CS 170: Efficient Algorithms and Intractable Problems

Spring 2017

●

Homework 5

Due on Tuesday, March 7th, 2017 at 11:59am

●

Solutions by

Michael Fan

26697596

In collaboration with

Cedric Nixon

Problem 1: Name Distances

★ Level

Let A be your first name (yes, you, the student) either reduplicated or trimmed until it is 10 letters long. Let B be your last name, similarly reduplicated or trimmed until it is 10 letters long.

Specifically, if your name is fewer than 10 letters long, reduplicate it as necessary (e.g. "ohio" becomes "ohioohiooh", and "nevada" becomes "nevadaneva"), and if your name is longer than 10 letters long, trim it (e.g. "mississippi" becomes "mississipp").

Find the edit distance between A and B! (You may assume upper-case and lower-case letters are equivalent).

Show your work by running the algorithm from the textbook/lecture, drawing the edit distance table, and indicating the path corresponding to the solution.

Solution

$$\left[\begin{array}{ccccccccccc}
 & F & A & N & F & A & N & F & A & N & F \\
M & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
I & 2 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
C & 3 & 3 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
H & 4 & 4 & 4 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
A & 5 & 5 & 5 & 5 & 4 & 5 & 6 & 7 & 8 & 9 \\
E & 6 & 6 & 6 & 6 & 5 & 5 & 6 & 7 & 8 & 9 \\
L & 7 & 7 & 7 & 7 & 6 & 6 & 6 & 7 & 8 & 9 \\
M & 8 & 8 & 8 & 8 & 7 & 7 & 7 & 7 & 8 & 9 \\
I & 9 & 9 & 9 & 9 & 8 & 8 & 8 & 8 & 8 & 9 \\
C & 10 & 10 & 10 & 10 & 9 & 9 & 9 & 9 & 9 & 9
\end{array}\right]$$

The solution is following the main diagonal:
M I C H A E L M I C
F A N F A N F A N F

Problem 2: Weighted Set Cover

★★★ Level

In class (and Chapter 5.4) we looked at a greedy algorithm to solve the *set cover* problem, and proved that if the optimal set cover has size k, then our greedy algorithm will find a set cover of size at most $k \log n$.

Here is a generalization of the set cover problem.

- *Input:* A set of elements B of size n; sets $S_1, \ldots, S_m \subseteq B$; positive weights $w_i, \ldots, w_m$.
- *Output:* A selection of the sets S_i whose union is B.
- *Cost:* The sum of the weights w_i for the sets that were picked.

Design an algorithm to find the set cover with approximately the smallest cost. Prove that if there is a solution with cost k, then your algorithm will find a solution with cost $\mathcal{O}(k \log n)$. Please provide the Main Idea and Proof of Correctness only. If you think Pseudocode can better convey your idea, add it to the *Main Idea*.

Note: We will accept solutions whose running time is a reasonable (low-order) polynomial in n and m. Do not expend too much effort trying to get the running time as low as possible; we are much more concerned with achieving the required approximation factor.

Solution
Main Idea:
Greedily choose and add the set with the lowest $\frac{cost}{uncoveredelements}$ ratio.

Proof of correctness:
Let us consider the pth chosen set from our algorithm. Let w_p be the cost of this set and u_p be the number of uncovered elements in the set. The ratio of cost to uncovered elements in the set is $\frac{w_p}{u_p} \leq \frac{k}{\sum_j u_j}$ because otherwise k would not be the optimal cost.
We now must show that $\sum_i cost(n_i) \in O(k \log n)$. At the jth iteration when the ith element is covered as a member of some set, $\sum_j u_j \geq n - i$ since at least i elements must have been covered. Therefore, $cost(n_i) \leq \frac{k}{n-i}$ and the worst case ends up being when every set chosen contains a single element such that $\sum_i cost(n_i) \leq \frac{k}{n} + \frac{k}{n-1} + ... + k = k(\frac{1}{n} + ... + 1 \approx k \log n$. Therefore, $\sum_i cost(n_i) \in O(k \log n)$.

