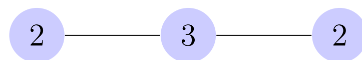
Rishad Rahman (Deepak's Recitation) February 11, 2015

6.046 Problem 1-1

Collaborators: *Cheng Wang*

(a) The graph below has 4 as the optimal value but the incorrect greedy algorithm would give 3 instead.



(b) **Algorithm:** Let u be any vertex in G and use BFS to convert G into a tree with root u. We output $\textsc{MaxProfit}(G, u)$.

$\textsc{MaxProfit}(G, u)$

    if $u = \emptyset$

        $G_u.profit = 0$
        return $\emptyset$

    else

        $S_1 = \bigcup_{v \in u.children} \textsc{MaxProfit}(G, v)$
        $S_2 = \bigcup_{v \in u.grandchildren} \textsc{MaxProfit}(G, v)$
        $M_1 = \sum_{v \in u.children} G_v.profit$
        $M_2 = p_u + \sum_{v \in u.grandchildren} G_v.profit$
        if $M_2 > M_1$

            $G_u.profit = M_2$
            return $\{u\} \cup S_2$

        else

            $G_u.profit = M_1$
            return S_1

Correctness: The top portion of the algorithm covers the base case of a u not being a root i.e. a tree with 0 levels. We now proceed via strong induction and we assume the algorithm holds for trees with up to $k \geq 0$ levels. A vertex u is either in the optimal set or not. If it is in the set, the profit is p_u plus the sum of the maximum profits of the trees rooted by the grandchildren of u since none of the children can now be in the set. Else if u is not in the set we just remove u, and sum over the trees rooted by the children of u. Obviously these trees are smaller than $|G|$ so the optimal sets/profits we get from them is correct by our assumption and the rest of the algorithm handles the logic for taking the set corresponding to a higher profit for G.

Runtime: A memo can be kept to make sure the recursion does not need to recompute subproblems. There are $\Theta(V)$ subproblems, 1 for each $v \in V$, with each subproblem taking $\Theta(c(v) + g(v))$ time where $c(v)$ is the number of children and $g(v)$ is the number of grandchildren of v. Note linked lists can make the union operation $O(1)$. Summing this over all v gives $\Theta(E)$ since this is equivalent to each edge being counted at most 2 times, for child and grandchild access. Hence our runtime, including the initial BFS, is $\Theta(V + E) = \Theta(V)$ since this is a tree.

(d) **Algorithm:** Again use BFS to convert G into a tree.

$L \leftarrow G.leaves$
MaxLocations(G)
  if $G = \emptyset$
    return $\emptyset$
  else
    $u \leftarrow L.pop()$
    $v \leftarrow u.parent$
    Remove u, v updating the pointers of v's neighbors accordingly to modify G.
    Also if v was the only child of its parent, insert that parent into L.
    return $\{u\} \cup$ MaxLocations(G)

Correctness: Base case is obvious. Otherwise, let G_1 be the graph after the modification, before the recursive step. Then our algorithm returns a set with length $1 + |\text{MaxLocations}(G_1)|$. Suppose on the other hand that u was not in our optimal set. This implies v is in the optimal set, otherwise we would be able to add u since it is a leaf connected to v. We then would have to remove u, v, *and* the neighbors of v reducing G to G_2 where $G_2 \subset G_1$. However this implies $|\text{MaxLocations}(G_2)| \leq |\text{MaxLocations}(G_1)|$ hence we cannot do better than when u is included.

Runtime: The recursion removes edges and points from G until it becomes $\emptyset$ hence our runtime is $\Theta(V + E) = \Theta(V)$.

(d) We use the algorithm as in (b) except we do not convert G into a tree (because we can't). Instead of children and grandchildren we have neighbors of distance 1 and 2 away and instead of using roots as a key we modify G into G_1 and G_2, corresponding to removing distance 1 neighbors and distance ≤ 2 neighbors, then recurse on both. Correctness follows easily since we are literally brute forcing based on whether u is in the optimal set or not. As a result our runtime is $O((V+E)2^E)$ since we take $O(V+E)$ time to iterate through a graph but there are $O(2^E)$ possible ways to go through the iteration.

Rishad Rahman (Deepak's Recitation) February 19, 2015

6.046 Problem 1-2

Collaborators: *Cheng Wang*

(a) The maximum distance in a $\frac{1}{2} \times \frac{1}{2}$ box is $\frac{\sqrt{2}}{2} < 1$.

(b) **Algorithm:**

FindBadDistance(S)

Let L be the set of points to the left of the median x-coordinate and R those to the right.
if FindBadDistance(L) $\vee$ FindBadDIstance(R)

return the pair found

else

Let S' be the set of points whose x-coordinate is < 1 away from the median x-coordinate, sorted by y-coordinate.
Let i iterate through S'

Let j iterate through the 11 closest points above i
if $d(i, j) < 1$

return (i, j)

Correctness: Base case is obvious, we are halving the problem size each time so eventually we will reach a size of 1 which is not a bad pair so we don't handle it. The divide part of our algorithm, checking L and R, sees if either side contains a bad pair, however it doesn't check to see if $d(p, q) < 1$ with $p \in L$ and $q \in R$. If this was the case then $q_x - p_x \leq d(p, q) < 1$, hence p, q must lie within the 2 unit strip centered on the median x-coordinate. We claim if this was the case, our algorithm will detect it. WLOG $p_y < q_y$ since we are iterating through S' in order of y-coordinate. If you divided the map into $\frac{1}{2} \times \frac{1}{2}$ squares so that the median coincides with the boundary of two consecutive squares, then we must have each point in the 2 unit strip must be in a unique square otherwise we would have two points on the same side in the same square which implies their distance is < 1 by (a) and we would've detected it in the recurrence. We have that there are 4 squares per row in our 2 unit strip. If $d(p, q) < 1$ we claim q cannot be in a square that is more than 2 rows above p's square which follows since $q_y - p_y \leq d(p, q) < 1 = 2 \times \frac{1}{2}$. Therefore (p, q) lie in a 3×4 set of squares where there is no more than 1 point per square hence there cannot be more than 11 points between them.

Runtime: The merge is $O(n)$ since we look at 11 points max per point in S' hence our recursion is $T(n) = 2T(\frac{n}{2}) + O(n)$ which by Master Theorem tells us our runtime is $O(n \log n)$.

(c) The algorithm is pretty much almost exactly the same as the one in (b) except a slight modification on the merge. We now let j iterate through the 23 closest points above i and instead of terminating once we find $d(i, j) < 1$ we let j finish iterating and add all j such that $d(i, j) < 1$ into a set J. Then we check all the pairwise distances in J and return (i, j_1, j_2) if $\exists j_1, j_2 \in J$ such that $d(j_1, j_2) < 1$. Correctness follows in the merge since there can be a maximum of 2 points per square now and if (i, j_1, j_2) has all its pairwise distances < 1 and WLOG $i \in L$ and $j_1, j_2 \in R$, then we must have j_1, j_2 must be in the 12 squares mentioned in the correctness of (b). At that point manually checking all possible j_1, j_2 suffices to see if the last distance is indeed < 1. The runtime is still $O(n \log n)$ since the merge time is still linear as the amount of time per point is still constant as we only need $\leq \binom{23}{2}$ comparisons.