CS 170
Spring 2017

Efficient Algorithms and Intractable Problems
Prasad Raghavendra and Sanjam Garg

Homework 5

Instructions: You are welcome to form small groups (up to 4 people total) to work through the homework, but you **must** write up all solutions by yourself. List your study partners for homework on the first page, or "none" if you had no partners.

If using LaTeX (which we recommend), you may use the homework template linked on this Piazza post to get started.

Begin each problem on a new page. Clearly label where each problem and subproblem begin. The problems must be submitted in order (all of P1 must be before P2, etc). For questions asking you to give an algorithm, respond in what we will refer to as the *four-part format* for algorithms: main idea, pseudocode, proof of correctness, and running time analysis.

Read the Homework FAQ Piazza post on Piazza before doing the homework for more explanation on the four-part format and other clarifications for our homework expectations.

No late homeworks will be accepted. No exceptions. This is not out of a desire to be harsh, but rather out of fairness to all students in this large course. Out of a total of approximately 12 homework assignments, the lowest two scores will be dropped.

Special Questions:

- *Shortcut questions*: Short questions are usually easy questions that give you opportunities to practice basic materials. However, we understand that some of you are very familiar with the topics and do not want to spend too much time on easy questions. Therefore, we design shortcut questions for this purpose. A shortcut question usually has multiple parts that build upon each other and are ordered by their difficulty level. You can work on those in order or start from wherever you like. However you only need to submit the last part you are able to solve. For example, if a question has 5 parts (a, b, c, d, e), you are confident about part e, you should submit part e without any of the previous four parts. If you are confident about d but not sure about e, you should submit d for grading purposes. Please clearly indicate in your submission which part you are submitting.
- *Redemption questions*: It is important that you carefully read the posted solutions, even for problems you got right. To encourage this, you have the option of submitting a redemption file, a few paragraphs in which you explain, for each problem you choose to cover, what you did wrong and what the right idea was in your own words (not cutting and pasting from the solution!), and appending it to your homework. For example, suppose that as you review your solutions to HW1, you realize you had misunderstood question 3 and answered it incorrectly. You would write down what you just learned, and then submit it in your HW2 assignment the following week. Because these are mainly for your benefit, feel free to format them however is most useful for you.
- *Extra credit questions*: We might have some extra credit questions in the homework for people who really enjoy the materials. However, please note that you should do the extra credit problems only if you really enjoy working on these problems and want an extra challenge. It is likely not the most efficient manner in which to maximize your score.

Due Tuesday, March 7, at 11:59am

1. (★ level) Name Distances

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: Solutions will naturally vary for each student. We will go through an example with $A =$ "prasadragh" and $B =$ "sanjamgarg", with the final path being bolded.

		S	A	N	J	A	M	G	A	R	G
	0	1	2	3	4	5	6	7	8	9	10
P	1	**1**	2	3	4	5	6	7	8	9	10
R	2	2	**2**	3	4	5	6	7	8	8	9
A	3	3	2	**3**	4	4	5	6	7	8	9
S	4	3	3	3	**4**	5	5	6	7	8	9
A	5	4	3	4	4	**4**	5	6	6	7	8
D	6	5	4	4	5	5	**5**	6	7	7	8
R	7	6	5	5	5	6	6	**6**	7	7	8
A	8	7	6	6	6	5	6	7	**6**	7	8
G	9	8	7	7	7	6	6	6	7	**7**	7
H	10	9	8	8	8	7	7	7	7	8	**8**

2. (★★★ level) Weighted Set Cover

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 $w_1, \ldots, w_m$.
- *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 $O(k \log n)$. 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:

As in the unweighted case, we will use a greedy algorithm:

Pseudocode:

1. **While** some element of B is not covered
2. Pick the set S_i with the largest ratio (Number of new elements covered by S_i) / w_i.

Now we will prove that if there is a solution of cost k, then the above greedy algorithm will find a solution with cost at most $k \log_e n$. Our proof is similar to the proof in the unweighted case in Section 5.4 of the textbook.

