Advanced Algorithms, Fall 2012

Prof. Bernard Moret

Homework Assignment #5: Solutions

Question 1. (Greedy Algorithms)

You are given an undirected connected graph $G = (V,E)$ such that $|E| = |V| + 5$ (i.e., this is a very sparse graph). Each edge of G has a distinct cost. Design a linear-time algorithm to find the Minimum Spanning Tree of G, and prove the correctness of your algorithm.

Given that the graph has only 6 edges more than a spanning tree would have, it makes sense to think of choosing the edges to remove. Our proof of the correctness of the Prim/Kruskal algorithms showed that underlying both of these algorithms is the principle that, in the presence of cycle in the graph, one should remove the longest edge in the cycle. These 6 extra edges form cycles. So we simply look for cycles in the graph; once we find a cycle, we remove the longest edge on it, and go look for the next cycle. After removing 6 edges, we are left with a minimum spanning tree. Finding a cycle is a simple traversal of the graph and so run in $O(|E|) = O(|V|)$ time; removing the longest edge takes constant time. Repeating this step 6 times just multiplies by a constant, which disappears in asymptotic notation.

Question 2. (Greedy Algorithms)

Given n positive integers, $a_1, a_2, \cdots, a_n$, design an polynomial-time algorithm to determine whether there exists an undirected simple graph (self loops and multi-edges are not allowed) with n nodes such that the degrees of the n nodes correspond to the given n integers; prove the correctness of your algorithm.

Such sequences of values are known in the mathematical literature as *graphic sequences*, since they define a collection of graphs. Notice that the problem does not require that you construct a graph, just that the algorithm properly respond yes or no. Conceptually, of course, it is easier to think of the algorithm as actually building a graph.

The proper greedy approach to this problem is to place edges (which create degrees) and to do so starting with the vertices of highest degree, since those will require the most work and thus might be the hardest to connect correctly. We can think of picking the two largest values in the sequence and decreasing each by 1, corresponding to putting down an edge between the two vertices of highest degree. Another, even greedier, approach would be to pick the highest value, say a_1, then remove it entirely and decrement by 1 each of the a_1 next highest value, corresponding to setting up the vertex of highest degree with all of its edges. In either case, we repeat the process as long as valid choices remain: if we end up with all zero values, then we claim that there exists a graph whose node degrees match the given numbers. This seems intuitive, but it is not as obvious as it may seem, because we are likely to encounter ties, in which case the choice of "edges" will depend on a tiebreaker. Obviously, these ties do not matter when it comes to the values in the sequence, since those are interchangeable; so there is a difference between running our algorithm, which works on sequences, and actually building a graph, which must choose vertices. Thus we simply need to show that, in case of ties in the choice of vertices, there always exists a way to break the ties that will result in a valid graph; finding that choice algorithmically may well take more than polynomial time, but that is not our concern in a proof of existence. The result is in fact nearly trivial once we think mathematically

rather than algorithmically. We prove this by induction: if the reduced degree sequence is a graphic sequence, then it has a corresponding graph and we can simply add an edge between its two vertices of highest degree to construct a graph that matches the original degree sequence.

We now prove the converse statement: if a graph exists with these degree values, then our algorithm will answer yes. We prove this by induction on the algorithmic step: if there exists a graph G with node degrees equal to the a_i values and if a_1 and a_2 are the largest two values, then there also exists a graph G' with node degrees a_1-1, a_2-1, a_3, …, a_n. If there exists a pair of vertices in G of degree a_1 and a_2 linked by an edge, we are done: just remove that edge to obtain G'. If, on the other hand, such vertices cannot be found, let v_1 be a node in G of degree a_1 and v_2 a node of degree a_2, and let V_1, resp. V_2, be the sets of neighbors of v_1, resp. v_2. We are going to identify a node x_1 in V_1 and a node x_2 in V_1 that are not connected by an edge; we can then remove the pair of edges $\{v_1,x_1\}$ and $\{v_2,x_2\}$ and add the edge $\{x_1,x_2\}$ to produce a G' meeting the requirements. To show that x_1 and x_2 must exist, assume they did not; then every vertex of V_2 is connected to every vertex of V_1 (and vice versa), in addition to their connection to v_2 (or v_1); thus every vertex of V_2 has degree at least a_1+1, which contradicts our assumption that a_1 is the largest value.

Question 3. (Greedy Algorithms)

This problem addresses optimal file allocation for computer networks. You are given a completely connected network of nodes, a set of files to be allocated among the nodes, and a sequence of retrieval and updating requests (the entire sequence is known in advance). A retrieval or updating request is a triple, consisting of the node initiating the request, the file involved, and the number of bytes to be transferred.

