CSOR W4231: Practice Midterm Exam 2(Fall'19) Solutions

Problem 1 Assume we have n miners and that each miner is assigned to one of m gold mines. Each mine will produce a different profit depending on the number of miners assigned to it. Let the profit of mine i with j miners assigned to it be $p_{i,j}$.

a) Describe an algorithm that produces an assignment of the miners to the m mines that produces the most profit. (Hint: consider a dynamic programming solution which uses the quantity $P[i, j]$, denoting the profit of the optimal assignment of the first j miners to the first i mines. Make sure to state what is your recurrence.)

Solution: As given in the hint, we let $P[i, j]$ be the cost of the optimal assignment of the first j miners to the first i mines. Thus, we have that:

$$P[i+1, j] = \max_{k \le j} \left(P[i, k] + p_{i+1}^{j-k} \right),$$

since the assignment to the $i+1$th mine will contribute some miners, and the rest will be assigned to the mines before $i+1$. In addition, we have

$$P[i, 0] = \sum_{i' \le i} p_i^0,$$

since that is the profit obtained from no miners on mines up to i. Thus, in order to compute $P[m, n]$, we must compute every value of the table, and each takes time n. Thus, the total runtime is $O(n^2 m)$.

b) Can we improve the runtime if we are guaranteed that the *law of diminishing returns* holds? In particular, assume that for each mine, each additional miner assigned to it can't result in a larger increase in profits than the previous miner? (As an example, if we have $p_{1,1} = 4$, $p_{1,2} = 6$, and $p_{1,3} = 7$, then the law of diminishing returns holds since the first miner adds a profit of 4 while the second and third add profits of 2 and 1 respectively). Design a greedy algorithm for this problem.

Solution: We first compute the $m \times n$ matrix $\Delta[i, j] = p_i^j - p_i^{j-1}$, i.e., we compute how much the jth miner would add if it were assigned to the ith mine. We then proceed greedily, where we maintain the count of how many miners are assigned to which mines, and we assign the miner to the value which maximizes $\Delta[i, \alpha_i + 1]$, were α_i is the number of miners currently assigned to mine i.

We note that this algorithm requires time $O(mn)$ in order to compute Δ, and time $O(mn)$ to assign miners.

Problem 2 Suppose you are given n positive integers $\{a_1, \dots a_n\}$, and a positive integer bound B. The goal is to take a subset of the n integers whose sum is as close as possible to B without exceeding it. Consider the greedy strategy that first chooses the largest integer $a_i \leq B$, then subtracts a_i from the bound B and repeats with the remaining integers and the new bound $B' = B - a_i$. Either prove this strategy is optimal or give a counterexample where it performs sub-optimally.

Solution: This strategy is suboptimal. Consider the following counterexample, where $B = 5$ and our set is $\{2, 3, 4\}$. Per the greedy strategy, we would find the subset $S = \{4\}$ and no further elements could be added to this set, yielding an optimality gap of 1 (where optimality gap is defined as $B - \sum_{i \in S} i$). However, under the optimal strategy, we can find the subset $\{2, 3\}$, whose optimality gap is 0. Note this problem should remind you of a simpler version of the 0-1 knapsack problem.

Problem 3 Although the streets in Manhattan are grid-like[1], it may still be tough to navigate driving because of the one-way streets. You are to design an algorithm to help your taxi driver get from an (avenue, street) intersection to another (avenue, street) intersection, given a map of the Manhattan. The map of the Manhattan is given as a graph where the nodes are such intersections, and the (directed) edges connect two adjacent intersections (ie, where the first or the second coordinate differs by exactly 1).

Design an algorithm for navigating from a start intersection to the end intersection in time $O(n+m)$, where n is the number of intersections and m is the number of edges, in the following two cases:

a) Assume that it takes 1 minute to traverse any given edge (adjacent intersections).

   Solution: Since each edge is of equal weight, we can use a simple Breadth-First Search (BFS) on the directed graph starting from the start intersection until we find the end intersection. This is because a BFS will first search all nodes that are 1 edge away from the start intersection, then reach all nodes that are 2 edges away, etc. Thus, if we find the end intersection, we know we will have found the shortest number of edges traversed to get from the start intersection to the end intersection. BFS runs in time $O(n + m)$, which satisfies the given time constraint.

b) Recognizing that avenue blocks are longer, assume that all edges have an associated time-to-traverse number, which is either 1 or 2, given as input as well.

   Solution: We can run BFS on a modified graph where edges (u, v) with weight 2 are replaced by two weight 1 edges (u, uv) and (uv, v) where uv represents a new node for each edge of weight 2 in the original graph. The time complexity for this algorithm is still $O(n + m)$ because we at most double the number of vertices and edges. A similar argument yields an algorithm for the bonus as well, splitting up both edges with weight 1 and weight 2.3 into edges of size 0.1.

   Solution: (Alternate but more complex) We can run Dijkstra's algorithm from the start intersection to the end intersection. Using the standard Dijkstra's algorithm with a Fibonacci heap would yield a runtime of $O(m + n \log n)$. However, we can improve upon this by modifying our priority queue implementation to make use of the fact that the edge weights lie in $\{1, 2\}$.

[1]Ok, unless we venture too much south, but we'll assume we don't.

