Computer Science CSC263H March 12, 2020
St. George Campus University of Toronto

Solutions for Homework Assignment #5

Answer to Question 1. Let σ be any sequence of k UNIONs followed by k' FINDs. Executing all the k UNIONs in σ results in a forest of rooted trees with a total of k edges; in the special case that $k = n - 1$, this is a single rooted tree. Since all the k' FINDs (with Path Compression) are executed ***after*** executing all the UNIONs, we have the following two properties:

- ***if a vertex is a root just before the execution of the FINDs, then it remains a root during the execution of all the FINDs.***
- ***if a vertex becomes a child of a root during the execution of one of the FINDs (because of Path Compression), then it remains a child of this root during all the subsequent FINDs.***

The above two properties are essential to proving that executing a sequence σ of k UNIONs followed by k' FINDs takes only $O(k + k')$. Note that these properties do ***not*** hold for arbitrary sequences of k UNIONs and k' FINDs, where UNIONs may be mixed with FINDs.

Running time analysis. We can do it in (at least) two ways. The first one is reminiscent of the way we analyzed the complexity of the Linked-List implementation of Disjoint Sets Union/Finds with Weighted Union. The second one is closer to the "accounting method" that we illustrated in the amortized analysis of binary counters and dynamic tables.

1. It is clear that the cost of doing all the k UNIONs is $O(k)$.

   We compute the cost of each FIND operation by using the following charging scheme. Charge 1 to each vertex v on the FIND path, except if v is the child of a root, in which case we charge 1 to the FIND operation itself. Note that:
   - Once a vertex is charged it becomes a child of a root, and so it cannot be charged again by any future FIND operation. Thus the maximum charge possible on any vertex is 1.
   - At most k vertices can be charged (because only a vertex "at the bottom" of an edge can be charged, and, as we noted above, only k edges exist at the end of the k UNIONs).

   So the total charge on all vertices is $O(k)$. Thus, in the worst-case the total cost for all k' FINDs is:

   Total charge on vertices + Total charge on FINDs = $O(k)$ + $O(k') = O(k + k')$.

   Thus the total cost of k UNIONs followed by k' FINDs is $O(k) + O(k + k') = O(k + k')$.

2. Let the "actual cost" of a UNION be 1, and the actual cost of a FIND(x) operation be the number of edges on the path from x to the root of the unique tree that x belongs to. These costs are up to constant factors equal to the running times of the respective operations.

   Let us now assign the amortized costs. In the following, we will use "coins" to represent units of cost. *Each UNION operation is charged 2 coins*: we use one coin to cover the actual the cost of doing the UNION operation, and we place the second coin as a *credit* attached to the vertex that becomes the new child of the root of the tree built by this UNION. *Each FIND operation is charged 1 coin.* Note that the total amortized cost of the sequence σ is $O(k + k')$. It is now sufficient to prove that we always have enough coins to cover the actual cost of executing all the operations in σ. It is clear that all UNION operations are paid for, so it remains to show that the cost of all FIND operations is also covered.

   After all k UNION operations are executed, and before any FINDs are executed, we have the property that every vertex, except the roots of the trees, have a coin attached to them. Throughout the execution of the FIND operations, we will maintain the invariant that every node has a coin attached to it, except possibly the roots and the children of the roots.

To show that the cost of all FIND operations is covered, note that the actual cost of a FIND(x) operation is equal to the number of nodes on the path from x to the root, including x itself, but excluding the root. To cover this cost, FIND(x) uses the coins attached to these nodes. If the node on the path that is a child of the root does not have a coin attached to it, then FIND uses the coin that is charged to the operation itself. This shows that, as long as we maintain the invariant that all nodes have coins attached to them, except possibly the roots and the children of roots, the FIND operation will be paid for. To show that the invariant is maintained, observe that, by Path Compression, after the FIND operation is executed, all nodes on the path to the root become children of the root, and will remain so thoughout the remaining FIND operations. Moreover, the root will also remain a root throughout the remaining FIND operations. Therefore, all nodes from which we removed coins become children of the root, and the invariant is maintained.

Answer to Question 2. In what follows $G = (V, E)$ is a finite, simple, undirected graph.

a. Adjacency list $\bar{L}$ requires n pointers to linked lists, one for each of $G.V$'s n vertices, just as adjacency list L does. Since undirected graph G has no more than $n(n-1)/2$ edges, the number of edges in $\bar{G}$ is $n(n-1)/2 - m$, so since G is assumed to be sparse:

$$\frac{n(n-1)}{2} \geq m \geq \frac{n(n-1)}{4} \geq \frac{n(n-1)}{2} - m \geq 0$$

Either adjacency list uses n linked lists with space proportional to their number of edges, so L requires $O(n+m)$ space, whereas $\bar{L}$ requires $O(n + (n(n-1)/2 - m))$ space, where the second term, $n(n-1)/2 - m$, is at most m and at least 0.

