CS 473: Fundamental Algorithms, Spring 2013

HW 0 solution

1. (50 PTS.) The multi-trillion dollar game.

You are given k piles having in total n identical coins (each coin is a trillion dollar coin, of course, see picture on the right). In every step of the game, you take a coin from a pile P, and you can put it in any pile Q, if Q has at least 2 less coins than P. In particular, if P has two coins, you are allowed to take one of the coins of P and create a new pile. If a pile has only a single coin, you are allowed to take the coin and buy whatever banana republic you want with it.

(A) (25 PTS.) Prove, formally, that this game always terminates after a finite number of steps.

Solution:

The key configuration is that we can never create a pile larger than a currently existing pile. A configuration of n coins is $T(x, y)$, if the largest pile in this configuration has x coins in it, and there are exactly y such piles.

Claim 0.1. *The game of coins described above always terminates.*

Proof: **Base case:** If the configuration is $T(1, n)$ then the game terminates in n steps, since at every step, we remove a coin that belongs to a pile of height 1.
Inductive hypothesis: Assume that we proved the claim for any configuration $T(x, y)$, such that $x < X$, or $x = X$ and $y \leq Y$. We now need to prove that the game terminates for any configuration $T(X, Y + 1)$.

$T(X, Y + 1)$: So consider such a configuration C, and partition its coins two sets:

(I) C_1 would be all the coins in the piles of height X, and
(II) C_2 would be all the other coins.

By induction, we know that any game that involves only the coins of C_2 must be finite, since C_2 is a configuration $T(x', y')$, where $x' < X$. As such, sooner or later, the coins in C_2 are exhausted and the game must take one of the coins in C_1. After this move, the resulting configuration is either

(i) $T(X, Y)$ (assuming $Y > 0$), or
(ii) $T(X', Z)$ where $X' < X$ (if $Y = 0$).

In either case, by induction, any game starting from this new configuration terminates after a finite number of steps. As such, any game starting with the configuration $T(X, Y + 1)$ terminates after a finite number of steps. ■

The beauty of this proof is that it works verbatim for (B). Nevertheless, here is a significantly simpler proof using other ideas.

Proof: Let c_i denote the height of the ith pile in the given configuration C, where i ranges from 1 to the total number of piles in C. We define the following potential function:

$$f(C) = \sum_i c_i^2.$$

Let C_t denote the configuration at time t. We claim that this function strictly decreases at every move. Indeed, consider the case where at time step $t+1$ we move a coin from pile i to pile j. Note that $c_j \leq c_i - 2$ (c_j might be zero if we start a new pile). Then

$$f(C_{t+1}) - f(C_t) = (c_i - 1)^2 - {c_i}^2 + (c_j + 1)^2 - {c_j}^2 = 2c_j - 2c_i + 2 \leq -2.$$

Now, $f(C_t)$ is always a non-negative integer number. Every move decreases its value by at least 1 (if we just take away a coin!). As such, after $f(C_0)$ bounds the number of steps in the game till it terminates, where C_0 is the initial configuration. ■

(B) (25 PTS.) Unfortunately, the world economy is suffering from hyper-inflation. A trillion dollar bucks do not go as far as they used to go. To keep you interested in the game, the rules have changed – every time you remove a coin from a pile, you are required to insert two coins into two different piles (again, these new piles need to have 2 less coins than the original pile) [you get the extra coin for such a move from the IMF]. Prove, formally, that this game always terminates after a finite number of steps.

Solution:

The first proof of part (A) holds verbatim here. Here is another alternative proof:

Proof: For a configuration C, let $g(C) = \sum_i (5^{c_i} - 1)$. Similarly, assume that at time step $t+1$ we move a coin from pile i and insert coins to piles j and k. Let c_i, c_j, c_k the heights of those piles at time step t (they can be zero if start new piles). We have that

$$\begin{aligned} g(C_{t+1}) - g(C_t) &= 5^{c_i - 1} + 5^{c_j + 1} + 5^{c_k + 1} - 5^{c_i} - 5^{c_j} - 5^{c_k} \\ &= 4(5^{c_j} + 5^{c_k} - 5^{c_i - 1}) = 4\big(5^{c_j} + 5^{c_k} - 5 \cdot 5^{c_i - 2}\big) \\ &\leq 4 \cdot (-3) \cdot 5^{c_i - 2} \leq -12, \end{aligned}$$

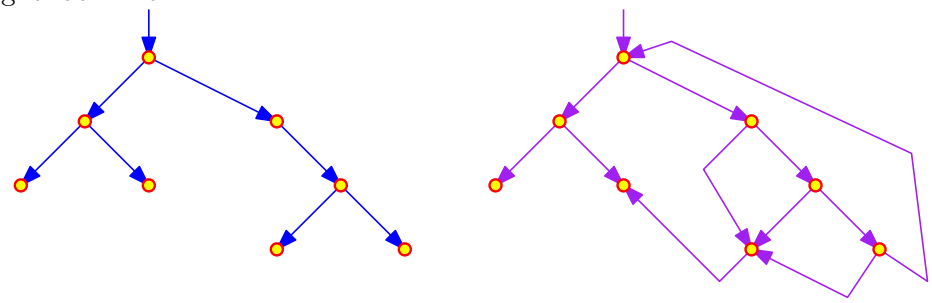
since $c_i \geq c_j + 2$ and $c_i \geq c_k + 2$. That is $g(C_{t+1}) \leq g(C_t) - 12$ in this case. If we take a coin from a pile with a single coin, then $g(C_{t+1}) = g(C_t) - 4$ (since a pile of size 1 contributes 4 to $g(C_t)$).