Our first observation is that since edge weights are bounded, the maximum length of any path in the graph is $2n$ (traversing n vertices each with edge weight a maximum of 2). Thus, instead of using a standard priority queue using a Fibonacci heap, we can initialize an array of size $2n+1$, corresponding to the values of the possible distances. Each element in this array is a bucket that stores a linked list of vertices. All vertices initially start off in bucket $2n$ (corresponding to distance ∞). Next, we need to describe the two operations needed: Extract-Min and Decrease-Key.

Extract-Min: We simply iterate through the buckets starting from bucket 0 and return a vertex from the first non-empty bucket. Because the buckets correspond to distances, this will be the minimum distance element. Rather than doing this scan every time, we can simply keep track of the last bucket extracted and start scanning from that bucket. Over the course of Dijkstra's algorithm, we call this function n times (once for every vertex), and each bucket is only seen at most once, yielding an amortized run-time of $O(1)$ for Extract-Min.

Decrease-Key(v): We simply move the key from its current bucket to the bucket corresponding to the updated distance to v, $d[v]$. Thus, Decrease-Key(v) also takes $O(1)$ time.

Using these two operations, we have that this modified Dijkstra's algorithm takes time $O(n+m)$ and by the correctness of Dijkstra's algorithm, will return a shortest path from the start intersection to the end intersection.

Problem 4 You want to plan a company party. The company has a hierarchical binary-tree structure; that is, the supervisor relation forms a binary tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend.You are given the tree that describes the structure of the corporation. You know for each node, the left-child, right-child and parent Each node of the tree also holds the name of an employee and that employee?s conviviality ranking.Give a dynamic programming recurrence that find the set of employees of maximum total conviviality that you can invite. Be sure to explain what your variables represent.Explain what the running time of a bottom-up implementation of the algorithm would be.You do not have to give pseudocode, or a proof of optimal substructure.

Solution:

$$f(v) = \max \begin{cases} \sum_{u \in child(v)} f(u), \\ conviviality(v) + \sum_{u \in child(child(v))} \end{cases}$$

Pseudocode:

```
DP($v$)
1: $f(v) = conviviality(v)$, $g(v) = 0$
2: $s = v- > left$
3: while $s! = NULL$ do
4:     call $dp(s)$
5:     $f(v) = f(v) + g(s)$, $g(v) = g(v) + f(s)$
6:     $s = s- > right$
7: end while
8: return $\max(f(v), g(v))$
```

In the pseudocode, $f(v)$ represent the maximum conviviality of the subtree rooted at v, conditions on pick v itself. The $g(v)$ is similar except that it conditions on not pick v itself. For the problem, the value returned by $dp(root)$ is the answer.

The correctness follows by induction. To see this, consider a node v, for which we only have two choices: pick v or not. If we pick v, then we cannot pick any child of v. The maximum value of each subtree rooted at a child of v is independent, so $conviviality(v) + \sum_{u \in child(child(v))}$ is still the optimum solution. If we do not pick v, then there is no constraints on child node, and still with the independence, $\sum_{u \in child(v)} f(u)$ is still the optimum solution.

Each node will be visited once, and there is only $O(1)$ calculation required on each node. So, the overall time complexity is $O(n)$.

Problem 5 The `Lazy-Min-Heap` data structure supports `Insert` and `Delete-Min` operations. Let L be a linked list and H be a normal heap. The operation `Insert`(x) is implemented by inserting x into the linked list L. The operations `Delete-Min` is implemented by 1) Inserting all the items in L into the heap H. 2) Setting the linked list L to be empty. 3) Performing `Delete-Min` on H.

a) What are the worst-case running times of `Insert` and `Delete-Min` in the lazy min-heap?

   Solution: First, we analyze the worst-case running time of `Insert`(x). If we insert to the head of the list L, we can always insert x into L in $O(1)$ time. The worst-case for `Delete-Min` occurs when we have n elements in L. Then, the time to insert all n items into the heap is $O(n \log n)$, the time to clear the linked list is $O(n)$, and the time to perform one `Delete-Min` operation on H is $O(\log n)$. This gives a total worst-case running time of $O(n \log n)$ for `Delete-Min` on the lazy min-heap.

b) Prove that in the lazy min-heap, `Insert` and `Delete-Min` each have an amortized running time of $O(\log n)$.

   Solution: We will prove this using the banker's method of assigning amortized costs to each operation $\hat{c}_i$ such that for any sequence of ℓ operations,

$$\sum_{i=1}^{\ell} \hat{c}_i \geq \sum_{i=1}^{\ell} c_i$$

   Suppose that $\hat{c}_i$ is $O(\log n)$ for both the `Insert` and the `Delete-Min` operation. For any sequence of ℓ operations, it is enough to prove the above equation holds for the first j operations such that the operations are some number of `Insert` operations followed by some number of `Delete-Min` operations (where either operation can occur 0 times as well).

If we perform n `Insert` operations, the real cost of the first `Delete-Min` after these operations is $O(n \log n)$, and all `Delete-Min` operations after that cost $O(\log n)$ (since the heap is already fully populated). Thus, assigning the amortized cost of `Insert` to be $O(\log n)$, we can "pay" for the first `Delete-Min` operation, and assigning the amortized cost of the `Delete-Min` operation to be $O(\log n)$ pays for the remaining operations. Thus, we have that the sum of the amortized costs is always at least as large as the real costs, proving the claim that in the lazy min heap, `Insert` and `Delete-Min` each have an amortized running time of $O(\log n)$.