Darth Vader (dvader)
CS 157 — Spring 2012

Problem Set 1

February 18, 2012

Problem 1

The following is the sketch of a solution for problem 1. Some details are left to the reader - this solution would not receive full credit but is meant as an aid to study for the oral exams.

Part 1

The resulting graph has 3 nodes, c_1, c_2, c_3 with edges (c_1, c_2), (c_2, c_3) of weight 4 each.

Part 2

Claim 1. REDUCE *can be done in* $O(m)$ *time.*

Consider the following implementation.

Input: A graph $G = (V, E)$ with unique edge weights
Output: A reduced version of G

Darth Vader (dvader)
CS 157 — Spring 2012

Problem Set 1

February 18, 2012

REDUCE($G = (V, E)$)

1 $E(H) \leftarrow \{(u, v) : v$ is the closest neighbor of u in $G\}$
// scan the adjacency list of each u to find u's closest neighbor
// in time O(sum of degrees) $= O(m)$. Create a list of those edges $H \leftarrow (V, E(H))$
// from that list of edges and V, create the graph H in $O(m)$.
2 Do depth-first-search (or some other graph traversal) on H to find its connected components
// you want the output to be an array $C : V$ –¿ $\{1, 2, ..., \#$ of connected
// components of $H\}$ such that $C[v]$ =name of connected component containing
// v this array can be filled as you do DFS, so it takes O(edges$(H)) = O(n)$.
// create a new set of vertices V', one for each connected component
3 **for** each connected component c_i of H
4 **do** add a corresponding vertex v_{c_i} to V.
//It is associated with component c_i.
5 **for** each pair of connected components c_i, c_j of H
6 **do if** G has edges between c_i and c_j
7 **then** Add an edge between v_{c_i} and v_{c_j} to E whose weight is the minimum weight of all edges between c_i and c_j in G.
// Make a list L of edges: scan the edges of H, and for each edge u, v,
// add to list L edge $C[u], C[v]$ with the associated weight.
// Use radix sort to sort L in time $O(m)$. We sort first based on the value
// of $C[v]$, then the value of $C[u]$.
// Scan L to clean up: remove edges from a component to itself (these will
// be adjacent in the sorted list), and only keep the one with min weight
// among other pairs (we need traverse the list only once to do this)
// Scan cleaned up list and add each edge c_i, c_j to the adjacency lists
// of c_i and of c_j.
// (Note: c_i, c_j will have a secondary field linking back to the edge
// u, v that it came from in the original graph.)
// That takes constant time for each edge of L, so in total $O(n)$.
8 **return** $G = (V, E)$

Summing the runtime for each step gives $O(m)$ time.

Part 3

Input: A graph $G = (V, E)$
Output: The minimum spanning tree of G

FindMST($G = (V, E)$)

1 Run Reduce $\log \log n$ times, first inputting G
and then inputting the result of Reduce for each subsequent iteration.
2 Run Prims-Algorithm on the resulting graph
3 **return** all edges in G selected by any iteration of Reduce or by Prims-Algorithm

Part 4

Claim 2. *The algorithm in part 3 runs in $O(n \log \log n)$ time when $m = O(n)$.*

In part 2, we showed that Reduce runs in $O(m)$ time, where m is the number of vertices.

Therefore, $\log \log n$ iterations take $O(m \log \log n)$ time.

At each iteration of Reduce on a connected graph, the number of vertices is, in the worst case, halved. Therefore, we have at most $\frac{n}{\log n}$ vertices left. The runtime of Prims-Algorithm is then $O(m + \frac{n}{\log n} \log \frac{n}{\log n}) = O(m + n) = O(m)$. The runtime of reassembling the MST is linear in the number of edges, since we do constant work for each edge. The total runtime is therefore $O(m \log \log n + m + m) = O(m \log \log n) = O(n \log \log n)$ because $m = O(n)$.

Part 5

We want $O(m \log \log n) < O(m + n \log n)$. Therefore, we need $O(m \log \log n) < O(n \log n)$ and so $m < O(\frac{n \log n}{\log \log n})$.