The function $g(C_t)$ is a non-negative integer number, it is strictly decreasing at each step, and such $g(C_0)$ bounds the number of steps in any game starting with the configuration C_0. ■

2. (50 PTS.) The baobab tree.

You are given a pointer to the root of a binary tree. Normally, each node in the tree has a pointer to its left child, right child, and its parent (these pointers are `NULL` if they have nothing to point to). Unfortunately, your tree might have been corrupted by a bug in somebody else

code[1], so that some of the pointers that should contain NULL, now point to some nodes in the tree. The following figure demonstrates how a good tree might look like, and how a corrupted tree might look like.



(A) Good tree. (B) A corrupted tree.

Describe an algorithm[2] that determines whether the tree is corrupted or not. Your algorithm must not modify the tree. For full credit, your algorithm should run in $O(n)$ time, where n is the number of nodes in the tree, and use $O(1)$ extra space (not counting the tree itself).

Solution:

We will describe an algorithm that traverses the tree only once starting from the root, and therefore runs in linear time. If a node is visited a second time, then the algorithm will output that the tree is corrupted (NO TREE). Otherwise, it will output that the tree is valid (GOOD TREE). The algorithm traverses the tree via its edges (pointers). First we check if given pointer to a node v, then is this node valid. In the following, root is the given pointer to the root of the tree. Note, that this is a constant time operation.

```
IsValid(v)
    if v = NULL then return TRUE
    if parent(v) = NULL and v ≠ root then return FALSE
    if parent(v) ≠ NULL and v = root then return FALSE
    if left(v) ≠ NULL and left(v) = right(v) then return FALSE
    if left(v) ≠ NULL and left(v) = parent(v) then return FALSE
    if right(v) ≠ NULL and right(v) = parent(v) then return FALSE
    if (left(v) = v) or (right(v) = v) or (parent(v) = v) then
        return FALSE
    if parent(v) ≠ NULL then
        if left(parent(v)) ≠ v and right(parent(v)) ≠ v then
            return FALSE
    if left(v) ≠ NULL then
        if parent(left(v)) ≠ v then
            return FALSE
    if right(v) ≠ NULL then
        if parent(right(v)) ≠ v then
            return FALSE
    return TRUE
```

[1] After all, *your* code is always completely 100% bug-free. Isn't that right, Mr. Gates?

[2] Since you've read the Homework Instructions, so you know what the phrase 'describe an algorithm' means. Right?

Since, in a valid tree, between every two nodes u, v where u is the parent of v there will be one edge (pointer) pointing from u to v and another from v to u, this implies that if we think about these pointers as edges, then this graph is Eulerian. As such, it has an Eulerian path. Namely, we can view a valid tree as a linked list (where every edge tree corresponds to two directed edges). Given such an edge, lets verify that it is valid:

```
IsValidEdge(u, v)
    if u = NULL or v = NULL then return FALSE
    if u = v then return FALSE
    if left(u) = v or right(u) = v then
        if parent(v) ≠ v then
            return FALSE
        else
            return TRUE
    if parent(u) ≠ v then return FALSE
    // must be parent(u) = v
    if left(v) ≠ parent(u) and right(v) ≠ parent(u) then
            return FALSE
        else
            return TRUE
```

Now, we define a "next" operator on the directed edges of the tree. That is, given a directed edge $u \to v$ in the tree, we compute the next edge:

```
next(u, v)
    if v = left(u) or v = right(u) then
        if left(v) ≠ NULL then
            return (v, left(v))
        if right(v) ≠ NULL then
            return (v, right(v))
        return (v, u)
    // must be that v = parent(u)
    if left(v) = u then
        if right(v) ≠ NULL then
            return (v, right(v))
        else
            return (v, parent(v))
    // must be that v = parent(u) and right(v) = u
    return (v, parent(v))
```

Our algorithm is now going to scan this "virtual" linked list, and check that every node along it is valid.

```
VerifyFromEdge( u, v )
    while v ≠ NULL do
        if not IsValidEdge(u, v) then return FALSE
        if not IsValid(u) then return FALSE
        if not IsValid(v) then return FALSE
        (u, v) ← next(u, v)
    return TRUE
```

We are now ready for the main function:

```
IsValidTree( root )
    if not IsValid(root) then
        return FALSE
    if left(root) ≠ NULL then
        return VerifyFromEdge(root, left(root))
    if right(root) ≠ NULL then
        return VerifyFromEdge(root, right(root))
    return TRUE // Tree is empty.  Boring.
```

Correctness and Running time: First, the algorithm clearly uses a constant amount of space. Now, if the traversal visits a node x twice, because two nodes think that x is their child, then the algorithm immediately stops and outputs that the tree is not valid, because one of these directed edges would fail the validity test, because the parent pointer of x is invalid as far a the second visit. This implies that we can not visit a node more than three times overall, which implies immediately a linear running time, and that the algorithm always terminates.
Similarly, the validity check per node, enforces that the only node with a `NULL`parent pointer is the root. Furthermore, all the parent pointers in the visited nodes are valid, and every node that has a pointer into a node, there is matching pointer in the other direction. Namely, if the algorithm returns that the tree is valid, then all the pointers in the scanned structure are valid. That is, the tree is valid.
If the algorithm claims that a tree is not valid, then it can points to what is invalid in the tree. As such, this algorithm is correct, works in linear time, and uses only $O(1)$ additional space.