Problem 3: Ternary Huffman

★★★★ Level

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. Your proof of correctness should prove that your algorithm achieves the maximum possible compression.

Please provide the main idea and proof of correctness only. If you think pseudocode can better convey your idea, add it to the *Main Idea*.

If you are stuck on the proof of correctness, please see proof of binary huffman from lecture.

Solution
Main Idea:
Take the three least frequent elements of the alphabet and collapse them into 1 node with frequency equal to the sum of the individual frequencies. Put this node back into the collection. Recurse until there is a root node that parents all members of the alphabet. If there size of the alphabet is even, add a dummy alphabet of frequency 0 before starting the algorithm.

Proof of Correctness:
Every time we take a group of 3 nodes, collapse them into 1 node, and put that node back in the pile, we are essentially a net of 2 nodes from the set that we still have to process. Thus, for an alphabet of size $k > 3$, there must be an odd number of elements in the set to start with if we are to end with a complete ternary tree. Thus, in the case that we have an even number of elements in the alphabet, we add a dummy element with frequency 0 to adjust the set to an odd parity. The proof of correctness now follows as it did in the case of a binary tree. We know that the encoding is prefix-free as traversing along the ternary tree will not allow any repeated prefixes by its construction. We now show that this algorithm achieves the maximum compression. We consider the three elements of the least frequency (this contains the dummy node if there is one). If these 3 nodes are not the siblings that are deepest in the tree, then we replace them with the 3 siblings that are deepest in the tree. There is guaranteed to be a group of 3 siblings on the deepest level of the tree for an alphabet of size greater than 3 by the nature of our algorithm and because the the construction ensures a complete ternary tree. However we will never have to do this replacement as we chose the three least frequent nodes to start with and thus they will inevitably be the deepest level of the tree. The second case is if there is an optimal encoding with three of the lowest nodes are on the same level. Then, they can always be rearranged into siblings and our algorithm outputs just that.

Problem 4: Birthday Surprise

★★★★ Level

Anakin Skywalker has assigned his droid, C3PO, a series of tasks to complete before Padme's birthday. C3PO has N tasks labeled $1, \ldots, N$. For each task, it gains some Republic credits $V_i \geqslant 0$ for completing the task, and some operating cost $P_i \geqslant 0$ per day that accumulates for each day until the task is completed. It will take $R_i \geqslant 0$ days to successfully complete task i. C3PO needs to accumulate enough Republic credits to have a droid makeover before it requires maintenance.

Each day, C3PO chooses one unfinished task to complete. A task i has been finished if it has spent R_i days working on it. This doesn't necessarily mean it has to spend R_i consecutive days working on task i. C3PO starts on day 1, and wants to complete all tasks and finish with maximum Republic credits. If C3PO finishes task i at the end of day t, it will get reward $V_i - t \cdot P_i$. Note, this value can be negative if it chooses to delay a task for too long.

Given this information, what is the optimal task scheduling policy to complete all of the tasks?

Please provide the main idea and proof of correctness only. If you think pseudocode can better convey your idea, add it to the *Main Idea*.

Solution

Main Idea:

We will solve this problem with a dynamic programming solution. Let us assume that there are N tasks that give rewards of $V_1, .., V_N$, give a daily penalty of $P_1, ..., P_N$, and take $R_1, ..., R_N$ days to complete. First, note 2 observations.

(1) It will take exactly $R_1 + ... + R_N$ days to complete every task.

(2) It is always better to finish a task in consecutive days than it is to pause and start another task.

