CS 561, HW5

Stephen Harding

November 5, 2012

1. Solve Problem 5 on the midterm (Drunken Debs):

   (a) Note that $f(1) = 1$. What is $f(2)$?
   $f(2) = \frac{1}{2}$

   (b) Write a recurrence relation for $f(n)$.
   To solve this, note that if debutant 1 selects porsche 1, $f(n) = 1$. If she instead picks porsche n, $f(n) = 0$. If she picks porsche i, then $f(n) = f(n-i)$. This is because debutants 2 through $i-1$ will find their own porsches and we essentially start over with debutant i (as if she was debutant 1) and the remaining debutants and porsches. Thus, to find $f(n)$ we find the average of all the probabilities $f(1)$ through $f(n-1)$. Thus:

$$f(n) = \frac{1}{n} \cdot \sum_{i=1}^{n-1} f(n-i) = \frac{1}{n} \cdot \sum_{i=1}^{n-1} f(i)$$

   (c) Now use the guess and check method to solve for the exact value of $f(n)$ for $n \geq 2$:
   Guess: $f(n) = \frac{1}{2} \quad \forall n > 1$
   Proof (by substitution method):
   Base case: $f(2) = \frac{1}{2}$ (because there are exactly two possible outcomes and in only one debutant 2 finds her own porsche) $\surd$
   Inductive Hypothesis: Assume that $f(n) = \frac{1}{2} \quad \forall_{(1<m<n)} \quad and\ f(1) = 1$
   Inductive Step: Show that the Inductive Hypothesis holds for n as well:

$$f(n) = \frac{1}{n} \cdot \sum_{i=1}^{n-1} f(i) = \frac{1}{n} \cdot \underbrace{\left(1 + \sum_{i=2}^{n-1} \frac{1}{2}\right)}_{\text{by IH}} = \frac{1}{n} \cdot \left(1 + \frac{n-2}{2}\right) = \frac{1}{n} + \frac{1}{2} - \frac{1}{n} = \frac{1}{2} \qquad \square$$

2. Solve Problem 2 on the midterm (Amortized Analysis with counter):

   (a) What is the worst-case run time of INCREMENT as a function of n?
   $\theta(\log n)$

   (b) What is the worst-case run time of RESET as a function of n?
   $\theta(\log n)$

   (c) Prove that in an arbitrary sequence of calls to INCREMENT and RESET, each operation has an amortized cost of $O(1)$.
   Accounting method:
   We collect \$3 in taxes for every call to INCREMENT . We immediately spend \$1 for the bit that is flipped from 0 to 1 and store the remaining \$2 on the bit for future use. (Note:

there is exactly one bit that is flipped from 0 to 1 for each call to INCREMENT.) Every time we flip a 1 to 0, we use one of the remaining dollars. Note that all bits $\leq m$ have at least \$1 on them. (With several calls to increment bits will tend to accumulate more than \$1.) When reset is called, we use one of the remaining dollars on each bit to pay the cost of the RESET operation. To account for the edge case where RESET is called prior to INCREMENT (or immediately following a call to RESET), we also tax \$1 for each call to RESET which will be immediately used to pay for the function overhead.

Hence, for n operations, the amortized cost is at most $3n$ and so the amortized cost for each operation is $O(1)$ $\square$

3. Problem 17-2 (Making Binary Search Dynamic):

   (a) To perform the SEARCH operation we must do a binary search on each of the k sorted arrays. Thus the worst-case running time is: (note: $k = \lceil \log(n+1) \rceil$)

$$\sum_{i=0}^{k} \log 2^i = \sum_{i=0}^{k} i = \frac{k(k+1)}{2} = \Theta(k^2) = \Theta(\log^2 n)$$

   (b) *Note: in this data structure, all lists A_i are either empty or full.* To perform the INSERT operation, we first examine the A_0 list. If it is empty, we simply insert the element in that list and we are done. Otherwise we use the merge routine of merge-sort to sort the contents of A_0 and our element (resulting in a list of length 2). If the list A_1 is empty we replace it with our sorted list. Otherwise, we merge our sorted list with with A_1 and look at A_2. If it is empty we replace it with our sorted list. Otherwise we merge it with our list and examine the next list. We continue this process until we find an empty list which we will replace with our list.

   Worst case running time: *Note: the running time of the merge operation is $O(n)$.* The worst case running time for the INSERT operation happens when all lists A_i are full. Since there are $\lceil \log(n+1) \rceil$ such lists and there are 2^i elements in each list, the running time is:

$$\sum_{i=0}^{\lceil \log(n+1) \rceil} 2^i = 2^{\lceil \log(n+1) \rceil + 1} - 1 \leq 2^{\log(n+1)+2} \leq 4 \cdot 2^{\log(n+1)} = 4 \cdot (n+1) = 4n + 4 = \Theta(n)$$

   Amortized running time: The amortized cost will be much better than the worst case because most of the time, not all of the arrays will be full. A_0 will be empty $\frac{1}{2}$ of the time, A_1 will be empty $\frac{1}{4}$ of the time, and so on. As we saw in the aggregate analysis of the binary counter, there are at most $\lfloor n/2^i \rfloor$ times that the A_i^{th} list will go from empty to full for n INSERT operations. Further, note that every time INSERT is called *exactly one* array is going to go from empty to full. This will be the first empty array that we encounter.

