CIT 596 Online **Summer 2020**

Module 7–8

Name: Solutions

Questions requiring written answers. Note that greedy algorithms are generally simple. It can be surprising that such simple algorithms can actually work for seemingly complicated problems. As a result of this, more emphasis will be placed on a rigorous proof of correctness when evaluating your solutions than usual; less credit will be apportioned to the algorithm itself.

Greedy

1. The *fractional knapsack problem* is stated as follows. You are given a knapsack that can carry a weight of at most W. You are also given a collection of n items, each with an associated weight and value (both positive). Given that you are allowed to take a fraction of an item (with corresponding proportional value), design an algorithm determines which items to take (along with the weight of each item taken) that maximizes the value of the knapsack.

   Solution:

   Algorithm: Sort the collection of items in decreasing order of value per weight. Iterate through the items, and while the next item doesn't overload the knapsack, take the full weight of the current item. On the last item, take as much as possible without overflowing the knapsack.

   Correctness: Assume for contradiction that there is a better choice of items compared to our algorithm, resulting in total value V^{OPT}. Consider the collection of items in non-increasing order of value per weight. Our algorithm chooses items with higher value per weight. Since the claimed optimal solution is distinct from our algorithm's choice, there must be some amount of some item that our algorithm chose that the optimal solution did not choose, and in place, the optimal solution chose some item with strictly lower value per weight. Let the item our algorithm chose that the optimal solution did not choose be item x of weight w. Any optimal solution clearly exhausts the entire capacity of the knapsack, so there is some other item or combination of items the optimal solution chose with total weight w in place of x. But then consider swapping these items for x in the optimal solution. Since the value per weight of item x is strictly higher, the new solution has total value at least as high as before. Repeating this procedure, we can see that in fact our algorithm's solution is at least as good as any optimal solution, a contradiction.

   Running Time: Sorting the list of items by their value per weight takes $O(n \log n)$ time. Then, iterating over the items takes $O(n)$ time. Thus, the total runtime is $O(n \log n)$.

2. The Huffman encoding algorithm for generating optimal binary codes for an alphabet can easily be extended to generating any k-ary codes. (In a k-ary code each alphabet symbol is encoded as a string of 'digits' where each digit can take on values $0, 1, 2, \ldots, k-1$. Consider how this would be done and what minor modifications are necessary in the proof of correctness. Try to think about small examples to help you design to a general algorithm.

   Solution:

   Algorithm: The algorithm follows the same pattern as the standard Huffman encoding algorithm. However, in order for k-ary codes to be correctly implemented, we require a preprocessing step to ensure the resulting Huffman tree is full, save perhaps the lowest levels. This is done by adding dummy symbols with frequency 0 until the total number of nodes is such that the process of iteratively subtracting k and adding 1 eventually results in a single node (these numbers of course refer to the step in the algorithm in which nodes are combined by taking the k lowest frequency nodes and combining them into a single node with k children). As with the original algorithm, this is done until only one node remains, at which point we have the optimal tree.

Correctness: The proof of correctness also follows the same pattern as the proof of correctness for the standard Huffman encoding algorithm. First, we argue that making the tree full is optimal. This is easy to see: if an optimal tree were not full, we could convert it into a full tree that is at least as good, by simply moving nodes on lower levels up to the non-full layers. It then remains to show why the decision to combine the lowest frequency nodes is an appropriate greedy choice. Again, we use an exchange argument. If the k siblings at the bottom of the tree are not the k lowest-frequency nodes, then we exchange one of the k lowest frequency nodes that is not in this bottom layer, with a node in this bottom layer that is not among the k lowest frequency nodes. The result is a tree that is at least as good.

The key point to note is that in a k-ary tree it is not always possible to have every internal node have k children (This is always possible in a binary tree.) But, we can always find a k-ary tree where at most one internal node has fewer than k children and all the others have k. The next point is, where to put this internal node with fewer than k children. It is easy to see by the exchange argument above that this internal node should be at the deepest possible level. This might give you some intuition for why the algorithm works the way it does.

3. You are given a long, straight road with n houses located at points $x_1, x_2, \ldots, x_n$. The goal is to set up cell towers at points along the road such that each house is in range of at least one tower. If each tower has a range (radius) of r, give an algorithm to determine the minimum number of towers needed to cover all houses and the locations of these towers.

   Solution:

   Algorithm: Sort the list of given locations. While not every house has been covered, consider the first uncovered house with location x_i. Place a cell tower at location $x_i + r$ and mark all houses in range of the new cell tower as covered. Return the total number of cell towers placed.