Problem 2

Part 1

Left to the reader.[1]

Part 2 - The Algorithm

We are given a tree $T = \langle V, E \rangle$, with a root $m \in V$ and edge lengths L_e for $e \in E$. Let C_v be the set of children of any vertex $v \in V$ (note if these are not given, they can be calculated in linear time using depth first search). Let m_v be a value for each vertex $v \in V$. It will be assigned by the algorithm and store maximum communication time to a leaf of v's subtree. The following algorithm assigns the minimum delay values d_e

[1] Making such a remark in an assignment is worth 0 points.

to each edge $e \in E$ when executed on the root node, i.e. ASSIGN-DELAY($\langle V, E\rangle, m$) . Note, for clarity, any edge $e \in E$ may be represented by a tuple $\langle v, w\rangle$ where $v, w \in V$.

```
ASSIGN-DELAY(⟨V, E⟩, v)
 1  if |C_v| = 0
 2     then
 3          m_v ← 0
 4     else
 5          for ⟨v, w⟩ ∈ C_v
 6               do ASSIGN-DELAY(⟨V, E⟩, w)
 7          for ⟨v, w⟩ ∈ C_v
 8               do m_v ← max(m_v, L_⟨v,w⟩ + m_w)
 9          for ⟨v, w⟩ ∈ C_v
10               do d_⟨v,w⟩ ← m_v − m_w − L_⟨v,w⟩
```

Part 3 - Correctness

Claim 3. ASSIGN-DELAY *terminates.*[2]

Proof. Each call of the ASSIGN-DELAY works on the child of some node. The input, $\langle V, E\rangle$, is assumed to be a tree, therefore at some point each call reaches a leaf and the clause on line 1 becomes true. □

Claim 4. ASSIGN-DELAY *satisfies the feasibility criteria that the distance from the root to all leafs is equal.*

Proof. The approach will be by induction on the high of the tree (note that this precisely coincides with recursive call structure). The base case with no children is true because the communication time is trivially 0. Hence, we can assume that the distance from our children to their leaves is equal, and we need to show that the delays we assign will maintain equal communication to our children. More formally, assume we are at vertex v and the time from each of the children $w \in C_v$ to their leaves is m_w. We must show that, for some constant c, $L_{\langle v,w\rangle} + d_{\langle v,w\rangle} + m_w = c \ \forall w \in C_v$. From line 10 we know that $d_{\langle v,w\rangle} = m_v - m_w - L_{\langle v,w\rangle}$. Combining these facts we have that $c = m_v \ \forall w \in C_v$. Hence by design, the communication from v to any of its leaves is exactly m_v. □

Claim 5. ASSIGN-DELAY *satisfies the optimality criteria that* $\sum_{e\in E} d_e$ *is minimal.*

Proof. The approach is by contradiction. Assume some alternate weighing d^* was better than the weighting produced by ASSIGN-DELAY, d. That is $\sum_{e\in E} d^*_e < \sum_{e\in E} d_e$. This

[2]Termination is often obvious and unnecessary to state explicitly. In the case of recessive algorithms a simple explanation is a nice touch.

implies that $\exists\ e \in E$ such that $d^*_e < d_e$. Lets consider the point at which Assign-Delay chooses d_e. The argument implies that,

$$d^*_{\langle v,w\rangle} < d_{\langle v,w\rangle} = m_v - m_w - L_{\langle v,w\rangle}$$

Because $L_{\langle v,w\rangle}$ is fixed, there are two possibilities $m^*_v < m_v$ or $m^*_w < m_w$. The later case would imply yet another smaller edge case so w.o.l.g.[3] we can assume $m^*_w = m_w$ and need only consider $m^*_v < m_v$. However, combing that fact with the definition of m_v from line 8, we have,

$$m^*_v < \max_{\langle v,w\rangle \in C_v} L_{\langle v,w\rangle} + m_w$$

This violates the feasibility criteria because at least one edge will be shorter than it needs to be to ensure the distance to the leafs are all equal. This implies the existence of d^* is a contradiction.

□

Part 4 - Runtime

Answer 1

The Assign-Delay method is simply a modified version of Depth First Search (DFS). At each node in the search a constant amount of work is performed (note the simple arithmetic in lines 3, 8, and 10 are all constant time). Hence, the DFS retains its running time of $O(|V| + |E|)$ because the input is a tree the running time reduces to $O(|V|)$.

