Homework 3

Frederick Robinson

28 January 2010

1 Problem 4.18

1.1 Algorithm

Consider the following algorithm for finding the shortest path from the start node to the end node.

Associate with each node two flags: Done/Not Done and Ready/Not Ready. Also mark each node with a time and a counter. When the algorithm starts each flag should be Not Done and Not Ready each time should be infinity, and each counter should be at 0. The only exception being the start node which should be marked Not Done, Ready and 0.

1. While there exists a node marked "Not done" and "Ready" call this node n
   (a) For each node n' connected to n
      i. Check the time it takes to get to n' from n starting at the time marked on n. If this time is less than the time marked on n' update n''s time with this time and create a pointer from n' back to n (replacing the current pointer if there is one)
      ii. Iterate the counter on n' and if doing so makes the counter equal to the number of edges coming in to n' mark n' Ready
      iii. If the node just marked ready is the end node, terminate returning the path generated by following the pointers backwards
      iv. Mark n done
2. If there are no nodes which are "Not Done" and "Ready" then find the node which has the smallest time, and is marked "Not Done." Mark this node "Ready" and go back to the beginning of the algorithm. (if the node marked ready is the end node, instead terminate the algorithm, returning the path generated by following the pointers backwards.)

1.2 Correctness

The algorithm will always terminate with a valid path to the end since it does not terminate unless it has found the end node. Also, in each loop it marks a

new node “Ready” node as ”Done”. If there are no such “Ready” nodes then Step 2 is executed to create a new one. Step 2 will always mark a node as “Ready” if there are any nodes left in the portion of the graph connected to the start since each node that has been visited is marked with a time.

Thus we have established that the algorithm must visit every node and therefore return a path from the start to end after it has been run (if such exists). It remains to show that such a path is the optimal one.

If the algorithm never needs to leave the first “for each” loop, it is fairly clear that it will return the optimal solution. It does not start checking paths out of a given node until each path into that node has been checked. In this way it insures that it has the best start time from a given node before checking further along the path.

Since each node before a given node must have been checked in this way it must be that the current node is marked with the minimal time it takes to get there. Moreover since the algorithm does not terminate until the end node is marked “Ready” (i.e., all paths into this node have been checked) it cannot be that there exists some better path into this node when the algorithm terminates.

The only potential for mistake is introduced when there are no natural “Ready” nodes and the algorithm marks the one with the least distance from the start as “Ready.” This however is correct as well since were there a shorter path to the node that the algorithm marks ready it could not be through any of the nodes marked “Done” (or it would be marked on the node already) Also, it can’t be through any of the nodes which are not marked as done, since each of these has a time greater than the node we chose to mark “Ready”

Thus, when we mark the node as “Ready” in this manner we really have already established the shortest path to it.

So, our algorithm correctly compute the minimal time from the first node to the last, and with this the optimal path since it maintains pointers along the best route.

1.3 Runtime

Step 1a is done once for every edge in the graph since after it has been performed on a given edge the node at the beginning of this edge is marked “Done” and not used afterwards.

Each time that step 2 needs to be executed it takes in the worst case time which is linear in the number of edges that is $(O(n))$. In the worst case there are no ready nodes after each execution of step 1a and 2 must be run once after each node. Thus, all together the algorithm is $O(n^2 + m)$.

2 Problem 5.2

2.1 Algorithm

The algorithm for this is very similar to the one used in the book to count inversions.

It breaks the set into successively smaller parts. Then, recombines those parts in order counting the number of inversions between them. It depends on the fact that the number of total inversions in a set ifs equal to the number of internal inversions in two sets, plus the number of inversions between the two sets. Moreover it is easy to find inversions between two ordered sets.

Note that we are assuming that the set we take as an input has a number or elements which is a power of two. If this is not the case we can just zero pad the input and run our algorithm the same way.

So, as in the book we will use several routines.

We call the first SortCount. It will take in a set L and sort it as well as count the number of significant inversions in it.

SortCount(L)

1. If L has only one element in it then return 0 since there can be no significant inversions in a set with only one element.
2. Otherwise do the following
   (a) Divide L into two halve
   (b) A contains the first $n/2$ elements in order
   (c) B contains the last $n/2$ elements in order
   (d) $(A, a) = SortCount(A)$ $(B, b) = SortCount(B)$
   (e) $t = Count(A, B)$
   (f) $L = Merge(A, B)$
   (g) Return $a + b + t$ and L which is now sorted

The second routine is called Count. It takes in two sets A and B which are both ordered, and returns the number of significant inversions between them.

Count(A,B)

1. Create pointers x and y initialized to point at the first elements of A and B respectively. Also create a t variable for the number of total significant inversions, initialized to 0.
   (a) While there remain items in either lists.
   (b) If the number pointed to by x is smaller than twice the one pointed to by y advance the pointer x.

(c) If the number pointed to by x is greater than twice the one pointed to by y then increment t by the number of elements remaining in A advance the pointer y

(d) Return t

The final routine, called Merge takes two ordered sets A, and B and returns a new set say C which consists of all the elements of A and B in order.

Merge(A,B)

1. Create pointers x and y initialized to point at the first elements of A and B respectively. Also create a set C, initially empty, to collect the merged list.

   (a) While either either pointer is pointing to its list.

   (b) If the number pointed to by x is smaller than the one pointed to by y add the number pointed to by x to C and advance x.

   (c) If the number pointed to by x is greater the one pointed to by y then add the number pointed to by y to C and advance the pointer y

   (d) If advancing one of the pointers would cause it to go beyond the end of its list just point it to ∞.

