Handout 22
April 04, 2002

Problem Set 5

This problem set is due **AT THE BEGINNING OF** class on *April 11, 2002*.

Both exercises and problems should be solved, but *only the problems* should be turned in. Exercises are intended to help you master the course material and will be useful in solving the assessed problems. You are responsible for material covered by the exercises. Each problem is to be done on a separate sheet (or sheets) of paper. Mark the top of each sheet with your name, 6.046J/18.410J, the problem number, your TA and recitation section, the date, and the names of any students with whom you collaborated.

Exercise 5-1. Soln: Anyone who did some studying got this one right.

Problem 5-1. Bottleneck Spanning Tree

(a) The following is an argument that a minimum spanning tree is a bottleneck spanning tree: A minimum spanning tree can be viewed as the set of edges selected based on weight (smallest first), such that no edge connects already-connected nodes.

Claim: The Minimum Spanning Tree of a graph depends only on the sorting of edges by weight, not on the weights themselves.

Proof. We assume Kruskal's algorithm is correct. Kruskal's algorithm doesn't use the actual value of the weight, only the sorted ordering. If the actual weight mattered, then Kruskal's algorithm wouldn't work. $\square$.

Let S be a minimum spanning tree of the graph. Assume S is not a bottleneck spanning tree. That means a tree could have been constructed that didn't use the heaviest edge in S. Let B be the tree that is a boottleneck spanning tree which only uses edges up to e_i (and S uses at least one edge greater than e_i).

Construct a new graph G_2 that has the same vertices as G, and the same edges as G. However, for all edges after e_i in sorted order, add a HUGE number to the edge weights. The sorting of the edges remains the same, so the minimum-spanning tree doesn't change (by the above theorem). The total weights for the bottleneck tree B doesn't change, but the total weight for S grows by at least 'huge number'.

Well, if we pick the huge number to be large enough, the total weight in B ¡ total weight in S. This means that there is a spanning tree whose total weight is less than that of the minimum spanning tree, which conotradicts our assumption that S is a minimum spanning tree.

Therefore, if S is a minimum spanning tree, it MUST also be a bottleneck-spanning tree.

(b) The following is a linear-time algorithm that given a graph G and an integer b decides if the value of the bottleneck spanning tree is at most b.

The idea is to assume that there is a way to reach every node in the graph by taking edges that have weight at most b. If we can do this, then clearly there exists a bottleneck spanning tree of weight b. If we can't, then we know that no such tree can exist.

What we do is a depth-first search through the graph, coloring every node we reach, and refusing to take edges whose weight is greater than b. When we are done searching the tree, we check if every node has been visited. If it has, a bottleneck spanning tree exists.

```
IS_BOTTLENECK(G, b)
for each vertex u ∈ V[G]
     do color[u] ← WHITE
u ← Some vertex in G
DFS(u, b)
for each vertex u ∈ V[G]
     do if color[u] = WHITE
          return false
return true
```

```
DFS(u, b)
color[u] ← GRAY
for each v ∈ Adj[u]
     do if w(v, u) ≤ b and color[v] = WHITE
          DFS(v)
color[u] ← BLACK
```

The above code has three parts. The first part paints all vertexes white, which clearly runs in time $\Theta(V)$. The last part scans through all vertices, which again run in time $\Theta(V)$. The middle part is basically a truncated depth-first search algorithm, which traverses all edges of weight at most b. This has a running time bounded by $\Theta(V + E)$.

Problem 5-2. Boy-Girl Matching For the end of term, we will be teaching tango dancing. Before we can do that, we need to pair up every male and female in the class (pretend that due to some miracle, the number of males and females in 6.046 is equal). We know the height of every student i to be h_i.

(a) Our algorithm tries to do the following: Take the shortest boy and shortest girl and pair them. Then repeat until there are no more students left. We will prove that this is an optimal strategy after describing the algorithm using pseudo-pseudocode.

```
TANGO-MERGE(class)
boys ← (boys ∈ class)
girls ← (girls ∈ class)
SORT(boys)
SORT(girls)
for i ← (1 → length(boys))
    PAIR(boy[i], girl[i])
```

Note that the above code is a very high-level description. First we analyze the running time of this code:

- Extracting the boys and girls can be done via a linear scan of the class. $\Theta(n)$
- sorting the boys and girls (based on height) depends on how the heights are presented. The example indicated that heights were all integers, so we can sort it in $\Theta(n)$. If it isn't, we can still sort it but in $\Theta(n \log n)$.
- Creating the pairs is once again a linear scan, which takes $\Theta(n)$.

