CMPT 706 — Algorithms for Big Data

Outline Solutions to Exercises on Greedy Algorithms.

1. **Some of your friends have gotten into the burgeoning field of time series data mining, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges — what is being bought — are one source of data with a natural ordering in time. Given a long sequence S of such events, your friends want an efficient way to detect certain "patterns" in them — for example, they want to know if the four events**

   buy Yahoo, buy eBay, buy Yahoo, buy Oracle

   occur in this sequence S, in order but not necessarily consecutively.

   They begin with a collection of possible events (e.g., the possible transactions) and a sequence S of n of these events. A given event may occur multiple times in S (e.g., Yahoo stock may be bought many times in a single sequence S). We will say that a sequence S' is a subsequence of S if there is a way to delete certain of the events from S so that the remaining events, in order, are equal to the sequence S'. So, for example, the sequence of four events above is a subsequence of the sequence

   buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo, buy Oracle

   Their goal is to be able to dream up a short sequences and quickly detect whether they are subsequences of S. So this is the problem they pose to you: Give an algorithm that takes two sequences of events — S' of length m and S of length n, each possibly containing an event more than once — and decides in time $O(m+n)$ whether S' is a subsequence of S.

   Let the sequence S consists of $s_1, \ldots, s_n$ and the sequence S' consists of $s'_1, \ldots, s'_m$. We give a greedy algorithm that finds the first event in S that is the same as s'_1, matches these two events, then finds the first event after this that is the same as s'_2 and so on. We will use $k_1, k_2, \ldots$ to denote the match we have found so far, i to denote the current position in S, and j the current position in S'.

```
initially i = j := 1
while i ≤ n and j ≤ m
   if s_i is the same as s'_j then
        set k_j := i
        set i := i + 1 and j := j + 1
   else set i := i + 1
endwhile
if j = m + 1 return the subsequence found:  k_1, ..., k_m
else return ``S' is not a subsequence of S''
```

   The running time is $O(n)$: one iteration through the while loop takes $O(1)$ time, and each iteration increments i, so there can be at most n iterations.

   It is also clear that the algorithm a correct match if it finds anything. It is harder to show that if the algorithm fails to find a match, then no match exists. Assume that S' is the same as the subsequence $s_{\ell_1}, \ldots, s_{\ell_m}$ of S. We prove by induction that the algorithm will succeed in finding a match and will have $k_j \leq \ell_j$ for all $j = 1. \ldots, m$. This is analogous to to the proof in class that Dijkstra's algorithm finds the shortest paths: we prove that the greedy algorithm is always ahead.

   Claim. For each $j = 1, \ldots, m$ the algorithm finds a match k_j and $k_j \leq \ell_j$.

   Proof. The proof is by induction on j. First consider $j = 1$. The algorithm lets k_1 be the first event that is the same as s'_1, so we must have $k_1 \leq \ell_1$.

Now consider a case when $j > 1$. Assume that $j - 1 < m$ and assume by the induction hypothesis that the algorithm found the match k_{j-1} such that $k_{j-1} \leq \ell_{j-1}$. The algorithm lets k_j be the first event after k_{j-1} that is the same as s'_j if such an event exists. We know that ℓ_j is such an event and $\ell_j > \ell_{j-1} \geq k_{j-1}$. So $s_{\ell_j} > k_{j-1}$. The algorithm finds the first such index, so we get that $k_j \leq \ell_j$.

2. **Let $G = (V, E)$ be an undirected graph with costs $c_e \geq 0$ on the edges $e \in E$. Assume you are given a minimum cost spanning tree T of G. Now assume that a new edge is added to G, connecting two nodes $v, w \in V$ with cost c.**

   (a) Give an efficient algorithm to test if T remains the minimum cost spanning tree with the new edge added to G (but not to the tree T). Make your algorithm run in time $O(|E|)$. Can you do it in $O(|V|)$ time.

   (b) Suppose T is no longer a minimum cost spanning tree. Give a linear time algorithm (time $O(|E|)$ to update the tree T to a new minimum cost spanning tree.