   Correctness: Let the placement of towers generated by our algorithm be S and consider some optimal placement of cell towers $\mathcal{O}$, both in increasing order of tower position. Consider the first instance where S diverges from $\mathcal{O}$, where S placed a tower at x but $\mathcal{O}$ placed a tower at x' (that is, all previous houses/towers correspond exactly between S and $\mathcal{O}$. Let the house our algorithm chose to cover be house i. We know i is located at position $x_i - r$. Since $\mathcal{O}$ must also somehow cover i but we assume it placed a tower differently that S, it must have placed the tower at some $x' < x_i - r$ (if it were placed at $x' > x_i$, then i would not be covered). Now, in $\mathcal{O}$, consider moving the tower from x' to x, producing a new solution $\mathcal{O}'$. No less houses are covered due to our assumption that x/x' is the first divergence between S and $\mathcal{O}$, and the solution is as optimal as it was before (in terms of number of cell towers required). Thus, this procedure can be repeated for any remaining divergences in S and $\mathcal{O}'$ until we arrive at some optimal solution with the same cell tower locations as S. Thus, S is optimal.

   Running Time: If the input locations are not assumed to be sorted, sorting them can be done in $O(n \log n)$ time. Then, cell towers are placed in a linear fashion, requiring $O(n)$ time. The total runtime is $O(n \log n)$.

4. There are n jobs that need to be processed by computers. For each i, job i must first run for s_i minutes on a centralized computer before running for an additional t_i minutes on a personal computer (each job has its own personal computer). Design an algorithm to schedule the order in which jobs should be submitted to the centralized computer so as to minimize the time in which all jobs are completely done processing.

   Solution:

   Algorithm: Run the jobs on the centralized computer consecutively in decreasing order of t_i.

   Correctness: First note that there exists an optimal scheduling of jobs without idle time on the centralized computer, justifying our consecutive scheduling of jobs. This is because any scheduling of jobs with idle time could have all later jobs pushed forward by the amount of idle time, resulting in a schedule with finish time no greater than before.

Now, we define an *inversion* of jobs as a pair of jobs i and j such that $t_i > t_j$ but i is scheduled immediately after j. Observe that our greedy algorithm produces a schedule with no idle time and no inversions. We now prove that any such solution is optimal.

Let $\mathcal{O}$ be an optimal schedule with no idle time. Suppose it has an inversion between jobs i and j. Consider creating a new schedule $\mathcal{O}'$ by swapping the ordering of i and j. Note that since this swap is between consecutively scheduled jobs, the finish times of all other jobs remain unaffected. In $\mathcal{O}$, we know that i finishes after j (since i starts later and additionally has a longer individual processing time). Thus, after the swap, it suffices to show that the finishing time of both i and j (in $\mathcal{O}'$) are no more than the finish time of i in $\mathcal{O}$. Since $\mathcal{O}$ and $\mathcal{O}'$ are identical up to the point that i and j are scheduled, we can ignore the common time elapsed up to this point and treat the start of the first of these two jobs as time 0.

The finish time of i in $\mathcal{O}'$ is

$$F_i = s_i + t_i$$

and the finish time of j is

$$F_j = s_i + s_j + t_j$$

Meanwhile, the finish time of i in $\mathcal{O}$ was

$$\begin{aligned} s_j + s_i + t_i &\geq s_i + t_i \\ &= F_i \end{aligned}$$

and also observe

$$\begin{aligned} s_j + s_i + t_i &= s_i + s_j + t_i \\ &\geq s_i + s_j + t_j && (t_i > t_j) \\ &= F_j \end{aligned}$$

Thus, whether i or j finished first in $\mathcal{O}$, fixing the i and j inversion never results in a longer finishing time. Note that we can keep repeating this procedure until there are no more inversions, which we know is possible since one can sort a list of numbers with a finite number of pairwise swaps.

We have thus proved that schedules without idle time or inversions are optimal. Our algorithm produces such a schedule, and is thus optimal.

Running Time: Computing order of the jobs simply requires one sort. The algorithm runs in $O(n \log n)$ time.

