College of Computer & Information Science
Northeastern University
Fall 2017

CS5800
MS Algorithms
7 November 2017

Sample Solutions to Problem set 4

1. (5 points) Identifying most influential reachable nodes

Let $G = (V, E)$ be a directed graph in which each vertex v has an integer label $\ell(v)$ that denotes the *influence* of v. Assume for simplicity that all the influences are distinct. For each $v \in V$, let $r(v)$ denote the vertex in G that has the largest influence among all vertices reachable from v in G.

Design an algorithm that takes as input G, with the influence values $\ell(v)$ for each v, and computes $r(v)$ for each vertex v; that is, your algorithm should return the following set:

$$\{\langle v, r(v)\rangle : v \in V\}.$$

Prove the correctness of your algorithm. Analyze its worst-case running time. The more efficient your algorithm is in terms of its asymptotic worst-case running time, the more credit you may get.

Answer: We will use the concept of strongly connected components (SCCs). We note if that if u and v are in the same SCC, then $r(v) = r(u)$ since u and v can reach exactly the same vertices. Here is the algorithm.

1. Find the SCCs of G. This returns a list L of SCCs C_1, C_2, …, C_k, and a dag D, in which each node is an SCC. We can also assume that for each vertex u, we have the SCC (call it $C(u)$) that contains it.
2. For each component C in L, we determine $m(C)$ to be the vertex that has the maximum influence in C.
3. Compute a topological ordering of D.
4. For component C in reverse topological order:
   (a) Set $r(C) = m(C)$. (we are setting $r(C)$ initially to be vertex that has most influence in C)
   (b) For every edge (C, C') in the dag D: if $\ell(r(C)) < \ell(r(C'))$, then set $r(C) = r(C')$ (we have found that the most influential vertex reachable from component C' has more influence than the most influential vertex in C, so we need to update $r(C)$).
5. For each component C:
   (a) For each vertex v in C, set $r(v) = r(C)$.
6. Return the set of pairs $\{\langle v, r(v)\rangle : v \in V\}$ as desired.

The correctness of the algorithm follows by induction in the reverse topological ordering of the dag D. For the base case, we consider a component C that has no outgoing edges. In this case, the above algorithm sets $r(C)$ to be the most influential vertex in C (step 4a) and $r(v)$ (for all v in C)

to be $r(C)$. This is correct since each vertex in C can only reach other vertices in C (since C is a sink component).

Now we prove the induction step. Suppose we are processing a component C at a particular point in step 4. Before this, by the choice of the ordering and the induction hypothesis we have calculated $r(C')$ correctly for all C' to which C has an outgoing edge. Now, the most influential vertex reachable by a vertex in C is either in C or reachable from C' for some C' to which C has an edge. Since step 4b takes the maximum of $m(C)$ and $r(C')$ over all such C', it follows that $r(C)$ is calculated correctly. And step 5 calculates $r(v)$ correctly.

We now compute the running time of the algorithm. Step 1 takes $O(n+m)$ time. Step 2 takes $O(n)$ time since it is a linear scan through all the connected components and their vertices. Step 3 takes $O(n+m)$ time since D has no more edges and vertices than G. Step 4 is a traversal through D, taking $O(n+m)$ time. Step 5 takes $O(n)$ time. So total time is $O(n+m)$.

2. (5 points) Determining a certain cycle

Let $G=(V,E)$ be a directed graph. Let $v \in V$ and $u \in V$ be two different vertices in V. Design an algorithm that determines if there exists a path from v to u that contains a (non-empty) cycle.

Prove the correctness of your algorithm. Analyze its worst-case running time. The more efficient your algorithm is in terms of its asymptotic worst-case running time, the more credit you may get.

Answer: Answer: The idea is the following. If a vertex x is on some path from v to u, then there is a path from v to x, and in the graph G^t there is a path from u to x. If we take only those vertices that are reachable from v in G, and are reachable from u in G^t, we get only vertices that are on some path from v to u. It is then enough to see if there is a cycle in the graph that contains only those vertices.

