COT 6405 Analysis of Algorithms
Homework 4

Michael Novo

November 21, 2015

Note: The following problems were discussed with Nestor Hernandez, Georges Kamhoua, and Gregory Reis.

1. *Two new operations of Fibonacci heap*

   a. *FIB-HEAP-CHANGE-KEY*

   Let us consider the FIB-HEAP-CHANGE-KEY(H, x, k) operation when k is equal to x. This is the easiest case, because we do not have to alter the structure of the tree. The resulting amortized time is $O(1)$.

   When k is less than x, we can use the FIB-HEAP-DECREASE-KEY(H, x, k) we have already seen. Once again, we see that the amortized analysis gives us $O(1)$.

   Perhaps, the trickiest part in implementing FIB-HEAP-CHANGE-KEY is the key change when k is larger than x. In keeping with the strategy of "laziness" that the Fibonacci Heap aims to follow, the implementation would be similar to that of DECREASE-KEY. That is, use CUT to move the x to the root list. Perform the CASCADING-CUT operation if this is the second child that the parent of x loses. The resulting amortized time analysis, is equivalent to that of DECREASE-KEY or $O(1)$.

   The change in potential is:
   $((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2m(H)) = O(1)$

   b. *FIB-HEAP-PRUNE*

   One's initial thought when considering the implementation of the PRUNE operation is to solely use the DELETE operation already implemented by the Fibonacci Heap. However, the problem is that the DELETE operation relies on the expensive CONSOLIDATE operation. So then, our approach should attempt to stay away from the DELETE operation already implemented.

   Instead, we prune by first removing those nodes on the root list that are without children. We can see that this is a constant time actual cost operation because it simply involves removing a node in a doubly-linked list. The next strategy is to delete leaves of the Fibonacci Heap. Just as we have maintained a root list, we maintain a leaf list. When removing a leaf node, we cut in a similar fashion as when we perform the DECREASE-KEY operation. That is, we move it to the root list and mark its parent. Once in the root list, it may be quickly deleted. In order to take care of the problem of when the parent of x has lost multiple children, we perform a CASCADING-CUT. Yet, this too we can exploit since we can use the nodes that are newly moved to the root list as the next target for deletion. The amortized cost for performing this operation is similar to that of the DECREASE-KEY operation. We have previously found that the DECREASE-KEY operation runs in $O(1)$.

   Once again, we have the following change in potential:
   $((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2m(H)) = O(1)$

2. *MST algorithms*

   a. *Kruskal's algorithm with union-by-rank and path compression*

   We saw in class that when using union-by-rank and path-compression, we get $O((m+n) \log^* n)$, where $\log^* n$ is a very slowly growing function, and for practical purposes is a constant.

CLRS uses a more complicated analysis using the inverse Ackermann function. Kruskal's algorithm performs m FIND-SET and UNION operations in its second for loop. It also initially performs n MAKE-SET operations. Using the inverse Ackerman function, α, we arrive at $O((n+m)\alpha(n))$. Assuming that G is connected, we can obtain $O((m)\alpha(n))$. Then, we see $\alpha(|n|) = O(\log n) = O(\log m)$, leaving us with Kruskal's running time at $O(m \log m)$. But, we know that $m < n^2$, so $O(\log m) = O(\log n)$ and we may write the running time as $O(m \log n)$.

b. *Prim's algorithm with Fibonacci heap-based priority queues*

We can write the running time of Prim's algorithm like that of Kruskal's, as $O(m \log n)$ with a binary min-heap. But, if we choose to speed up by using a Fibonacci heap, we can arrive at the faster $O(m + n \log n)$. The speed-up is the result of the Fibonacci heap's superior amortized performance, $O(1)$ when performing the DECREASE-KEY operation.

3. *Union-by-rank is $O(m \log n)$*

In using union-by-rank, the rank of a node only increases if it is a root node. Additionally, the increase in rank is only made when two subtrees of equal size are merged. In order to prove that every node in the tree-based implementation of UNION-FIND has rank of at most $\log n$, we use induction first to show that if a root node has rank r, then it has at least 2^r nodes in its tree. For the base case, we have that $r = 0$, and the rule holds.

