note that FIND-SET(f) will modify the number of children for all nodes between node f and the root.

From this example it is apparent that FIND-SET must: (1) update the *children* attribute for all nodes between the search node and the root of the tree and (2) remove any marked nodes that no longer have children due to path compression. The following pseudocode achieves both of these goals.

```
FIND-SET(x)
 1  if x ≠ p[x]
 2     then p[x] ← FIND-SET(p[x], children[x] + 1)
 3  return p[x]

FIND-SET(x, c)
 1  parent ← p[x]
 2  if x ≠ p[x]
 3     then parent ← FIND-SET(p[x], children[x] + 1)
 4          p[x] ← parent
 5          children[x] ← children[x] − c
 6          if marked[x] = 1
 7             then if children[x] = 0
 8                     then FREE(x)
 9  return parent
```

3.2 Proof of Correctness for FIND-SET

To begin our analysis, consider FIND-SET(x). Notice that this pseudocode is nearly identical to CLRS page 508, except for the addition of a second argument in FIND-SET(x, c). This second form of FIND-SET uses the argument c to pass the number of children contained in a subtree. For example, if x is not the root node, then it has *children*$[x]$ children and, after path compression, the parent of x will no longer have x as a child; as a result, the parent of x will lose *children*$[x] + 1$ children. Line 5 of FIND-SET(x, c) is required to correctly update the *children* attributes as a result of path compression.

In addition to correctly updating the children counters for each node, FIND-SET must also remove any nodes marked for deletion that have become leaf nodes. This is accomplished on lines 6 through 8 of FIND-SET(x, c). First, notice that the node at which a FIND-SET is initiated will still have the same number of children after path compression. As a result, we are not required to check for deletion in FIND-SET(x). Alternatively, the number of children will change for all nodes between x and the root node. This is why the deletion is placed inside FIND-SET(x, c).

3.3 Analysis of Running Time for FIND-SET

Examining the pseudocode provided for FIND-SET(x) and its FIND-SET(x, c) subroutine, we see that only $O(1)$ modifications have been made to the original version from CLRS page 508. In particular, lines 5 through 8 of FIND-SET(x, c) have constant running time (assuming FREE is $O(1)$). In addition, passing the number of children using the argument c does not alter the asymptotic worst-case running time. In conclusion, we find that the running time of FIND-SET, and correspondingly UNION, is not altered by the modifications required for lazy deletion. As presented in [2], the asymptotic worst-case running time is proportional to the length of the path from the

search node to the root and is given by $\Theta(\log n)$ for a set with n nodes. It is important to note, however, that the amortized cost is only $O(\log^* n)$ [2].

4 Discussion

As presented, union-find with lazy deletion is a straightforward modification of the classic union-find data structure. In the previous sections we showed how the pseudocode from CLRS on page 508 could be modified to allow deletion without changing the asymptotic worst-case running times of UNION, INSERT, or FIND-SET. The greatest drawback of this approach, however, is that is doesn't achieve *immediate* deletion. When nodes are marked for deletion, they continue to consume memory (or other resources) until they no longer have any children. It is easy to think of examples where marked nodes could persist indefinitely. For example, if a root node is marked for deletion and there are no subsequent calls to UNION involving this set, then the root node will never be deleted (unless all of its children are removed). For resource-constrained problems, this behavior could be problematic – requiring an additional "garbage collection" procedure to ensure that all deleted nodes are immediately removed from memory. Since there are no child pointers, it is not immediately apparent how to implement such a policy. In any case, it is likely that garbage collection would be an expensive operation – limiting the utility of this implementation.

As suggested in the problem statement, deleting a node with children is only difficult because nodes do not have pointers to their children. If nodes did maintain child pointers, then when a node was deleted it could be replaced with one of its children. While this strategy would ensure that nodes are immediately deleted, it requires additional resources to be allocated for each node – namely, the set of child pointers. Consider the simple example shown in Figure 1(b). In this case, if node b only maintains a pointer to one child, then we could potentially cause an error. That is, if we delete node b and replace it with node d, then if we subsequently try to delete node d it will not report any children and, as a result, its deletion will cut off node e from the root. From this example, it is apparent that nodes must keep points to all of their immediate children. It is not apparent what data structure will work best in such a situation; it seems difficult to maintain a linked list of children for each node and not alter the running times of UNION, INSERT, or FIND-SET. As a result, one benefit of lazy deletion is it does not significantly alter the resources required for each node (i.e., it only adds the *children* and *marked* attributes).

In conclusion, we find that union-find with lazy deletion is relatively easy to both implement and analyze. In particular, it achieves the goal of adding deletion to the classic union-find data structure without modifying the running time of UNION, INSERT, or FIND-SET. For a more advanced treatment of the union-find with deletion data structure, see Kaplan *et al.* [3].

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill, 2001.

[2] Jeff Erickson. Union-find. `http://compgeom.cs.uiuc.edu/~jeffe/teaching/373/notes/09-unionfind.pdf`, 2003.

[3] H. Kaplan, N. Shafrir, and R. Tarjan. Union-find with deletions. In *Proceedings of 13th SODA*, pages 19–28, 2002.

[4] Claire Kenyon. Analysis of union-find tree data structure. `http://www.cs.brown.edu/courses/cs157/unionFind.ps`, 2006.