The final problem that we have to solve is the optimal value that C3PO can attain given $R_1 + ... + R_N$ days to complete the task. Let us consider the subproblem of the maximum value that he can attain in just k days. If we can figure out how to compute this value with the solutions to subproblems of having $1, ..., k-1$ days left, then we can apply the same principle and iterate upwards to solve the original problem. Let us sort the tasks by their completion times such that $R_1 \leq R_2 \leq ... \leq R_N$. Without loss of generality, let us assume $k > R_1, ..., R_i$. By observation 2, we know that C3PO should complete tasks back to back - choose something to do, finish it, and then recurse. Thus, he has i choices on what he can do first which leads to i different scenarios that lead to i possibly distinct values: $v_1 = V_1 - R_1 * P_1 + subproblem(k - R_1, 1)$, ..., $v_i = V_i - R_i * P_i + subproblem(k - R_i, i)$. We pass in the index of the task as the second parameter to make sure that we can check that we don't choose the same task twice. The important nuance is that each $subproblem(k - R_e), 1 \leq e \leq i$ is assumed to be ALREADY SOLVED! Thus, to get the optimal value with k days to do stuff in, we just take $max(v_1, ..., v_i)$. Each subproblem "remembers" the optimal actions that it took to get the optimal value it contains as well because we pass in the current action in as the second parameter, so if one queries for the sequence of actions, that can be returned immediately. If we consider the sequence of subproblems starting from the very beginning, we first consider $subproblem(0)$. In the case that no tasks can be done in 0 days, we trivially return 0. Otherwise, $subproblem(0) = max(V_1 - R_1 * C_1 + subproblem(0, 1), ..., V_b - R_b * C_b + subproblem(0, b))$ where task e, $1 \leq e \leq b$ can be done in 0 days. We then recurse onwards until we finally solve $subproblem(R_1 + ... + R_N) = max(v_1, ..., v_N)$ where v_e, $1 \leq e \leq N$ is as defined as $v_e = V_e - R_e * c_e + subproblem(N - R_e, e)$.

Proof of correctness:

We will first prove the observations:

(1) Because C3PO cannot "stall" and do nothing on any one day (and it would be suboptimal for him to so when he could be working toward completing any one task) no matter how optimal the final attained value is, C3PO will take exactly $R_1 + ...R_N$ days to complete everything.

(2) This is because the fast C3PO finishes one task, the sooner he can stop paying the daily penalty for that task. Given any 2 tasks t_1 and t_2, it will always take $R_1 + R_2$ to complete the 2 tasks so it is always optimal to finish the task with the higher daily cost before starting the other one.

Solution (cont.)

We will proceed via induction to prove the correctness of the algorithm. Assume that we have sorted the tasks in ascending order of the time it takes to complete them.
Base case (0 days left): If there are no tasks that can be completed in 0 days, then we trivially return 0. Otherwise, $subproblem(0) = max(V_1 - R_1 * C_1 + subproblem(0, 1), ..., V_b - R_b * C_b + subproblem(0, b))$ where task e, $1 \leq e \leq b$ can be done in 0 days. There may be multiple tasks that can be done 0 days so we can stack all of them which this algorithm accomplishes. Note that we pass in the index of the action taken which we save in a set of some sort so we don't repeat actions.
Inductive Hypothesis: Assume that for k days left, $k > 0$, we correctly return an optimal value v_k. We will use strong induction here and claim that for all $subproblem(e)$, $0 \leq e \leq k$, our algorithm is correct.
Inductive Step: Consider the case of $k + 1$. There are 2 cases.
Case 1: There is no new task that C3PO can accomplish in $k + 1$ days as opposed to k days. Then, $subproblem(k + 1) = max(v_1, ..., v_k)$ where v_e, $1 \leq e \leq k$ is defined as $v_e = V_e - R_e * c_e + subproblem(k + 1 - R_e, e)$. We know from observation 2 that it is optimal to start and complete a task before moving onto another, so it is always optimal to do one of the k actions before considering the subproblem of what to do in the remaining time, $subproblem(k + 1 - e)$. Furthermore, it is given by strong induction that $subproblem(k + 1 - e)$ is correctly solved as $k + 1 - e \leq k$. Thus, in this case, we will find the optimal value for day $k + 1$. Case 2: There is an additional task indexed $k + 1$ that can now be finished in k+1 days that could not be accomplished before. Now, the $k + 1$ day can be formalized as $subproblem(k + 1) = max(v_1, ..., v_{k+1})$ where v_e, $1 \leq e \leq k + 1$ is as defined as $v_e = V_e - R_e * c_e + subproblem(k + 1 - R_e, e)$. Again from observation 2, we know that it is optimal to start and complete a task before moving onto another which also applies to the task indexed at k+1. The rest of the proof follows as above, and we have shown that our algorithm produces the optimal value for day $k + 1$.