We can apply DFS to G with v as source vertex to get a tree with all the vertices that are reachable from v. We consider the subgraph G' of G that is defined by the vertices that are reachable from v. That is, $G' = (V'E')$, where $V' = \{x \in V | x \text{ is reachable from } v\}$ and $E' = \{(a,b) | (b,a) \in E \text{ and } a,b \in V'\}$. If we apply DFS to $(G')^t$ with u as source vertex, we will get a tree with only those vertices that are reachable from both u in $(G')^t$. Vertices that are reachable from u in $(G')^t$ are vertices in G' that has a path connecting them to u in G'. We now construct another graph $G'' = (V'', E'')$ where $V'' = \{x \in V' | \text{x is reachable from } u \text{ in } (G')^t\}$ and $E'' = \{(a,b) \in E | a,b \in V''\}$. Next we look for cycles in G" using DFS. If there is a cycle in G'', then there is a path from v to u with a cycle, otherwise there is no such path.

3. (5 points) Total weighted finish time

A scheduler needs to determine an order in which a set of n processes will be assigned to a processor. The ith process requires t_i units of timw and has weight w_i. For a given schedule, define the *finish time* of process i to be the time at which process i is completed by the processor. (Assume that the processor starts processing the tasks at time 0.)

For example, consider 4 processes with $t_1 = 5$, $t_2 = 2$, $t_3 = 7$, $t_4 = 4$. And weights $w_1 = 1$, $w_2 = 3$, $w_3 = 2$, and $w_4 = 2$. Consider the schedule $1, 3, 2, 4$. The finish time for process 1 is 5, for process 2 is $t_1 + t_3 + t_2 = 5 + 7 + 2 = 14$, for process 3 is $t_1 + t_3 = 5 + 7 = 12$, and for process 4 is $t_1 + t_3 + t_2 + t_4 = 5 + 7 + 2 + 4 = 18$. The *total weighted finish time* of the schedule is then $5 \cdot 1 + 14 \cdot 3 + 12 \cdot 2 + 18 \cdot 2 = 5 + 42 + 24 + 36 = 107$.

Give a greedy algorithm to determine a schedule that minimizes the total weighted finish time. Prove the correctness of your algorithm, and analyze its worst-case running time.

Answer: Suppose we only had two tasks. In what order would we schedule them? If we did $1;2$, then the total weighted finish time would be $w_1t_1 + w_2(t_1 + t_2)$. If the schedule was $2;1$, then the total weighted compleion time would be $w_2t_2 + w_1(t + 1 + t_2)$. Both have a common term $w_1t_1 + w_2t_2$; so the first schedule is better if

$$w_2t_1 < w_1t_2 \Rightarrow \frac{t_1}{w_1} < \frac{t_2}{w_2}.$$

Consider the following simple algorithm.

- Schedule the tasks in increasing order of their t_i/w_i values.

The running time of the algorithm is just the time needed for sorting, which is $\Theta(n \log n)$.

The proof is by contradiction. Suppose a different schedule has a smaller total weighted finish time. Then, it must have two consecutive tasks i and j such that $t_i/w_i > t_j/w_j$, yet i is scheduled right before j. If we swap i and j in the schedules, then (i) the weighted finish time of every job scheduled before i and j remains the same, (ii) the weighted finish time of every job scheduled after i and j remains the same, and (iii) the difference between the sum of the *new* weighted finish times of i and j and that of the *old* weighted finish times of i and j is $w_it_j - w_jt_i < 0$. So the total weighted finish time of the new schedule is strictly less than that of the old (supposedly optimal) schedule. Hence, we have a contradiction.

4. (5 points) "Three-Ary Huffman"

You are given a file with characters drawn from an alphabet of size n, with the ith character of the alphabet having frequency $f[i]$. Your goal is to construct a modified "Three-Ary" Huffman code where you are allowed to use 0, 1, or 2 in your codeword. Design an algorithm that encodes each of the n characters with a variable-length codeword over the values 0, 1, and 2 such that no codeword is a prefix of another codeword and so as to obtain the maximum compression.

Prove the correctness of your algorithm and analyze the worst-case running time of your algorithm.