Answer 2

Based on the termination argument it is clear that Assign-Delay executes once for each node in V. The non-recursive body of the method (the loops on lines 7 and 9) takes $2|C_v|$ time. Combining these facts we have,

$$\sum_{n \in V} 2|C_v|$$

The sum of all the children of V exactly covers E. Hence, $\sum_{n\in V} 2|C_v| = 2|E|$. Because this is a tree the running time can be restated as $O(|V|)$

[3] without loss of generality

Problem 3

Part 1

$4, 3, 2, 1, 5$ is the optimal order with the total room volume over time being 182.

Part 2 - The Algorithm

Homework Sort the values $1..n$ in descending order by $\frac{v_i}{p_i}$.[4] The resulting order is optimal.

Part 3 - Correctness

A solution at the preferred level of detail (i.e. style points)

Let $\mathcal{O}^*$ denote an optimal ordering and $\mathcal{O}_{Alg}$ denote the ordering produced by the algorithm. We will use the function $\text{cost}(\mathcal{O})$ as a shorthand for evaluating $\sum_{i=1}^{n} v_i C_i$ on the ordering $\mathcal{O}$.

Claim 6. *$\mathcal{O}_{Alg}$ is at least as good as $\mathcal{O}^*$.*

First, observe that one can transform $\mathcal{O}^*$ into $\mathcal{O}_{Alg}$ by a sequence of adjacent swaps, similar to bubblesort: Let $\mathcal{O}$ be a transitional ordering initially starting at $\mathcal{O} = \mathcal{O}^*$, while the current ordering $\mathcal{O}$ is different from $\mathcal{O}_{Alg}$, take two adjacent jobs in $\mathcal{O}$ whose relative order is different in $\mathcal{O}$ and in $\mathcal{O}_{Alg}$, and modify $\mathcal{O}$ by swapping them. Repeat this process until $\mathcal{O} = \mathcal{O}_{Alg}$.

Claim 7. *When transiting from $\mathcal{O}^*$ to $\mathcal{O}_{Alg}$, each swap either reduces the cost or keeps it the same. Therefore $\mathcal{O}_{Alg}$ is optimal.*

Proof. Consider what changes during a single swap. Let i and j denote the two jobs being swapped, where i immediately precedes j in the transitional ordering $\mathcal{O}$ but is scheduled later than j in $\mathcal{O}_{Alg}$. By definition of the algorithm,

$$\frac{v_i}{p_i} \leq \frac{v_j}{p_j} \tag{1}$$

What is the effect of the swap? Let $\mathcal{O}$ denote the ordering just before and $\mathcal{O}'$ denote the ordering just after the swap.

- Jobs other than i and j are unaffected: They have the same completion times in both $\mathcal{O}$ and $\mathcal{O}'$.

[4] Sorting the reverse direction with an inverted ratio is the same, but several parts of the correctness proof would need to be updated.

- Job i now has a later completion time: $C'_i = C_i + p_j$.
- Job j now has a earlier completion time: $C'_j = C_j - p_i$.

The difference in cost is thus,

$$\text{cost}(\mathcal{O}') - \text{cost}(\mathcal{O}) = v_i C'_i + v_j C'_j - v_i C_i - v_j C_j = v_i p_j - v_j p_i = p_i p_j (\frac{v_i}{p_i} - \frac{v_j}{p_j})$$

The algorithm's property from equation (1) ensures $\frac{v_i}{p_i} - \frac{v_j}{p_j} \leq 0$ therefore, $\text{cost}(\mathcal{O}') \leq \text{cost}(\mathcal{O})$, which proves the claim. □

A solution that is more detailed than necessary

Let $\mathcal{O}_{Alg}$ be the ordering produce by sorting the values of $1..n$ and let the function $\text{cost}(\mathcal{O})$ be a shorthand for evaluating $\sum_{i=1}^{n} v_i C_i$ on the ordering $\mathcal{O}$. Also note that the objective can also defined non-recursively as,

$$\sum_{i=1}^{n} v_i C_i = \sum_{i=1}^{n} (v_i \sum_{j=1}^{i} p_j) = \sum_{i=1}^{n} \sum_{j=1}^{i} v_i p_j$$

Claim 8. *The algorithm satisfies the optimality criteria that $\sum_{i=1}^{n} v_i C_i$ is minimal.*