   Let A_z be the first empty array. The cost of one INSERT operation is 2^z (because $\forall i < z$, all A_i lists are full). With this fact, we can compute the cost of n INSERT operations using the aggregate method:

$$T(n) = \sum_{z=1}^{\lceil \log(n+1) \rceil} \left\lfloor \frac{n}{2^z} \right\rfloor \cdot 2^z \leq \sum_{z=1}^{\lceil \log(n+1) \rceil} n \;=\; O(n \log n)$$

   Hence, the amortized running time for n operations is $\frac{O(n \log n)}{n} = O(\log n)$ $\square$

(c) To perform the DELETE operation we keep track of the first non-empty array. When DELETE is called, we find the element to be deleted using the SEARCH operation and replace it with the first element of the first non-empty array identified above. The array must be sorted again. Now we must handle the non-empty array from which we took an element. Since, by definition, all arrays before it are empty, we can simply take the first remaining element and put it in A_0, the next two elements into A_1, the next four into A_2, and so on. Since the array was sorted to begin with, the sub arrays will also be already sorted. □

4. **Problem 22-4 (Reachability)**

We maintain an array of length $|V|$ called *mins* initialized to ∞ and another array called *visited* which we initialize to false. We start with any $v \in V$ and recursively find $min(v)$ with the following algorithm:

Algorithm 1 Find the node v such that v's label is the minimum of the nodes reachable from u

```
function MIN(u)
    if mins[u.label] ≠ ∞ then
        return mins[u.label]
    end if
    if u.label = 1 then
        mins[u.label] ← 1
        return mins[u.label]
    end if
    visited[u.label] ← TRUE
    minVal ← u.label
    for i ∈ children of u do
        if mins[i.label] = ∞ ∧ ¬visited[i.label] then  ▷ The ¬ visitied check is to prevent cycles.
            mins[i.label] ← MIN(i)
        end if
        if minVal > mins[i.label] then
            minVal ← mins[i.label]
        end if
        if minVal > i.label then                ▷ case where i was visited but has a smaller label
            minVal ← i.label
        end if
    end for
    mins[u.label] ← minVal
    return mins[u.label]
end function
```

After the run we look at the *mins* array. If all values are filled, we are done. Otherwise, we call MIN on the first element that is not filled in.

This algorithm visits each node and edge at most once (due to our memoization) and thus the running time is $O(V + E)$ □

5. **Professor Curly conjectures that if we do union by rank, *without path compression*, the amortized cost of all operations is $o(\log n)$. Prove him wrong by showing**

that if we do union by rank without path compression, there can be m MAKE-SET, UNION and FINDSET operations, n of which are MAKESET operations, where the total cost of all operations is $\theta(m \log n)$.

For this counterexample we will consider m to be asymptotically larger than n.

First, we consider n MAKESET operations which takes $\Theta(n)$ time.

Then we make calls to UNION in a clever way to construct a binary tree. First we make $n/2$ calls to UNION to make $n/2$ sets with a root node and once leaf (height 1). Then we make $n/4$ calls to UNION to make $n/4$ trees of height 2. We continue in this way until we have one tree of height $\log n$.

Note: the work so far is $\sum_{i=0}^{\log n} \frac{n}{2^i} = 2n - 1 = \Theta(n)$

Finally, we make the remainder of our calls to *FindSet* (i.e. $m - 2n + 1$ calls). Since the running time of FINDSET is $\Theta(\log n)$, the total cost is:

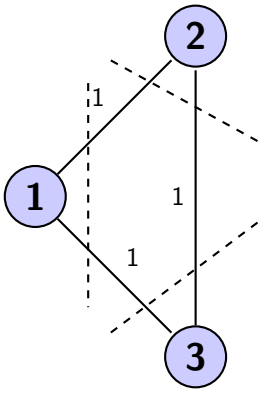
$\Theta(n) + \Theta((m - 2n + 1) \log n) = \Theta(n) + \Theta(m \log n - 2n \log n + \log n) = \Theta(m \log n)$ □

6. **Assume you are given a connected graph G. Give an algorithm that returns a vertex v in G, such that if v is removed, G is still connected.**

   Algorithm: Find the minimum spanning tree using Kruskal's or Prim's algorithm. Perform depth first search on the resulting tree to find a leaf node. Return that node. This algorithm is correct because removing a leaf from a tree does not affect the connectedness of the tree (or the original graph). □