As mentioned above, the rank of a node only increases when two trees of equal rank are joined. Because of this, our inductive hypothesis is that the property is true for 2^{r-1}. Then, we obtain a node of rank r with 2^r nodes in its subtree by using our inductive hypothesis and merging two trees of rank $r-1$. Therefore, we have proven that in a tree with node of rank r, the size of the tree is at least 2^r.

Since we now know that $n \geq 2^r$, we can easily see that the highest rank of a node in a tree of size n is $\log n$. We have seen in class that UNION takes $O(1)$ amortized time. In addition, we can now see that we can perform the FIND operation in $O(\log n)$, rank is an upper bound on the height of a node (CLRS 21.3). Therefore, a sequence of m operations can be performed in $O(m \log n)$.

4. *Tarjan's off-line least common ancestors problem*

a. The LCA procedure works by visiting nodes starting at the root and proceeding to the leaves in a recursive fashion. The procedure finishes at the leaves first, and proceeds to finish at each depth above it until reaching the root. Line 10 only occurs if the color of both $\{u, v\}$ are BLACK. This would imply that both nodes have been visited. When u is visited before v, Line 10 will not execute. However, when v is later visited, Line 10 will then execute. Since there are only two opportunities for executing Line 10 for a given vertex pair, and on one opportunity Line 10 is not executed, Line 10 is only executed on when the second vertex is visited. Thus, line 10 executes only once for each pair $\{u, v\} \in P$

b. The LCA procedure calls itself for each child of v of the vertex u on which it is called. Although the procedure does perform the UNION operation on Line 5, it only does so after the recursion has finished with nodes at larger depth. Therefore, as the recursion unfolds and reaches the leaves from the root, each node is in its own set. Thus, the number of sets in the disjoint-set data structure at the time of the call is equal to the depth of u, or equivalently, one set for all nodes above u until the root is reached.

c. *Proof of Correctness*

First, we realize that the algorithm will execute the print statement if and only if the colors of both u and v are BLACK. This corresponds to when LCA has been finished for one node and is past line 7 for the other node.

Assume for the sake of contradiction that LCA does not print out the least common ancestor of u and v for each pair $\{u, v\} \in P$. As stated above, Line 10 will only execute when both u and v are BLACK. When the FIND-SET(v).*ancestor* is called the FIND-SET operation will stop when it finds a node z that is its own parent. This only occurs if the LCA procedure has not finished for z. The only nodes that have not finished LCA are those nodes that are at lower depths than both u and v. Additionally, the first node that is its own parent is that with the greatest depth with respect to the root, but at a smaller depth when compared to u and v. This node, z , has is also its own ancestor. So, when it is found by FIND-SET(v), its *ancestor* attribute will be set to itself. This corresponds to the node that has smaller depth than both u and v and is an ancestor. This is a contradiction of our original claim. Thus, LCA correctly prints the least common ancestor of u and v for each pair $\{u, v\} \in P$.

d. *LCA running time*

Using the data structures discussed in 21.3 along with union by rank with path compression, the worst-case running time is $O(m\alpha(n))$ for a sequence of m UNION and FIND-SET operations. Since we have n vertices and each vertex performs a search and union, the entire algorithm runs in $O((m+n)\alpha(n))$. This is assuming we have an additional data structure that allows us to find $\{u, v\}$ pairs when presented with u.

Note: Tarjan's *Applications of Path Compression on Balanced Trees, 1979* was consulted for this problem.

5. *Independent Set Search*

The independent set problem finds a subset k of edges. Because of this, the worst, yet sufficient search we can perform is in $O(n)$ by simply trying all subset sizes from 1 to n. For a more efficient search, we can perform a binary search on the sorted sequence from 1 to n in $O(\log n)$. Each time we try to see if M_{IS} returns true with a particular k. If it does, move to the right half of the sorted list. If it returns false, move to the left half of the sorted list. Continue this procedure until finding the maximum k that is accepted by M_{IS}.