   (a) Let $e = (v, w)$ be the new edge being added. We represent T using an adjacency list, and we find the $v - w$ path P in T in time linear in the number vertices and edges of T, which is $O(|V|)$. If every edge on this path in T has cost less than c, then the Cut Property (or the Cycle Property from the next problem) implies that the new edge $e = (v, w)$ is not in the minimum spanning tree, since it is the most expensive edge in the cycle C formed from P and e, therefore, it is not the cheapest edge in the cut separating v from the rest of the graph. So, the minimum spanning tree has not changed. On the other hand, if some edge e' on this path has cost greater than c, then e is a cheaper edge in the cut separating the fragment of C between e and e' from the rest of the graph. So, T is no longer the minimum spanning tree.

   For (b), we replace the heaviest edge on the $v - w$ path P in T with the edge $e = (v, w)$, obtaining a new spanning tree T'. We claim that T' is a minimum spanning tree. To prove this, we consider any edge e' not in T', and show that we can apply the Cut Property to conclude that e' is not in any minimum spanning tree. So let $e' = (v', w')$. Adding e' to T' gives us a cycle C' consisting of the $v' - w'$ path P' in T', plus e'. If we can show e' is the most expensive edge on C', we are done.

   To do this, we consider one further cycle: the cycle K formed by adding e' to T. By the Cut Property, e' is the most expensive edge on K. So now there are three cycles to think about: C, C', and K. Edge f is the most expensive edge on C, and edge e' is the most expensive edge on K. Now, if the new edge e does not belong to C', then $C' = K$, and so e' is the most expesive edge on C'. Otherwise, the cycle K includes f (since C' needed to use e instead), and C' uses a portion of C (including e) and a portion of K. In this case, e' is more expensive than f (since f lies on K), and hence it is more expensive than everything on C (since f is the most expensive edge on C). It is also more expensive than everything else on K, and so it is the most expensive edge on C', as desired.

   Now, if we want to consider the case when the edge costs are not all distinct, we apply the approach in the class: we first perturb all edge costs by extremely small amounts so they become distinct. Moreover, we do this so we add a very small quantityt ε to the new edge e, and we perturb other edges f by even much smaller, distinct, quantities δ_f.For a tree T, let $c(T)$ denote its real (original) cost, and let $c'(T)$ denote the perturbed cost.

   Now we use the above solution with distinct edge cost. Our perturbationhas the following two properties.

   (i) First, for trees T_1 and T_2, if $c'(T_2) < c'(T_1)$, then $c(T_2) \leq c(T_1)$.

   (ii) Second, if $c(T_1) = c(T_2)$, and T_2 contains c but T_1 does not, then $c'(T_2) > c'(T_1)$.

   It follows from these two properties that our conclusion in (a) is correct: since $c'(T') < c'(T)$, and T' contains e but T does not, property (i) implies $c(T') \leq c(T)$, and then property (ii) implies $c(T') < c(T)$. Now, in (b), we compute a minimum spanning tree with respect to the perturbed costs which, by property (i), is also one of (possibly several) minimum spanning trees with respect to the real costs.

3. **One of the basic motivations bihind the Minimum Spanning Tree problem is the goal of disigning a spanning network for a set of nodes with minimum total cost. Here we explore another type of objective: designing a spanning network for which the most expensive edge is as cheap as possible.**

Specifically, let $G = (V, E)$ be a connected graph with n vertices, m edges, and positive edge costs that you may assume are all distinct. Let $T = (V, E')$ be a spanning tree of G; we define the *bottleneck edge* of T to be the edge of T with the greatest cost.

A spanning tree is a minimum-bottleneck spanning tree if there is no spanning tree T' of G with a cheaper bottleneck edge.

(a) Is every minimum-bottleneck tree of G a minimum spanning tree of G? Prove or give a counterexample.

(b) Is every minimum spanning tree of G a minimum-bottleneck tree of G? Prove or give a counterexample.

(a) This is false. Let G have vertices $\{v_1, v_2, v_3, v_4\}$, with edges between each pair of vertices, and with the weight on the edge from v_i to v_j equal to $i + j$. Then every tree has a bottleneck edge of weight at least 5, so the tree consisting of a path through vertices v_3, v_2, v_1, v_4 is a minimum bottleneck tree. It is not a minimum spanning tree, however, since its total weight is greater than that of the tree with edges from v_1 to every other vertex.