5. Let $G = (V, E, w)$ be a directed graph, where w is a weight function that assigns a positive weight to each edge. Let s be a node in G. Suppose we are given a program P that claims to implement Dijkstra's algorithm. For each vertex v in the graph this program produces $v.\pi$ which is supposedly the vertex that precedes v on the shortest path from s to v. It also produces $v.d$, which is supposedly the distance from s to v. Design and analyze an efficient algorithm that takes G and P's output as its inputs and checks if P's output is correct. Your algorithm should obviously be faster than rerunning Dijkstra's algorithm on G. For simplicity (though not necessary) you may assume that all vertices are reachable from s.

   Solution:

   Algorithm: We simply iterate through all the edges and verify that each edge is relaxed. In particular, we verify that $v.d \leq u.d + w(u, v)$ for $u, v \in V$, $(u, v) \in E$. This verifies P is a shortest path tree. We should also verify that this tree is consistent, i.e. that P is a valid tree, that P is rooted at S, and that $v.d = v.\pi.d + w(v.\pi, v)$.

Correctness: The basic idea is that this algorithm depends on the principle of relaxation. If P is a shortest path tree, then clearly no edges can be relaxed, so P will pass. If P is not a shortest path tree, then we can use the optimal substructure property of shortest paths to show there must exist some $e = (u, v)$ such that taking the shortest path to u then using e is better than the computed $v.d$. Thus, we will find this edge and output that P is not a shortest path tree.

Running Time: The algorithm performs a constant number of operations (checks) for each edge in the graph. Thus the runtime is $O(n + m)$.

6. Suppose a graph G has two minimum spanning trees. Prove that in any cycle formed by the union of the edges in the two trees, at least two of the edges have the same weight. Is it also true that this duplicated weight is the largest weight on the cycle? Prove or give a counter-example.

   Solution:

   We can prove that there must be a repeated edge and that duplicated edge weight must have the largest weight on the cycle all at once. Suppose there was not a duplicated edge weight or the duplicated edge weight was not the maximum weight. In any case, the largest weight edge in the cycle belongs to one (or both) of the trees, say T_1. Remove this edge from T_1 and consider the cut induced by the remaining edges in T_1. There is at least one edge from the cycle crossing this cut, which is of lower weight than the edge we removed. By the cut property, this contradicts the fact that T_1 was a valid MST.

7. You have to cut a log of wood into several pieces at specified points along the log. To do so, you bring it to a company: For each cut, the company charges money equal to the length of the piece being cut. Their cutting method allows only one cut to be made at a time.

   For example, suppose that a log of length 10 meters has to be cut at positions 2, 4, and 7 from the left end. There are several ways to do this. One way is to first cut at 2, then 4, and then at 7. This gives a price of $10 + 8 + 6 = 24$ because the first piece was 10 meters in length, the second was 8 meters, and the third was 6 meters in length. Another way is to cut at 4, then at 2, then at 7. This would give a total price of $10 + 4 + 6 = 20$, which is better.

   Not all problem have greedy algorithms; when they do exist they are simpler to design but harder to prove correct since you have to prove that the first decision the algorithm makes is the right one.

   One could consider several different greedy algorithms for this log cutting problem.

   (a) Provide a counterexample to show that a greedy algorithm that always makes the median cut first is not correct. (If we require n cuts on a piece of log, the median cut is the $\lceil \frac{n}{2} \rceil$th cut, where ever it is located.)

   (b) Provide a counterexample to show that a greedy algorithm that always makes the cut closest to the center of the log first is not correct.

   (c) Propose another simple, greedy way of choosing the first cut, and show that that does not work either.

   Solution:

   (a) Imagine a log of length 20 that needs to be cut at positions 4, 5, and 6 from the left end. The median cut is at position 5 and costs 20. It divides the log into pieces of length 5 and 15, each requiring one more cut of cost 5 and 15 respectively. Thus the total cost is 40.

   On the other hand, if we make the first cut at 6 (for a cost of 20), the second cut at 4 for a cost of 6, and the final cut at 5 for a cost of 2, we get a total cost of 28, which is better than the previous solution. Thus making the median cut is not optimal.

   (b) Imagine a log of length 10 with cuts needed at positions 4, 5, and 7 from the left end. The greedy algorithm would cut at 5 first, for a cost of 10. It would have two pieces of length 5 each requiring one more cut, costing 5 each for a total cost of 20.

But it is better to cut first at 4 for a cost of 10, then at 7, for a cost of 6, and finally at 5 for a cost of 3, leading to a total cost of 19.

(c) This is an open-ended question. Here are some examples: (i) always make the leftmost cut on a log, (ii) always make the most extreme cut, (iii) always cut out the longest piece (choosing which end of this piece to cut in some order). For each one you should be able to find an easy counter-example.