An allocation scheme is an assignment of each file to one or more nodes. Having multiple copies of a file is advantageous in retrieval: the cost of a retrieval is zero when the file is held locally, but equals the number of bytes to be transferred when the file must be accessed remotely. On the other hand, multiple copies of a file increase the cost of an updating operation, because each copy must be updated and thus the number of bytes needed for updating is multiplied by the number of remote copies. Multiple copies are also desirable for reasons of reliability, but only up to a point, since relatively few nodes are expected to fail. The gain in reliability is a function of the number of copies and obeys the law of diminishing returns: each additional copy of a file gives a smaller gain than the previous copy. The cost of an allocation scheme is the cost of the given sequences of retrieval and updating requests minus the gain in reliability.

Develop a greedy algorithm that constructs the optimal allocation scheme and prove its correctness.

This problem is simpler than its statement would have us believe. First, note that the allocation problem can be solved separately and independently for each file, since the value of a solution is simply the sum of all costs and benefits and the presence or absence of a file at one node does not affect the costs of retrieval or update of another file at the same node or any other node. Next, since the allocation is static, the order in which the retrievals and updates are made is irrelevant: all the matters is the total amount of bytes retrieved and the total updated from each location. So we can restate the problem as follows, for just one file: given n locations, and given, for each location i, a number r_i of bytes to be retrieved from that location, and a number u_i of bytes to be updated from that location, choose the value (0 or 1) of variables x_i (one for each location) such that

$$\sum_i r_i \cdot (1-x_i) + \sum_i u_i \cdot (c-x_i) - b(c-1)$$

is minimized, where $c = \sum_i x_i$ and $b(k)$ is the reliability bonus for k (extra) copies, with $b(1) = 0$ and $b(j) \geq 0$ and, for all $j > 1$, $b(j+2) + b(j) \leq 2b(j+1)$ (law of diminishing returns). We can rewrite the objective:

$$\sum_i r_i + c\sum_i u_i - b(c-1) - \sum_i (r_i+u_i)x_i$$

The first term is a constant and can be dropped; the second term is c times a constant, so, like the third term, it depends only on c, the number of copies; finally, the last term shows the number of bytes accessed and updated locally. Obviously, it makes sense to place the file at the node with the largest $r_i + u_i$ sum; whether to place additional copies simply depends on whether the decrease $r_j + u_j$, combined with the decrease due to increased reliability, is larger than the increase $sum_i u_i$ in update costs for the new copy.

Here then is our greedy algorithm:

> Compute the $vol(i) = r_i + u_i$ sums and sort them in decreasing order by value—that is, $vol(1)$ is the largest, $vol(2)$ the second largest, etc. Compute the value $U = \sum_i u_i$. Set $j = 2$. While $vol(j) + b(j) - b(j-1) > U$, place a copy of the file at the node corresponding to index j (reindexed through the sorting).

Optimality follows immediately from our assumption of diminishing returns for $b(j)$, the sorting on $vol(\,)$, and the fact that U is a constant.

Question 4. (Matching)

We are given a $n \times n$ 0-1 matrix M. We can swap two columns of M, or two rows of M. We say that M can be diagonalized if we can perform a sequence of (row and/or column) swaps to make all the diagonal items of M equal to 1. Design an algorithm to decide whether a given binary matrix M can be diagonalized and prove the correctness of your algorithm.

Note that M must have at least one 1 on each row and on each column to make a solution possible. We do not care what happens to any matrix entry in the swaps except for the n entries of value 1 that end up on the diagonal. Note also that we cannot make use of more than one 1 per row or per column; thus the diagonal elements must come from positions that share neither row nor column indices. This condition is very similar to the requirements for a matching: share neither one nor the other endpoints.

Hence we can solve our problem using the following transformation: given the matrix M, build a bipartite graph with n vertices on each side and an edge $\{v_i, v_j\}$ whenever we have $M_{ij} = 1$. Then M can be diagonalized if and only if the maximum matching in the graph has n edges. If the graph has a matching of size n, then an edge $\{v_i, v_j\}$ in the matching corresponds to a matrix entry $M_{i,j}$ that ends up on the diagonal; by the definition of matchings, this matrix entry shares neither row nor column indices with other matrix entries that end up on the diagonal. Conversely, if M can be diagonalized, then it had n entries that shared neither row nor column entries, so that the corresponding edges form a matching of n edges.