CS60003 Algorithm Design and Analysis, Autumn 2010–11

Mid-Semester Examination

Maximum marks: 50 | September 18, 2010 (AN) | Total time: 2 hours

---

Roll no: _____ **Name:** _____

[*Write your answers in the question paper itself. Be brief and precise. Answer all questions.*]

1. Let S, T_1, T_2 be strings of lengths n, m_1, m_2 with $m_1 + m_2 \leqslant n$. Your task is to locate whether the pattern $T_1 * T_2$ (that is, T_1 followed by zero or more symbols followed by T_2) can be found in S.

(a) Show by means of an example that there can be $\Theta(n^2)$ different matches of $T_1 * T_2$ in S. **(5)**

Solution Take $S = a^n$, $T_1 = T_2 = a^{n/4}$. The total number of matches of $T_1 * T_2$ in S is $(\frac{n}{2}+1)+\frac{n}{2}+(\frac{n}{2}-1)+\cdots+1 = \frac{1}{2}(\frac{n}{2}+1)(\frac{n}{2}+2) = \frac{1}{8}(n+2)(n+4) = \Theta(n^2)$.

(b) Supply an $O(n^2)$-time algorithm to compute all matches of $T_1 * T_2$ in S. **(5)**

Solution

1. Use the KMP algorithm to find all matches of T_1 in S. This takes $O(n)$ time.
2. Use the KMP algorithm to find all matches of T_2 in S. This again takes $O(n)$ time.
3. For each match position j_1 of T_1 and for each match position j_2 of T_2, report a match (j_1, j_2) of $T_1 * T_2$ in S if and only if $j_1 + m_1 \leqslant j_2$. Since there are at most $O(n)$ matches of T_1 or T_2 in S, the running time of this step is $O(n^2)$.

Note that Step 3 can be implemented more efficiently. However, in view of Part 1(a), we cannot avoid a $\Theta(n^2)$ running time in the worst case.

(c) Supply an $O(n)$-time algorithm to decide whether there is any match of the pattern $T_1 * T_2$ in S. **(5)**

Solution

1. Use the KMP algorithm to compute the leftmost match of T_1 in S. If no matches are found, return *false*, else go to Step 2. This step takes $O(n)$ time.
2. Let j be the position of the leftmost match of T_1 in S. Run the KMP algorithm to find a match of T_2 in $S[j + m_1 \ldots n - 1]$. If the KMP algorithm returns *no match*, return *false*, else return *true*. This step also takes $O(n)$ time, and can be used for example, to detect the leftmost match of $T_1 * T_2$ in S.

2. Prof. Avarice proposes an algorithm to compute the minimum spanning tree in a connected undirected graph $G = (V, E)$ with a cost $c(e)$ associated with each edge $e \in E$. Your task is to assist Prof. Avarice by supplying an efficient algorithmic implementation of his idea.

(a) Propose an efficient algorithm that, given an edge $e \in E$, determines whether or not e belongs to a cycle in G. What is the running time of your algorithm? **(5)**

Solution The following algorithm is based on the fact that e lies on a cycle if and only if $G - e$ is connected.

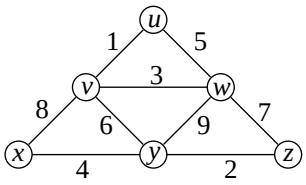
1. Delete the edge e from the given graph $G = (V, E)$. Call this graph G'.
2. Run a breadth-first or depth-first traversal in the graph G' obtained in Step 1. If the traversal indicates that G' is disconnected, return *false*, else return *true*.

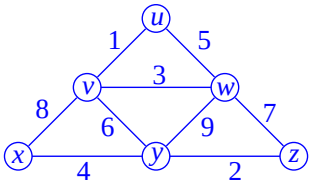
Step 1 can be completed in $O(1)$ time for adjacency-matrix representation, or in $O(|V|)$ time for adjacency-list representation. A BFS or DFS in G' can be implemented to run in $O(|E| - 1 + |V|)$ time. Therefore, the total running time of the above algorithm is $O(|E| + |V|)$.