Proof. The approach is by contradiction. Assume there exists an ordering $\mathcal{O}^*$ for which $\text{cost}(\mathcal{O}^*) < \text{cost}(\mathcal{O}_{Alg})$. $\mathcal{O}^*$ implies that at least one item in $\mathcal{O}_{Alg}$ is out of order. Lets assume that $\mathcal{O}^*$ suggests the a^{th} item of $\mathcal{O}_{Alg}$ should move between the b^{th} and $b^{th}+1$ position. There are two cases, $a < b$ or $a > b$. Lets first consider $a < b$. The first step is to rewrite the cost and break it into the ranges $1..a-1, a, a+1..b, b+1..n$.

$$\begin{aligned} &\sum_{i=1}^{a-1} \sum_{j=1}^{i} v_i p_j + \\ &\sum_{j=1}^{a} v_a p_j + \sum_{i=a+1}^{b} \sum_{j=1}^{i} v_i p_j + \\ &\sum_{i=b+1}^{n} \sum_{j=1}^{i} v_i p_j \end{aligned} \tag{2}$$

To move a^{th} item between the b^{th} and $b^{th}+1$ positions causes the following modifications, we must remove a's pages (i.e. p_a) from all items in the range $a+1..b$ because it no longer precedes these items. And we must add the pages of $a+1..b$ to a because it now

proceeds those items. Formally,

$$\sum_{i=1}^{a-1}\sum_{j=1}^{i} v_i p_j + \sum_{i=a+1}^{b}[(\sum_{j=1}^{i} v_i p_j) - v_i p_a] + \sum_{j=1}^{a} v_a p_j + \sum_{j=a+1}^{b} v_a p_j + \sum_{i=b+1}^{n}\sum_{j=1}^{i} v_i p_j \quad (3)$$

By definition, $\text{cost}(\mathcal{O}^*) < \text{cost}(\mathcal{O}_{Alg})$ implies that equaHomeworktion (3) $<$ equation (2), i.e. $(2)-(3) > 0$. In subtracting equation (3) from equation (2) we can see that the terms in the ranges $1..a-1, b+1..n$ cancel and we only need to focus on the $a, a+1..b$ range, which is,

$$\sum_{j=1}^{a} v_a p_j + \sum_{i=a+1}^{b}\sum_{j=1}^{i} v_i p_j - \sum_{i=a+1}^{b}[(\sum_{j=1}^{i} v_i p_j) - v_i p_a] - \sum_{j=1}^{a} v_a p_j - \sum_{j=a+1}^{b} v_a p_j > 0$$

Notice all of the terms from equation (2) cancel and we are left with,

$$\sum_{i=a+1}^{b} v_i p_a - \sum_{j=a+1}^{b} v_a p_j > 0 \equiv \sum_{i=a+1}^{b} v_i p_a - v_a p_i > 0 \quad (4)$$

Now we are in a position to demonstrate the contradiction. For all i in $a+1 \leq i \leq b$, $i > a$, because $\mathcal{O}_{Alg}$ was constructed by descending values $\frac{v_i}{p_i}$, we have $\frac{v_a}{p_a} \geq \frac{v_i}{p_i}$ which alternatively is, $v_a p_i \geq v_i p_a$.[5] This sorted property contradicts equation (4) because,

$$v_a p_i \geq v_i p_a \; \forall \; a+1 \leq i \leq b \Rightarrow \sum_{i=a+1}^{b} v_i p_a - v_a p_i \leq 0$$

Recalling the beginning of this proof. The argument for the other case where $a > b$ is systemic. Thus, we have shown that it is impossible for any reordering to improve the solution of $\mathcal{O}_{Alg}$, therefore the existence of $\mathcal{O}^*$ is impossible.

□

Part 4 - Runtime

Sorting OMG! Q.-to the- E. -to the- D.[6]

[5] Assuming all inputs are positive.

[6] Footnotes are a great way to insert side comments demonstrating broader understanding. Just be careful that they don't state incorrect properties without justification. That does quite the opposite.

Darth Vader (dvader)
CS 157 — Spring 2012

Problem Set 1

February 18, 2012

Problem 4

Part 1

a.) Yes b.) No

Part 2

Verify by inspection. An example is 2 2 4 4 4

Part 3

Our algorithm will operate on the numbers $d_1, d_2, ..., d_n$. After the operations are complete, we will inspect $d_1, ..., d_n$ and decide whether a satisfactory graph exists.

