15-451 Algorithms, Fall 2004

Homework # 6 **due: Mon–Tues, Nov 22–23, 2004**

Reminder:

- This is an oral presentation assignment. You should work in groups of three. At some point before Sat Nov 20 at midnight you should sign up for a 1 hour time slot on the signup sheet on the course web page. *Note:* If you prefer to sign up for an earlier date (e.g., because you will be going out of town) then feel free to contact your TA and arrange a time.

Problems:

1. [Euler tours] Given a graph G, the *Euler Tour problem* is to find a tour (a path that ends back where it started) that traverses each edge exactly once. It may visit some vertices multiple times — i.e., it doesn't have to be a *simple* cycle. In this problem you will give an efficient algorithm to find an Euler tour if one exists. We will assume for this problem that G is undirected.

   (a) Suppose the graph has some node of odd degree. Then there cannot be an Euler tour. Why?

   (b) On the other hand, if all nodes have even degree (and the graph is connected) then there always does exist an Euler tour. Prove this by giving a polynomial-time algorithm that finds an Euler tour in any such graph. Your algorithm should work for multigraphs too (multiple edges allowed between any two vertices).

   Hint: Suppose you start at some node x and just arbitrarily take a walk around the graph, never going on any edge you've traversed before. Where will you end up? Now, what about parts of the graph you haven't visited? This problem is given as problem 22-3 in the book.

   Note: it is interesting that the very similar sounding *Hamiltonian Cycle* problem, which is to find a tour that visits each *vertex* exactly once, is NP-complete.

2. [TSP approximation] Given a weighted undirected graph G, a *traveling salesman tour* for G is the shortest tour that starts at some node, visits all the vertices of G, and then returns to the start. We will allow the tour to visit vertices multiple times (so, our goal is the shortest cycle, not the shortest simple cycle). This version of the TSP that allows vertices to be visited multiple times is sometimes called the *metric* TSP problem, because we can think of there being an implicit complete graph H defined over the nodes of G, where the length of edge (u, v) in H is the length of the shortest path between u and v in G. (By construction, edge lengths in H satisfy the triangle inequality, so H is a metric. We're assuming that all edge weights in G are positive.)

(a) Briefly (just a few sentences): prove that we can get a factor of 2 approximation to the TSP by finding a minimum spanning tree T for H and then performing a depth-first traversal of T. (If you get stuck, the book does this in a lot more sentences in section 35.2.1.)

(b) The minimum spanning tree T must have an even number of nodes of odd degree (only considering the edges in T). In fact, *any* (undirected) graph must have an even number of nodes of odd degree. Why?

(c) Let M be a minimum-cost perfect matching (in H) between the nodes of odd degree in T. I.e., if there are $2k$ nodes of odd degree in T, then M will consist of k edges no two of which share an endpoint. Prove that the total length of edges in M is at most one-half the length of the optimal TSP tour.[1]

(d) Combine the above facts with your algorithm from 1(b) to get a 1.5 approximation to the TSP. Hint: think about the (multi)graph you get from the union of edges in T and M.

The above algorithm is due to Christofides [1976]. Extra credit and PhD thesis: Find an algorithm that approximates the TSP to a factor of 1.49.

3. [The List-Update Problem] Suppose we have n data items $x_1, x_2, \ldots, x_n$ that we wish to store in a linked list in some order. Let's say the cost for performing a $lookup(x)$ operation is \$1 if x is in the head of the list, \$2 if x is the second element in the list, and so on.

For instance, say there are 4 items and it turns out that we end up accessing x_1 3 times, x_2 5 times, x_3 once, and x_4 twice. In this case, in hindsight, the best ordering for a linked list would have been (x_2, x_1, x_4, x_3) with a total cost of \$21.

The *Move-to-Front* (MTF) strategy is the following algorithm for organizing the list if we don't know in advance how many times we will access each element. We begin with the elements in their initial order $(x_1, x_2, \ldots, x_n)$. Then, whenever we perform a $lookup(x)$ operation, we move the item accessed to the front of the list. Let us say that performing the movement is free. For instance, if the first operation was $lookup(x_3)$, then we pay \$3, and afterwards the list will look like $(x_3, x_1, x_2, x_4 \ldots)$.

(a) Suppose $n = 4$ and we use MTF starting from the order (x_1, x_2, x_3, x_4). If we perform the following 4 operations:

$$lookup(x_4), lookup(x_2), lookup(x_4), lookup(x_2).$$

What does the list look like in the end and what was the total cost?

(b) Your job is to prove that the total cost of the MTF algorithm on a sequence of m operations (think of m as much larger than n) is at most $2C_{static} + n^2$ where C_{static} is the cost of the best static list in hindsight for those m operations (like in our first example). We will prove this in two steps.

[1] We didn't prove it in class, but there are efficient algorithms for finding minimum cost perfect matchings in *arbitrary* graphs (not just bipartite graphs).

i. First prove the somewhat easier statement that the cost of Move-to-Front is at most $2C_{initial}$ where $C_{initial}$ is the cost of the original ordering $(x_1, x_2, \ldots, x_n)$.

   Hint: If $i < j$ but x_j is in front of x_i in the MTF list, let's say that x_j has "cut in line" in front of x_i. Now, imagine that each element x_i has a piggy bank with \$1 for everyone that is currently cutting in line in front of it.

ii. Now prove the $2C_{static} + n^2$ bound.