(b) The MST algorithm of Prof. Avarice goes as follows.

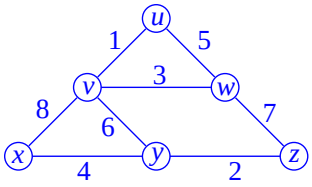
1. Sort E in non-increasing order of the edge costs. Store this sorted list in T.
2. So long as G contains more than $|V| - 1$ edges, repeat Steps 3–5:
3. Pick the edge e from T with largest cost.
4. If e belongs to a cycle in G, remove e from E.
5. Remove e from T.
6. Output the reduced graph (V, E).

Demonstrate how Prof. Avarice's algorithm works on the following graph. **(5)**

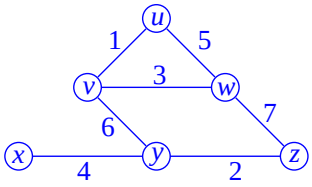




(a) The original graph

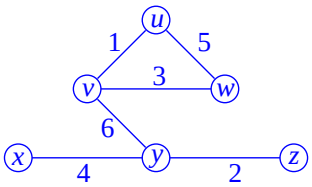


(b) Deletion of edge of cost 9

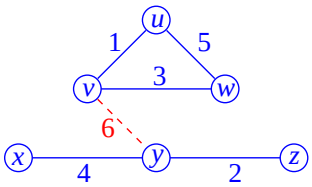


(c) Deletion of edge of cost 8

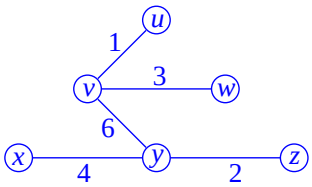
Solution



(d) Deletion of edge of cost 7



(e) Edge of cost 6 is not deleted



(f) Deletion of edge of cost 5

(c) Describe an efficient implementation of Prof. Avarice's algorithm. What is the running time of your implementation? **(5)**

Solution Let $n = |V|$ and $m = |E|$. Steps 1, 3 and 5 deal with the edge set of G. If the set is first sorted, then Step 1 requires $\mathrm{O}(m \log m)$ time, and each execution of Steps 3 and 5 takes $\mathrm{O}(1)$ time, so the total time for these three steps is $\mathrm{O}(m \log m)$. We may alternatively use a max-heap to implement this set T. If so, $\mathrm{O}(m)$ time is needed for initial building of the heap. Subsequently, each max finding or max deletion (Steps 3 and 5) takes $\mathrm{O}(\log m)$ time. With this implementation too, the total time for Steps 1, 3 and 5 is $\mathrm{O}(m \log m)$.

The most costly step turns out to be Step 4. This is executed $\mathrm{O}(m)$ times with each execution taking $\mathrm{O}(n+m)$ time (by Part 1(a)). So the total complexity of Step 4 is $\mathrm{O}(m(n+m))$. Since G is connected, $m \geqslant n-1$, so this complexity is $\mathrm{O}(m^2)$.

To sum up, Prof. Avarice's algorithm can be implemented to run in $\mathrm{O}(|E|^2)$ time.

(**Remark:** This is significantly poorer than $\mathrm{O}(|E| \log |V|)$ running times of Prim's and Kruskal's algorithms. The difficulty with Prof. Avarice's algorithm is that Step 4 turns out to be too costly. Both Prim's and Kruskal's algorithms cleverly avoid such costly checks for cycles or connectedness.)