Degree(G, i): Return the degree of vertex $i \in G.V$

In either representation L or $\bar{L}$ we must count the elements of the relevant linked list, $L[i]$ or $\bar{L}[i]$ (in $\bar{L[i]}$ you must subtract the count from $n-1$ to find the degree). If vertex i has degree $n-1$ this has complexity $O(n)$ in list $L[i]$, but if vertex i has degree 0 this has complexity $O(n)$ in list $\bar{L}$. Thus worst-case complexity $O(n)$ in both cases.

AverageDegree(G): Return the average degree over all vertices in $G.V$

In L, count all $2m$ nodes of L's n linked lists, and then divide the count by n, for worst-case complexity $O(n+m)$. In $\bar{L}$, count all $2(n(n-1)/2 - m)$ nodes of $\bar{L}$'s n linked lists, then subtract the count from $n(n-1)$, and finally divide the difference by n, for worst-case time complexity $O(n + (n(n-1)/2 - m))$.

ContainsEdge(G, i, j): Given $i, j \in G.V$, return True if there is an edge $(i, j) \in G.E$, False otherwise

In L, traverse linked list $L[i]$ until either j is encountered (return true), or the end of the linked list is encountered (return false). In $\bar{L}$, traverse linked list $\bar{L}[i]$ until either j is encountered (return false), or the end of the linked list is encountered (return true). Hence worst-case complexity $O(n)$, since $L[i]$ or $\bar{L}[i]$ may have $n-1$ entries.

InsertEdge(G, i, j): Given $i, j \in G.V$ and edge $(i, j) \notin G.E$, add edge (i, j) to $G.V$

In L we insert j at the head of $L[i]$ and i at the head of $L[j]$, hence worst-case complexity $O(1)$. In $\bar{L}$ we must remove i from $\bar{L}[j]$ and remove j from $\bar{L}[i]$. Since these elements may be in arbitrary positions in $\bar{L}[i]$ and $\bar{L}[j]$, we have worst-case complexity $O(n)$.

b. space: Nodes in AVL trees are proportional in size to nodes in singly-linked lists (a constant number more pointers plus a balance factor), so worst-case space complexity is the same: $O(n+m)$.

Degree(G, i): This requires counting the nodes in AVL tree $L'[i]$, which has as many nodes as the degree of vertex i. We can traverse all the nodes of an AVL tree using, for example, pre-order traversal, in time proportional to the number of nodes. Thus worst-case time complexity $O(n)$.

AverageDegree(G): This requires counting the nodes in all n of the AVL trees of L', which is proportional to the number of edges plus accessing each tree, hence worst-case complexity $O(n+m)$.

ContainsEdge(G, i, j): This requires an AVL search of $L'[i]$, so worst-case complexity $O(\log n)$.

INSERTEDGE(G, i, j): This requires two AVL insertions, one at $L'[i]$ and the other at $L'[j]$, hence $O(\log n)$ worst-case complexity.

Answer to Question 3.

a. Below is a modified BFS search for **target** starting at **start** that assumes that dictionary D allows worst-case time complexity of $O(\log n)$ to look up each of the k-character strings, and $O(n)$ time complexity to traverse all entries in D. I annotate each dictionary entry with fields to record COLOUR and PARENT, which are needed by breadth-first search. For brevity, I omit distance, although it can be added if I wanted to keep track of how short a sequence between **start** and **target** is.

```
   BFS-WORD-SEARCH($G, D, k, start, target$)
1  for each string $t \in D, t \neq start$
2      $t$.COLOUR ← WHITE
3      $t$.PARENT ← NIL
4  $start$.COLOUR ← GRAY
5  $start$.PARENT ← NIL
6  Q ← $\emptyset$
7  ENQUEUE(Q, start)
8  while $Q \neq \emptyset$
9      $t$ ← DEQUEUE(Q)
10     if $t = target$ return $t$
11     for each $i \in \{1, 2, ..., k\}$
12         for each $\alpha \in \{a, b, ..., z\}$ where $\alpha$ is not $i$th character of $t$
13             $t' = t$ with $i$th character replaced by $\alpha$
14             if SEARCH$D, t) \neq$ NIL and $t'$.COLOUR = WHITE
15                 $t'$.COLOUR ← GRAY
16                 $t'$.PARENT ← t
17                 ENQUEUE(Q, $t'$)
18     $t$.COLOUR ← BLACK
19 return no path exists!
```

If reference t to the **target** string object is returned, we use its parent field to print a sequence from **start** to **target**, at worst-case time complexity $O(n)$, the size of D:

```
   Reverse-Print($t$)
1  if $t$.PARENT $\neq$ NIL
2      REVERSE-PRINT($t$.PARENT)
3  print($t$)
```

b. Since D has n entries, the loop at lines 1–3 of BFS-WORD-SEARCH$(G, D, k, start, target)$ executes $O(n)$ times, with a cost $O(k)$ to determine the string is not **start**. Lines 4–7 contribute time complexity $O(1)$. Lines 9 and 17 enqueue/dequeue up to n length-k strings, and for each string the nested loop at lines 11 and 12 executes lines other than line 17 $25k$ times, at a cost $O(\log n)$ per iteration, since dictionary D must be checked each time.

Altogether this yields worst-case time complexity $O(n+25kn \log n)$, or $O(25kn \log n)$. The extra space required is $O(nk)$ for enqueuing strings.