Our running time is either $\Theta(n)$ if we assume student heights are integers (as presented in the example) or $\Theta(n \log n)$ otherwise.

Proof that the algorithm is optimal:

Assume P is an optimal pairing. Let G be the pairing assigned by our algorithm. Sort P and G on the boys' heights (and name if heights are not unique) so that we can compare it to G. The high-level approach is that we will make changes to P that keep it optimal, but that will in the end turn P into G. If P is still optimal, we have proven that G is an optimal assignment.

- if $P = G$ then G is optimal $\square$.
- else $P \neq G$. We know that P and G are the same length (all students must be paired). Let i be the first element in P that isn't the same in G. That is $P_i \neq G_i$. Since they are sorted on the boys' heights, they have the same boy but a different girl. Scan through the remaining pairs in P for the one that has the same girl, and swap girls. We know that this girl comes later in P (all the previous boys and girls lined up). Let (b_1, g_1) be the before pair, and (b_2, g_2) be the before pair that we swap girls with. Now we want to analyze what happens to the maximum Δ. Specifically, we want to show that it can't increase when we make this change. If we can show that the max-Δ for just the two pairs doesn't increase, and since we don't modify other elements, we have shown that the max-Δ everywhere doesn't increase.
  We know that $b_1 \leq b_2$ (sorted lists). We know that $g_1 \geq g_2$ since the greedy algorithm would have chosen g_2 otherwise. We know that the localized max-Δ before is $max(|b_1 - g_1|, |b_2 - g_2|)$. We know that after the swap it is $max(|b_1 - g_2|, |b_2 - g_1|)$. Pairwise, if we have two boys and two girls, there is no better strategy than pairing (large boy, large girl), (small boy, small girl). This is clear

since one of (large boy, small girl), (small boy, large girl) is guaranteed to be the largest distance possible.
So, our change in P preserves optimality locally, which preserves optimality globally. After this one change, P is closer to G by at least one more couple, and P is still an optimal solution. Now we go back to start and repeat again.

Clearly after —G— passes through this loop, P will remain optimal, and $P = G$. This proves that G is an optimal solution.

(b) Does the algorithm minimize the sum of Δ? The following proof shows that it does: Sort P and G on the boys' heights as before.

- if $P = G$, then clearly G is an optimal solution. □
- if $P \neq G$. as before, take the first difference i and swap girls. Now we analyze the effect of this swap. Let (b_1, g_1) be the old pair in P, (b_2, g_2) be the pair in P that had the girl we swapped with. We want to examine what happens to the total Δ, and so need to compare heights. Specifically, since only two pairs change, we want to make sure that the total change in $\Delta = (|b_1 - g_2| + |b_2 - g_1|) - (|b_1 - g_1| + |b_2 - g_2|)$ is not greater than zero. Because of the sorting, we know that $b_1 \leq b_2$. We know that $g_1 \geq g_2$ (if it wasn't, then g_2 would have shown up earlier in G, and so we contradict the fact that i is the first position where P and G differ).
We now analyze all the possible values of b_1, b_2, g_1, g_2. We know $b_1 \leq b_2$ and $g_1 \geq g_2$, which gives five possible ways of placing the numbers around:
  - $g_2 \leq g_1 \leq b_1 \leq b_2$: After swapping, we get $\Delta() = ((b_1 - g_2) + (b_2 - g_1)) - ((b_1 - g_1) + (b_2 - g_2))$ which reduces to $\Delta() = 0$. So, after the swap P remains optimal.
  - $g_2 \leq b_1 \leq g_1 \leq b_2$: Total $\Delta() = ((b_1 - g_2) + (b_2 - g_1)) - ((g_1 - b_1) + (b_2 - g_2))$. This reduces to $\Delta() = 2(b_1 - g_1)$. This value is not greater than zero, so P is still optimal after the change.
  - $g_2 \leq b_1 \leq b_2 \leq g_1$: Total $\Delta() = ((b_1 - g_2) + (g_1 - b_2)) - ((g_1 - b_1) + (b_2 - g_2))$. This reduces to $\Delta() = 2(b_1 - b_2)$. This value is not greater than zero, so P remains optimal.
  - $b_1 \leq g_2 \leq g_1 \leq b_2$: Total $\Delta() = ((g_2 - b_1) + (b_2 - g_1)) - ((g_1 - b_1) + (b_2 - g_2))$. This reduces to $\Delta() = 2(g_2 - g_1)$. This value is not greater than zero, so P remains optimal.
  - $b_1 \leq g_2 \leq b_2 \leq g_1$: Total $\Delta() = ((g_2 - b_1) + (g_1 - b_2)) - ((g_1 - b_1) + (b_2 - g_2))$. This reduces to $\Delta() = 2(g_2 - b_2)$. This value is not greater than zero, so P remains optimal.
  - $b_1 \leq b_2 \leq g_2 \leq g_1$: Total $\Delta() = ((g_2 - b_1) + (g_1 - b_2)) - ((g_1 - b_1) + (g_2 - b_2))$. This reduces to $\Delta() = 0$. So P remains optimal.