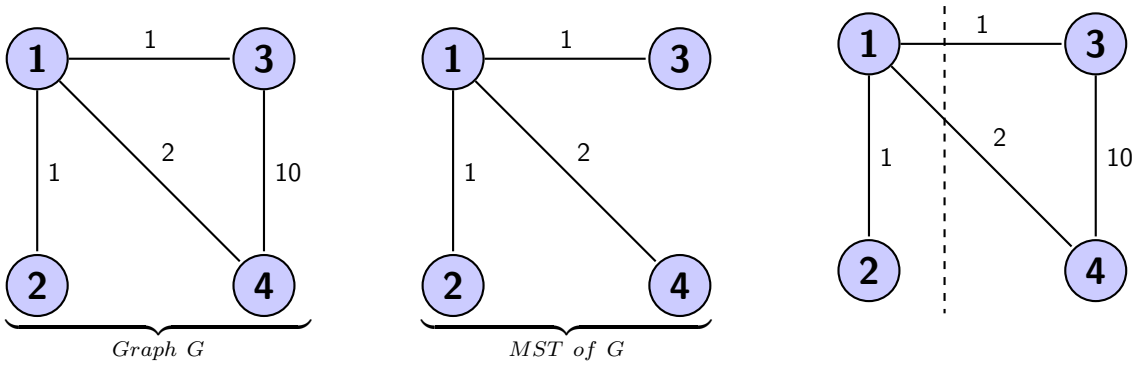
7. **Professor Moe conjectures that for any graph G, the set of edges {(u,v) : there exists a cut (S,V-S) such that (u,v) is a light edge crossing (S, V-S)} always forms a minimum spanning tree. Given a simple example of a connected graph that proves him wrong.**

   Consider the following connected graph (and the cuts as depicted):



   Each edge in the graph satisfies the condition because for each edge there exists a cut such that the edge is a light edge in a cut. However, this is not a minimum spanning tree because (among other reasons) this is not a tree! □

8. **Exercise 23.1-2 ("Professor Sabatier conjectures")**



The conjecture essentially states that if edge (u,v) is a safe edge (i.e. $A \cup \{((u, v)\}$ is a subset of a minimum spanning tree) then it is a light edge of a cut that respects A. Consider the above graph and its minimum spanning tree. The third graph depicts a a cut that respects A, where $A = \{(1, 2)\}$. Clearly, A is a subset of the MST and the light edge of the cut is (1, 3). From the theorem, the edge (1, 3) is safe for A. Note that the edge (1, 4) is also safe for A, but it is not a light edge for the cut. Thus, this is a counterexample to Prof. Sabatier's conjecture. □

9. **Exercise 23.1-3** If an edge (u, v) is in a minimum spanning tree, then it is a light edge in some cut of the graph.

   To show this, note that if we were to remove the edge (u, v) from the tree, we are left with two distinct connected graphs. Let one of the graphs be denoted S and the other $V - S$. Now we consider the cut $(S, V - S)$ of the original graph. The edge (u, v) *must* be a light edge of the cut, otherwise we could select a 'lighter' edge and replace it with (u, v) in our minimum spanning tree. This new minimum spanning tree would have a total weight strictly less than the original minimum spanning tree. This is a contradiction! Therefore, (u, v) *is* a light edge of some cut of the graph. □

10. **Exercise 22.2-6 / 22.2-7 ("There are two types of professional wrestlers")**

    Given a list of n wrestlers and a list of r wrestler pair rivalries we are asked for an algorithm that determines if we can designate some of the wrestlers as 'good guys' and the rest as 'bad guys' such that for every rivalry, the rivalry is between a 'good guy' and a 'bad guy'. If it is possible to partition the n wrestlers in this way the algorithm should return that partitioning. The algorithm must run in $O(n + r)$ time.

    First we will construct a graph using the list of n wrestlers as nodes and the list of pair rivalries will be the edges in the graph. If the list of rivalries is empty, we will say that all the nodes are 'good guys'. Otherwise, each node with zero edges will be designated as a 'good guy' and for each remaining connected component we will generate a breadth first tree using the algorithm described in §22.2 of our text where G is the connected component and s is a node in the connected component chosen at random.

    Once all the trees are constructed we designate each root node as a 'good guy' and each node u if $u.d$ is even, we will designate it as a 'good guy' and if $u.d$ is odd, we will designate it as a 'bad guy'. We will maintain two lists (one for 'good guys' and one for 'bad guys') and every time a node is designated, it will be added to the appropriate list.

    Finally, we examine each edge in the original graph and verify that it connects a 'good guy' node to a 'bad guy' node (i.e. it connects nodes (u, v) where u.d is even if v.d is odd and vise versa). If we find an edge that fails this test, we return FAIL to indicate that no such

partitioning exists. *Note: this will always occur if there is an odd length cycle in the original graph.* If each node passes this test, we return the two lists that represent the partition.

Running time: BFS runs in $O(V + E)$ time and so the sum of each BFS is $O(n + r)$. It takes $O(n)$ time to perform the designations and $O(r)$ time to verify each rivalry. Thus the total runtime is: $O(n + r) + O(n) + O(r) = O(n + r)$ $\square$