(b) This is true. Suppose that T is a minimum spanning tree of G, and T' is a spanning tree with lighter bottleneck edge. Thus, T contains an edge e that is heavier than every edge in T'. So if we add e to T', it forms a cycle C on which it is the heaviest edge (since all other edges in C belong to T'). By the Cut Property, then, e does not belong to any minimum spanning tree, contradicting the fact that it is in T and T is a minimum spanning tree.

3. **Give a linear time algorithm that takes as input a tree and determines whether it has a perfect matching: a set of edges that touches each vertex exactly once.**

   We will use the following obvious property: a leaf of a tree can only be matched with its parent. The idea of the algorithm is then clear, repeatedly perform three steps: find a leaf match with its parent, remove them from the graph, continue. As a graph traversal algorithm we use DFS, and assume that the given tree is directed from the root to leaves. Let r be the root.

```
TreeMatching(G,v)
   if outdeg(v) = 0, return 'LEAF'
   for each (v,w) ∈ E do
        if TreeMatching(G,w) = 'LEAF' then
           if matched(v) = 1 then output 'no matching'
           print (v,w)
           matched(v) = 1
           remove v,w from G
   endfor
```

   The process is initialized by

```
TreeMatchingInit(G)
   for each v ∈ V set matched(v) = 0
   if TreeMatching(G,r) = 'LEAF' then output 'no matching'
```

   Note that since G is a tree, DFS never revisits a vertex, therefore we do not need to mark visited vertices. Also, if two leaves attached to a vertex, then a perfect matching does not exist, as there is no way to match these leaves. This condition is checked using the $matched(v)$ value.

   The running time of this algorithm is essentially that of BFS, that is, $O(m + n)$, where n is the number of vertices, and m is the number of edges. However, as G is a tree, $m = n - 1$.

4. ***Ternary Huffman.*** **Trimedia Disks Inc. has developed 'ternary' hard disks. Each cell on a disk can now store values 0,1, or 2 (instead of just 0 or 1). To take advantage of this new technology, provide a modified Huffman algorithm for compressing sequences of characters from an alphabet of size n, where the characters occur with known frequencies $f_1, f_2, \ldots, f_n$. Your algorithm should encode each character with a variable-length codeword over the values 0,1,2 such that no codeword is a prefix of another codeword and so as to obtain the maximum possible compression. Prove that your algorithm is correct.**

   In this problem we need to find a collection of codewords $c_1, c_2, \ldots, c_n$ over 0,1,2 (let us call these 3 values tribits) such that (a) these codewords are prefix free, and (b) they minimize the function $\sum_{i=1}^{n} f_i|c_i|$. As in the class, condition (a) can be enforced by modelling the codewords as paths in a tree, only this time every vertex may have 0,2, or 3 children. If the corresponding tree is $T = (V, E)$, where the symbols are the leaves, then the function above can be replaced with $\sum_{v \in V} w(v)$, where $w(v) = f_i$ is v is the leaf corresponding to the i'th symbol, and $w(v) = w(v_1) + w(v_2) + w(v_3)$ otherwise, v_1, v_2, v_3 being the children of v. The only difference with the binary case is that when n is even, one of the internal vertices has degree 2. Obviously, this vertex must be in very bottom of the tree.

   The algorithm then is as follows:

```
TriHuffman(f_1, f_2, ..., f_n)
   create a list H of leaves, i.e.  the alphabet symbols
   while H contains at least 3 elements do
        if H contains even number of elements, then
             pick the elements x,y from H of minimal weight
             remove x,y from H
             create new element v, assign weight w(v) = w(x) + w(y) to it,
                  and include it to H
        otherwise
             pick the elements x,y,z from H of minimal weight
             remove x,y,z from H
             create new element v, assign weight w(v) = w(x) + w(y) + w(z) to it,
                  and include it to H
   endfor
```

   The correctness of the algorithm is proved by the same exchange argument as was used in the class.