   (e) Return C

2.2 Correctness

Since the number of significant inversions in a list is equal to the number of significant inversions between the members of two halves of the list, together with the number of significant inversions between the two lists our algorithm is correct if and only if it can correctly detect the number of significant inversions between two ordered lists, and if it correctly merges two ordered lists into a new ordered list.

We have previously proven that the routine used to merge two ordered lists is correct in the context of mergesort.

Now it remains only to prove that if there is a significant inversion it is detected by the merge. It clearly does this though, since for every member say y of the second list there are precisely n significant inversions where n is the number of members of the first list which are more than twice as large as y.

Our routine compute exactly the sum of each n though since it does not proceed past a member of the first set until it has checked that the current (and thus every) member of the second set is not twice as large as it. Moreover, if it does detect that the current member of the second set is twice as large as the current member of the first one it increments the counter by the number of elements remaining in the first set - precisely those which are more than twice as large as the current element of the second set.

Therefore our routine determines the number of significant inversions in two ordered sets correctly, and our entire algorithm is correct.

Proof. □

2.3 Runtime

We note that this algorithm is almost identical to the one used in the book which was $O(n \log n)$ except, that where the book does just one linear time operation per recursive "level" we do too. However, both $T = n$ and $T = 2n$ correspond to $O(n)$. Thus our algorithms must have the same runtime.

More formally, the reoccurrence relation for this algorithm is of the form $T(n) = 2 \cdot T(n/2) + 2n$ and we know that the runtime corresponding to this relation is just $O(n \log n)$.

3 Problem 5.6

3.1 Algorithm

Initialize a pointer pointing to the root node. Also initialize a variable x with the probe value of the root node.

1. While the current node has children
   (a) Probe each child.
   (b) If both children have values greater than x the current node is minimal. Return it and end the algorithm.
   (c) Else change the pointer to refer to a child whose probe value is less than that of the current node, and set x to this child node's value

   Return the current node as it is minimal.

3.2 Correctness

The algorithm either moves on to a child node or returns a node at every execution of the loop. Furthermore if it reaches the end of the tree it returns a node. Thus it is clear that it always returns some node. Moreover the node that it returns must indeed be minimal since

Base Case: It is the root. Clearly the root has no parents of lower value since it has no parents. The algorithm checks to see that it has no children of lower value.

Induction Step: The only way that the algorithm could be looking at a given node is that it was lesser in value than its parent. Thus, it is a local minimum if and only if its children both have greater value than it does. This is however the criteria that the algorithm checks.

Special Case: Leaf: If the algorithm determines that the current node has no children the current node must be a local minimum since it has no children, and by construction its parent node has a greater value than it does.

3.3 Runtime

The algorithm runs in $O(\log n)$ probes since in the worst case it probes two or less nodes per "level" of the tree (both children of one node in the preceding leve). However the number of levels in a tree with n nodes is just given by $\log_2 n$. Thus it is on order $O(\log n)$ as desired.

4 Problem 1

4.1 Algorithm

We just use a simple algorithm. First we use as many quarters as possible, then as many dimes as possible, then nickels, and finally pennies.

More formally, let n denote the number of cents we need to return to the customer.

1. First return $\lfloor n/25 \rfloor$ quarters, setting $n = n - 25 \cdot \lfloor n/25 \rfloor$.
2. Then return $\lfloor n/10 \rfloor$ dimes, setting $n = n - 10 \cdot \lfloor n/10 \rfloor$
3. Next return $\lfloor n/5 \rfloor$ nickels, setting $n = n - 5 \cdot \lfloor n/5 \rfloor$
4. Finally return n pennies

4.2 Correctness

It is fairly clear that this algorithm will return correct change. It will never return more change than is appropriate. Before returning l coins of a given value m it checks that doing so will not result in returning too much change. In particular, if performing this operation would result in returning too much change then $l \cdot m$ would exceed the amount of money owed the customer n. Moreover the algorithm cannot return too little money since the last step will always pay the remaining amount owed the customer in pennies.

I claim furthermore that the algorithm gives the optimal solution.

Proof. Suppose our algorithm is not optimal. There are a few ways in which another algorithm might differ from ours. Firstly, it might use a different number of quarters.

However, in order to be optimal an algorithm must use the maximum number of quarters possible. If an algorithm does not, then there are other coins whose value, taken together is 25 cents exactly since any combination of pennies, nickels, and dimes which is worth at least 25 cents has a subset which is worth exactly 25 cents. If we take these coins, and replace them with a quarter then we will have improved the solution.

Thus, for a given n the number of quarters used in an optimal algorithm is uniquely determined. Moreover, the total value of all remaining coins must be 24 cents or lower. This, established we observe further that the optimal algorithm must maximize the number of dimes, among the remaining coins. For, if the

algorithm has at least 10 cents not in quarters or dimes it could reallocate it into dimes, thereby reducing the number of coins it uses.

Similarly, with a fixed number of quarters and dimes the total of the remaining value must be 9 or less, for if it were 10 or more then the maximum number of dimes, or quarter would not have been used. So, again the optimal algorithm must use the most nickels possible since if there are 5 or more pennies, they can be changed for a nickel, reducing the number of coins.

These steps uniquely determine an amount of pennies as well and our algorithm does precisely these steps. □

4.3 Runtime

Our algorithm is $O(1)$ since irrespective of n it just performs 3 division/floor operations along with 3 subtractions.

4.4 (b) Failure

One place where this algorithm might fail is in a currency where the denomination are $\{1, 5, 10, 13\}$. Faced with the task of returning 15 cents a greedy algorithm like ours would return a 13 cent coin and two 1 cent coins. This takes 3 coins total, whereas returning a 10 cent and a 5 cent coin results in the same value, but uses one less coin.