Answer: Suppose we are given an alphabet of size n. It is tempting to consider the following generalization of Huffman's algorithm. If $n \leq 2$, then have a root r with its children the nodes corresponding to the two characters. If $n \geq 3$, then select the 3 characters with minimum frequency (say a, b, and c) and make these characters siblings with a single parent, introduce a new corresponding to the parent assigning it with the frequency being the sum of the frequencies of a, b, and c, and then recurse.

There is a problem with the above approach, though. Notice that the above algorithm may produce a tree in which every internal node has 3 children except for the root which has 2 children; this is because n will be 2 only at the end of the computation. An optimal three-ary code, however, should not have the root with only two children (for $n > 2$). For instance, consider the example with 4 characters a, b, c, and d with frequencies 1, 2, 3, and 4, respectively. The above algorithm will have a, b, and c as siblings and their parent being a sibling of d. The overall cost will be $2 * (1 + 2 + 3) + 4 = 16$. The optimal solution, however is to have a and b as siblings and their parent as sibling with c and d to give a cost of $2 * (1 + 2) + 3 + 4 = 13$.

So how do we generalize Huffman encoding? If we are guaranteed that every internal node has 3 children, then we can apply the above greedy choice. But the preceding condition may not always hold, as we saw in the case $n = 4$. However, we can see that in an optimal tree there can be only one internal node with fewer than 3 children; furthermore, such an internal node should have two children and should be an internal node with largest depth. Why?

First, any internal node u with only child v can be removed and v made the child of the parent of u, hus decreasing the cost of the tree. Now consider the case when there are two internal nodes u and v each having two children. Let u_1 and u_2 be the children of u and v_1 and v_2 be that of v. Without loss of generality, suppose that the depth of u is smaller than the depth of v. We can remove v and make v_1 the child of u and v_2 the child of the parent of v, thus decreasing the cost of the tree, again a contradiction. We leave it as an exercise to the reader to argue that if an internal node does have fewer than 3 children it must be an internal node with largest depth.

We now know that an optimal prefix-free three-ary code can be one of two kinds: (i) every internal node has exactly 3 children; or (ii) there exists one internal node with 2 children, all others have 3 children, and the node with 2 children has largest depth. Given a character set of size n, can we determine which of the two types the optimal tree T is? Somewhat surprisingly, yes. Let k be the number of internal nodes in T. If T is of type (i), then the number of edges in the tree, which is $k + n - 1$, also equals $3k$. We thus get $2k = n - 1$. Since k and n are integers, it follows that n is odd. On the other hand, if T is of type (ii), then the number of edges in the tree, which is $k + n - 1$, also equals $3k - 1$. We thus get $n = 2k$. Hence, we have proved that the optimal tree is of type (i) if n is odd; otherwise, it is of type (ii).

We thus have the following algorithm. If n is 1, there is nothing to do. If $n \geq 2$ and odd, then we select the 3 lowest frequency characters, make them siblings, add a parent character with frequency of their sum and repeat. Note that when we repeat, the new value of n is two less than before, implying that the new value is still odd. Similarly, if $n \geq 2$ and even, then we select the 2 lowest frequency characters, make them siblings, add a parent character with frequency of their sum and repeat. Now when we repeat, the new value of n is odd. And from now on, the algorithm will repeatedly select the 3 lowest frequency characters until there is exactly one character left.

Greedy choice: If n is odd, then we select the 3 smallest frequency characters and make them siblings. We have already argued that when n is odd, the optimal tree is of type (i). Given this, we can invoke a swapping argument similar to the binary case to claim that the 3 smallest frequency characters have to be siblings. When n is even, we know that the optimal tree is of type (ii). Therefore, the internal node with two children needs to be of largest depth. We invoke a swapping argument similar to the binary case to claim that the 2 smallest frequency characters have to be siblings, and their parent must be the internal node with two children.

The remainder of the proof – that once the 3 or 2 smallest frequency characters are combined, the remaining problem is to compute an optimal solution for the instance with the combined characters – is almost identical to that for the binary case the only difference is that we are working with three-ary trees and the proof we went through in class is working with binary trees.

