COL 351 Aug 16, 17, 19

TUTORIAL SHEET 3

1. **(KT-Chapter 1)** Gale and Shapley published their paper on the stable marriage problem in 1962; but a version of their algorithm had already been in use for ten years by the National Resident Matching Program, for the problem of assigning medical residents to hospitals.

   Basically, the situation was the following. There were m hospitals, each with a certain number of available positions for hiring residents. There were n medical students graduating in a given year, each interested in joining one of the hospitals. Each hospital had a ranking of the students in order of preference, and each student had a ranking of the hospitals in order of preference. We will assume that there were more students graduating than there were slots available in the m hospitals. The interest, naturally, was in finding a way of assigning each student to at most one hospital, in such a way that all available positions in all hospitals were filled. (Since we are assuming a surplus of students, there would be some students who do not get assigned to any hospital.) We say that an assignment of students to hospitals is stable if neither of the following situations arises.

   - **First type of instability:** There are students s and s', and a hospital h, so that (i) s is assigned to h, (ii) s' is unassigned, and (iii) h prefers s' to s.
   - **Second type of instability:** There are students s and s', and hospitals h and h', so that (i) s is assigned to h and s' is assigned to h', (ii) h prefers s' to s, and s' prefers h to h'.

   So we basically have the stable marriage problem, except that (i) hospitals generally want more than one resident, and (ii) there is a surplus of medical students. Show that there is always a stable assignment of students to hospitals, and give an efficient algorithm to find one. The input size is $\theta(mn)$; ideally, you would like to find an algorithm with this running time.

   Solution Sketch: The algorithm is similar to the stable matching algorithm. The hospitals propose to the candidates in order of preference. A candidate, if not already assigned or assigned to a less preferable hospital, accepts the proposal; otherwise rejects it. There can be at most mn proposals, and so we are done. Check that all of the operations can be implemented using arrays.

   A common pitfall is that if candidates propose to hospitals, then hospitals need to check if they have a less preferable candidate. Since a hospital can have large number of positions, it is not clear how to perform this operation in constant time.

2. Consider the following algorithm for finding minimum spanning tree: sort all edges in *decreasing order* of weight. Let the edges be $e_1, \ldots, e_m$. Consider edges in this order, and initialize the set T to G, the entire graph. When we consider an edge e_i, we remove it from T if T contains a cycle containing e_i; otherwise we keep e_i. Prove that the final set T will be a minimum spanning tree (assume that G is connected).

   Solution Sketch: You can argue directly as in proof of Kruskal's algorithm, or prove that the tree constructed here will be exactly the one constructed by Kruskal's algorithm (when it looks at these edges in the reverse order). Let us try the latter approach. Suppose this algorithm discards an edge $e = (u, v)$. Let L_e be the set of edges appearing after e in the ordering above. Because the algorithm discards e, it follows that there is already a path from u to v in L_e. Now consider running Kruskal's algorithm. It will consider edges in L_e before looking at e. Note that that if x and y are in the same connected component in L_e, they will also be in the same connected component in Kruskal (why?). So u and v will be in the same connected component when the Kruskal's algorithm considers e. But then, it will not select e.

3. You want to throw a party and is deciding whom to call. You have n people to choose from, and you have made up a list of which pairs of these people know each other. You want to pick as many people as possible, subject to two constraints: at the party, each person should have at least five other people whom they know and five other people whom they don't know. Give 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. Give the running time in terms of n.

   Solution Sketch: Initialize a set S to the set of all n people. Repeat the following step as long as we are able to shrink the size of S – if there is a person x in S who knows less than 5 other persons *in* S or there are less than 5 people in S who do not know x, we remove x from S.

   In order to prove correctness, let $x_1, x_2, \ldots, x_k$ be the persons discarded by our algorithm (in this order). Now prove by induction on i that no feasible solution can invite $x_1, x_2, \ldots, x_i$.

4. Prove the following two properties of the Huffman encoding algorithm (assume that the sum of the frequencies of the characters is 1):(i) If some character occurs with frequency more than 2/5, then there is guaranteed to be a codeword of length 1, (ii) If all characters occur with frequency less than 1/3, then there is guaranteed to be no codeword of length 1.

   Solution Sketch: Let a be the character with the highest frequency, and suppose for the sake of contradiction that Huffman's algorithm assign length more than 1 to a. Consider the step in the algorithm when a gets merged with a (perhaps new) character X. At this time there must be another character Y in the current input (otherwise the final tree will have a as a child of the root). Since we are combining a and X at this step, the frequency of Y is at least that of a, and hence, Y cannot be a character in the original input (since a has the highest frequency). Thus, Y would have been obtained by merging two characters Y_1, Y_2. Now observe a few facts: (i) frequency of

Y must be more than that of a (because we are merging a and X at this step), and hence its frequency is more than $2/5$, (ii) Therefore, frequency of X is less than $1/5$, (iii) At least one of Y_1 and Y_2 has frequency more than $1/5$, say it is Y_1.

Now consider the step when we are merging Y_1 and Y_2. At this moment the frequency of X (may be X is not formed yet, in which case we consider any descendant of X which is a valid character at this step) is less than the frequency of Y_1 and so we shouldn't have merged Y_1 and Y_2.

The argument for the second one is simialr. Suppose all characters have frequency less than $1/3$ and suppose a character A in the original input gets code of length 1. Then A is merged with a character X at the last step of the Huffman coding procedure. Therefore the frequency of X must be more than $2/3$. Now X must have been obtained by combining two other characters, say X_1 and X_2, one of which has frequency more than $1/3$. But again this is not possible (why?).

5. Suppose you are given a text of length n^c where c is a constant and n is the number of distinct characters in the alphabet. Show that the Huffman tree for this text has height $O(\log n)$.

   Solution Sketch: We show the following fact. Suppose a, b, c are three nodes in the Huffman tree where a is the parent of b and b is the parent of c. We argue that the frequence of a is at least twice that of c. Indeed, let the other child of a be b' and similarly, let c' be the second child of b. So the algorithm merges c and c' into b. We argue that the frequency of b' must be at least that of c – otherwise we should not have merged c and c'. Since b is obtained by combining c and c', frequency of b is also at least that of c. Now when we combine b and b' into a, it follows that the frequency of a is at least twice that of c. Thus, we see that when we go two levels up from a node in the Huffman tree, the frequency doubles. It follows that the depth of any leaf node cannot be more than $2c \log_2 n$ (why?).