- **Step 1:** Sort the node degrees in increasing order in linear time. w.l.o.g., $d_1 < d_2 < ... < d_n$
- **Step 2:** Initialize current element 'd_c' to be d_n, the largest node degree.
- **Step 3:** Subtract 1 from $d_{c-1}, d_{c-2}, ...$, etc. until d_c subtractions have been performed. If a d_i becomes 0 after its subtraction, remove it from the list. If there are no more non-zero entries in the list from which to subtract (i.e. if we CANNOT perform d_c distinct subtractions), the algorithm terminates and we say that no such graph exists. If d_c subtractions CAN be performed, set d_c to 0.
- **Step 4:** If all d_c subtractions succeed, set the current element 'd_c' to be the largest remaining nonzero d_i and re-sort the list. If no such d_i exists (all d_i are 0), the algorithm terminates and we say that such a graph DOES exists. Otherwise, repeat from step 2 until termination.

Note: care must be exercised to maintain a linear running time. once the degree list L is sorted, we inspect the values in L and create one 'bucket' for each value. the buckets themselves will form the elements in a linked list, and we maintain a sentinel for the non-vacant bucket with the largest corresponding node degree. AFTER the completion of ALL of the subtractions in Step 3, we move each d_i that has been decremented (in this iteration) to its new bucket (can do this simply and efficiently by maintaining a list of nodes affected during Step 3 and updating that list after Step 3).

Part 4

Terminology Specific to This Solution: We say that an undirected graph is valid if it is finite, has no multiple edges, and has no self loops. We say that a list L of node

degrees is valid if there exists a valid graph G whose node degrees are precisely L. Also, the numbers $d_1, ..., d_n$ may change during our algorithm, but we understand that such changes are 'local' in scope and when determining whether L is valid for G, we use the ORIGINAL values of $d_1, ..., d_n$.

Format of proof: We demonstrate that,

a) For every (finite) set of node degrees L that our algorithm says is valid, we can construct a valid graph G with node degree list L

b) For every valid graph G with node degree list L, our algorithm will decide that L is valid

and conclude that for a given list L of node degrees, our algorithm will say that L is valid if and only if a graph G exists with node degree list L. Hence, our algorithm is correct.

a.) Is straight forward: for each d_i, create a node. perform the above algorithm to check validity, but this time, each time a node degree subtraction is performed in Step 3, add an edge between the node corresponding to d_c and the node corresponding to d_s, where d_s is the current node degree that is being decremented.

From now on, we abuse notation and refer to the node corresponding to d_i as d_i, despite the fact that d_i is the degree of that node (or a quantity in our algorithm that is changing).

b.) Is more interesting: sort the node degrees as in Step 1. we will rearrange the edges of G so that the node degree list L is preserved but the edges of G will 'obviously' be generated by our method in **(a.)**. In fact, proceed as if **(a.)** were being executed. Each time you go to add an edge to G (via Step 3), instead of adding an edge you will check to see if that edge is already in G. If it is, proceed. If it is not, we switch the endpoints of two edges in a way that preserves node degrees but ensures that the edge we want is included. At the end, all edges will be as if they were generated by **(a.)**, and hence will be valid. All that is left to explain is the 'endpoint switching'.

Suppose we are inspecting d_c and d_{c-1} (with $d_{c-1} \leq d_c$) and notice that there is no edge between d_c and d_{c-1}. Assume that $2 \leq d_{c-1}$. Since there is no edge between d_{c-1} and d_c and both d_{c-1} and d_c have at least 2 neighbors, we can find $j, k < (n-1)$ (with j and k distinct) such that (an edge exists between d_n and d_j) and (an edge exists between d_{n-1} and d_k) and (no edge exists between d_j and d_k). Remove edges (d_n, d_j) and (d_{n-1}, d_k) and insert edges (d_n, d_{n-1}) and (d_j, d_k). Node degrees have been preserved, and the edge that we desire is now in the set. If $1 = d_{n-1}$, we can just swap d_{n-1} with d_j in the list order (or replace the edges appropriately) to obtain the desired edge while maintaining node degree counts. Other edge cases are similar. One last detail: if every d_j, d_k pair we consider has an edge (d_j, d_k) already, then d_j and d_k necessarily have inappropriately large degrees and thus shouldn't be candidates (contradiction).

Problem Set 1

Part 5

While traversing our linked list in Step 3 and subtracting 1 from each of d_c node degrees, we make d_c traversals. Each node is the current node at most once, hence the number of traversals is bounded above by the sum of the node degrees.