5. (5 points) Planning a largest party that promotes diverse social interactions

You want to throw a big party and have put together a list of your n friends. You also know who knows whom, so you have a list containing which pairs of people know each other. Being a major socialite and a networking guru, you decide that it would best to invite the maximum number of friends possible, subject to the following constraint: at the party, each person should have at least four people they know and at least four people they do not know.

Design an efficient algorithm that takes as input the list of n people and the list of pairs who know each other and outputs the best choice of party invitees. Analyze the worst-case running time in terms of n. The more efficient your algorithm is in terms of its asymptotic worst-case running time, the more credit you may get.

Answer: Here is a simple greedy procedure. First, form the graph based on the given data so that each vertex is a person and two persons have an edge if they know each other. Repeatedly do the following: If you find a vertex that knows fewer than four persons (from the current set left) or knows all but at most three persons (from the current set left), remove it. The set of persons remaining form the party.

One challenge in implementing the above algorithm is to maintain the degrees and identifying the lowest and highest degree vertices. One possibility is to maintain a binary search tree (or a heap) of the vertices, based on their degrees, removing or decrementing them as we keep removing vertices from the graph. This will take $O(m \log n)$ time, where m is the number of edges and n is the number of vertices.

One can, however, implement the above algorithm more efficiently in $O(m + n)$ time by a more clever use of simpler data structures. We maintain a current degree for each vertex. And, in fact, we can maintain a sorted list of these degrees in $O(n)$ time since each degree is between 1 and $n - 1$. In particular, we can maintain an array $A[1..n - 1]$ such that $A[i]$ is a doubly linked list of all vertices that have degree i. Note that $A[i]$ could be empty.

1. Build knowledge graph G.
2. For each node v:
   (a) Calculate degree $d[v]$. Insert v into linked list $A[d[v]]$.

   Note that this step takes time $O(n + m)$ overall.
3. Set $p = n$ (initial number of vertices in the graph) and D to be the maximum degree. This takes time $O(n)$.
4. While $p \geq 9$:
   (a) Let B be the boolean ($A[1]$ is empty) and ($A[2]$ is empty) and ($A[3]$ is empty) and ($D \leq p - 4$). This takes $O(1)$ time.
   (b) If B is true then return the current set of vertices and exit, else
      i. If $D > p - 4$, then let u be any vertex in $A[D]$, else let u be any vertex in $A[1]$, $A[2]$, or $A[3]$. This takes $O(1)$ time.
      ii. Remove u from the list it lies in, and for every edge (u, v) in G remove the edge, move v from list $A[d[v]]$ to $A[d[v] - 1]$ and decrement $d[v]$ by 1. This takes time as much as the degree of u. So overall, across all the iterations, this step only takes $O(m)$ time.
      iii. If $A[D]$ becomes empty, then set D to be the largest index i for which $A[i]$ is nonempty by doing a backward scan from index D in A. Across the entire algorithm, this step only takes $O(n)$ time.

5. If we arrive at this step, then we have removed all but 8 vertices. It is now impossible to organize the desired party. So we indicate so and return.

As we have reasoned above, the running time is $O(n + m)$.

6. (2 + 3 = 5 points) Spanning trees and cycle representatives

Let $G = (V, E)$ be a connected undirected graph, and let $w : E \to \mathbb{R}_{>0}$ be a (positive) weight function on the edges of G.

(a) We define a *maximal spanning tree* for G to be a spanning tree T such that $\sum_{e \in T} w(e)$ is maximal among all the spanning trees of G. Give an algorithm that finds a maximal spanning tree for G.

Answer: Use the same algorithm as MST, with min and max reversed.

(b) We say that a set $F \subseteq E$ of edges *represents* all the cycles of G if every cycle in G has an edge in F; that is, for every cycle C of G, $C \cap F \neq \emptyset$. Give an algorithm that finds a set $F \subseteq E$ of *minimal weight*, such that F represents all the cycles of G.

Answer: Consider such a set F. $E - F$ cannot have cycles since otherwise F does not represent all cycles. So $E - F$ must be a forest. Since F needs to have maximal weight, $E - F$ is a maximal forest. Since G is connected and all weights are positive, $E - F$ is a maximal spanning tree. Use algorithm of (a) to find $E - F$, and return the complement of this set.