  i increments each time we do the above. Since i is increasing, we know once $i = |G|$, $G = P$. Since P is always optimal, this means G is optimal.

(c) Does the algorithm from part (a) maximize the total number of of equal-height pairs (where the $\Delta = 0$) ?

No it doesn't. For example, take the class formed of the following:

$$\begin{aligned} boys &= 1, 3, 5, 7 \\ girls &= 3, 5, 7, 9 \end{aligned}$$

The pairs the greedy algorithm generates are $(1, 3), (3, 5), (5, 7), (7, 9)$. HOWEVER, a different assignment could be $(1, 9), (3, 3), (5, 5), (7, 7)$. The latter maximized the total number of equal-height pairs, whereas my greedy algorithm clearly didn't.

Problem 5-3. Disjoint Sets Remembered

Argue that the total time spent on performing n MAKE-SET and m UNION operations is $O(n + m \log n)$.

SOLN: First we note that there is no way that $m > n$, since after n UNION operations, there is only one set left which contains all the elements.

So, for the case where $m = n$, we have to show $O(n + m \log n)$, which is the same as $O(m + n \log n)$ which was proven in lecture.

So, the only thing left to prove is for $m < n$.

First we note that every time an element in a set has its representative modified, the size of its set grows by ≥ 2. This comes directly from the fact that we only modify the representative of the smaller set.

Lemma 1*After m union operations, at most m elements have their representative modified (possibly more than once for some nodes).*

Proof. *by induction.*

- *Base Case: $m = 0$. No elements have had their representative changed, since there are no calls to* UNION. *So, true for base case.*
- *Inductive Step: Assume true for $m - 1$, show true for m. Well, after $m - 1$ operations, we have either one set left, or several sets. If we have several sets, they may be single-item sets, or multiple item sets. Lets itemize all the cases:*
  - *If there is only one set left, we really shouldn't be calling union again, but even so, no representatives are modified, so if it was true for $m - 1$, it's certainly true for m.*
  - *If there are several states, and at least one of the two states is a single-item set, the union operation will modify the representative of the single-item set (by our efficient way of doing unions). Therefore, assume the number of modifications for the $m - 1$ pass was z, we know $z \leq m - 1$. The new number of modifications is $z + 1$. We know $z + 1 \leq m$, so the inductive step is true for this case.*

– *The last possibility is that neither set is a single-item set. Note that the only way to make a non-single item set is by combining single-item sets. To make a length x set, you need to modify the representatives of $x - 1$ single-item sets. So, we have a length x and length y set that we're combining. We take the smaller one, and modify the representative of all the nodes in it. Since all the nodes except one have already had their representative modified, the sum of nodes who have had their representative modified ever is increased by one. Again $z + 1 \leq m$*

So, we know that there are at most m node items who have their representative modified (possibly more than once). we know that the cost for modifying a node item each time is $O(1)$. We also know that a node item can't be modified more than $O(logn)$ times, since each time it is modified, it enters a list that is at least twice as big.

So, m items have an $O(1)$ operation conducted on them up to $O(\log n)$ times. Therefore, the m union operations cost at most $O(m \log n)$. We only have to include the cost of the n MAKE-SET operations, which is $O(n)$, which gives us a total running time of $O(n + m \log n)$.