(d) Although Prof. Avarice's algorithm may be poorer than Prim's and Kruskal's MST algorithms in terms of running time, a more potent danger awaits you. Prove or disprove: Prof. Avarice's algorithm always outputs a minimum spanning tree of a connected graph. **(5)**

Solution Let $e = (u, v)$ be an edge of G of maximum cost among those that lie on cycles. It suffices to prove that G has a minimum spanning tree not containing the edge e. Let T be a minimum spanning tree of G, that contains the edge e. Removing e from T disconnects T to two components P and Q (the first containing u, and the second v). Since e lies on a cycle in G, this cycle must contain an edge $e' = (u', v')$ of G with $u' \in P$ and $v' \in Q$. We may have $u' = u$ or $v' = v$, but not both. Since $T - e + e'$ is again a spanning tree of G, and T is a *minimum* spanning tree of G, we must have $c(e') = c(e)$, that is, $T - e + e'$ is again a minimum spanning tree of G.

3. We want to merge n sorted lists $L_1, L_2, \ldots, L_n$ of sizes $l_1, l_2, \ldots, l_n$, respectively. Suppose that at any point of time, we are allowed to merge only two sorted lists, that is, simultaneously merging t sorted lists for $t \geqslant 3$ is not allowed. (For example, the sorted lists may be residing in files in a palmtop computer which has very little memory and allows only three opened file pointers at any time.) The effort associated with merging two sorted lists of sizes u and v is taken as $u + v$. Your task is to select the merging sequence in such a way that the total effort of merging the given n lists is minimized.

(a) Suppose that $n = 4$ lists are given with respective sizes $10, 20, 30, 40$. Find the efforts of the following two ways of merging L_1, L_2, L_3, L_4: **(5)**

$$\text{merge}(\text{merge}(L_1, L_2), \text{merge}(L_3, L_4)) \text{ and } \text{merge}(\text{merge}(\text{merge}(L_1, L_2), L_3), L_4).$$

Solution $\text{effort}(\text{merge}(\text{merge}(L_1, L_2), \text{merge}(L_3, L_4)) = (10 + 20) + (30 + 40) + \text{effort}(\text{merge}(L_1 + L_2, L_3 + L_4)) = 30 + 70 + (30 + 70) = 200$. On the other hand, $\text{effort}(\text{merge}(\text{merge}(\text{merge}(L_1, L_2), L_3), L_4) = (10 + 20) + \text{effort}(\text{merge}(\text{merge}(L_1 + L_2, L_3), L_4)) = 30 + (30 + 30) + \text{effort}(\text{merge}(L_1 + L_2 + L_3, L_4)) = 30 + 60 + (60 + 40) = 190$.

(b) Describe an $O(n \log n)$-time algorithm to compute a way of merging the input lists with minimum possible effort. The input consists only of the lengths $l_1, l_2, \ldots, l_n$ of the lists. Your algorithm should produce only an optimal merging strategy. It does not have to merge the lists. However, you should supply an optimality proof for your algorithm. **(10)**

Solution Obtain the Huffman tree on n symbols with weights $l_1, l_2, \ldots, l_n$. If d_i is the depth of the leaf l_i in a prefix tree with leaves $l_1, l_2, \ldots, l_n$, then Huffman's algorithm minimizes $\sum_{i=1}^{n} d_i l_i$. It suffices to note that this quantity is precisely the total effort of merging $L_1, L_2, \ldots, L_n$ according to the prefix tree.

(**Remark:** Strictly speaking, the above algorithm cannot run in $O(n \log n)$ time. Asymptotically, as n grows to infinity, the sum $\sum_{i=1}^{n} l_i$ too grows to infinity, even in the case when each l_i fits in a constant amount of storage (like 32-bit int variables). Working with the fractional relative weights $l_j / \sum_{i=1}^{n} l_i$ in place of l_j does not help much, since the precision of these floating-point weights needs to grow logarithmically in n. Nonetheless, the idea of this exercise was to avoid unnecessary complications about input complexity, and focus instead upon conceptual algorithmic developments. Incidentally, the same problem pertains to the Huffman algorithm to an exactly equal extent.)