Problem 5: Finding Maul

★★★★ Level

The Jedi Council has heard rumors that Maul has been reborn and amassing an army. Obi-Wan Kenobi has been tasked with finding Maul before he can threaten the republic. In order to do so, Obi-Wan leverages the underworld network and aims to find all of the *crime-lords*. A *crime-lord* is a person in the black market who has a link to at least 20 other crime-lords.

We can formalize this as a graph problem. Let the undirected graph $G = (V, E)$ denote the underworld's relationship graph, where each vertex represents a person who has significant dealings in the black market on Coruscant. There is an edge $\{u, v\} \in E$ if u and v have a known relationship with each other on Coruscant (we will assume that relationships are symmetric). We are looking for a subset $S \subseteq V$ of vertices so that every vertex $s \in S$ has edges to at least 20 other vertices in S. And we want to make the set S as large as possible, subject to these constraints.

Design an efficient algorithm to find the set of crime-lords (the largest set S that is consistent with these constraints), given the graph G. Unlike previous questions in this homework, please use a 4-part algorithm solution.

Hint: There are some vertices you can rule out immediately as not crime-lords.

Solution
Main Idea:
Iterate through all edges to find the degree of all vertices. Delete the vertices with degree < 20 and their incident edges, and recurse until nothing is deleted. Return the remaining vertices. These are the crime lords.

Pseudocode:

```
function findCrimeLords(G(V, E))
    degreeMap = map of (vertex, int)
    for all v in V, put (v, 0) in degreeMap
    for (u, v) in E, increment degreeMap(u), degreeMap(v)
    vertexQueue = all v in V
    while vertexQueue is not empty:
        v = vertexQueue.pop
        if degreeMap(v) < 20:
            for each (u, v) in E:
                decrement degreeMap(u)
                vertexQueue.push(u)
                Remove edge(u, v) from G
            Remove v from G
    return remaining vertices in V
```

Proof of correctness:
A crime-lord is a vertex that is connected to at least 20 other degree 20 vertices. Thus, any vertex that has degree less than 20 cannot be a crime-lord. Furthermore, any connection that a vertex has with another vertex with degree less than 20 does not contribute to the vertex's crime-lord status. Thus, we can safely remove all vertices with degree less than 20 (they cannot be crime-lords as stated) and their incident edges without disrupting the crime-lord status of any other vertex. Recursively, this algorithm is correct by the same analysis. In the end when no more vertices are removed, every remaining vertex is connected to at least 20 other vertices (which by consequence of recursive deletion also must have at least 20 other connections) and is a crime-lord. We have already shown that no potential crime-lords are ever deleted by a consequence of this algorithm and no non-crime-lords are left undeleted so this algorithm is correct.

Runtime analysis:
It takes O(V) time to create the map. It takes O(E) time to go through all edges initially and O(1) time to update each entry. For each vertex that is removed, there is a max of 19 map entry updates which is O(1) all

Solution (cont.)

together and then it is constant time to remove the vertex after all incident edges are already disposed. We at most remove every vertex eventually which leads to an inner loop of V. Thus, the algorithm is dominated by checking every edge which is $O(E) = O(V^2)$.