After t iterations of the algorithm, let n_t be the number of elements still not covered, and let k_t be the total weight of the sets the algorithm chose, so $n_0 = n$ and $k_0 = 0$. Since the remaining n_t elements are covered by a collection of sets with cost k, there must be some set S_i such that S_i covers at least $w_i n_t / k$ new elements. (This is easiest to see by contradiction: if every set S_i covers less than $w_i n_t / k$ elements, then any collection with total weight k will cover less than $k n_t / k$ elements.) Therefore the greedy strategy will ensure that

$$n_{t+1} \leq n_t - \frac{n_t(k_{t+1} - k_t)}{k} = n_t(1 - (k_{t+1} - k_t)/k).$$

Now, we apply the fact that for any x, $1 - x \leq e^{-x}$, with equality iff $x = 0$:

$$n_{t+1} < n_t e^{-(k_{t+1} - k_t)/k}.$$

By induction, we find that for $t > 0$, $n_t < n_0 e^{-k_t/k}$. If we choose the smallest t such that $k_t \geq k \log_e n$, then, n_t is strictly less than 1, which means no elements remain to be covered after t steps. Since $k_{t-1} < k \log_e n$ and we will never add a set of weight more than k, it follows that $k_t < k \log_e n + k = O(k \log n)$.

Running Time (Not required for this problem)
For a loose running time bound, note that the while loop runs for at most m iterations (one for each set S_i in the worst case). Further, in each iteration, computing the set with the best ratio takes at most mn time, because at each step there are at most m sets left and each has size at most n. Thus the total time taken is $O(m^2 n)$. Note the main point of this problem was to get the correct approximation ratio, any reasonable low-order polynomial bound on the running time is fine.

3. (★★★★ level) 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. 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 As in the binary case, the idea is to repeatedly delete the three characters of smallest frequency, and replace them by a new character whose frequency is their sum. We wish to show that the ternary tree we get by this process is optimal. There is just one hitch — what happens if in the last step of the process we are left with only two characters. To see when this could happen, note that each step of the above process removes three characters and adds one, for a net decrease of 2. So if we start with an odd number of characters, we will eventually get down to exactly 3 characters, whereas if we start with an even number

we will eventually get down to exactly 2 characters, leading to a problem. The most elegant way to resolve this problem is to add a dummy character of frequency 0 if you start with an even number of characters, thus making sure that the above procedure always results in a full ternary tree.

Algorithm

TERNARY HUFFMAN($f[1..n]$):
1. If n is even, set $f[n+1] = 0$ and set $n = n+1$
2. Let H be a priority queue of integers
3. For $i = 1$ to n: insert i into H with priority $f[i]$
4. Set $l = n$
5. While H has more than one element
6. Set $l = l+1$
7. i = deletemin(H), j = deletemin(H), k = deletemin(H)
8. Create a node numbered l with children i, j, k
9. $f[l] = f[i] + f[j] + f[k]$
10. Insert l into H with priority $f[l]$
11. Return the tree rooted at node l

After running the above algorithm, we can assign codewords in the same manner as in normal Huffman encoding– i.e. write 0, 1 and 2 on the outgoing edges of each node in the tree and encode a character by the sequence of digits encountered along the path from the root of the tree to the leaf corresponding to that character.

Proof of Correctness First we observe that adding a character with a frequency of 0 does not change the maximum possible compression since the length of the codeword associated to that character does not affect the length of any encoded message. So it suffices to prove that the above algorithm gives the optimal encoding tree when n is odd.

Now observe that when n is odd, any optimal encoding tree must be a full ternary tree. i.e. every internal node has exactly three children. To see this, note that if a node has two children and is not at the second from the bottom level of the tree, then we can simply move a leaf from the bottom level to be its child, and improve the cost of the solution. So there can be at most one node in the tree at the second from the bottom level, which has two children. But as we argued earlier, if the total number of leaves is odd, such a node cannot exist.

Now we argue as in the binary case that the three lowest frequency leaves must be at the bottom level, since otherwise we could exchange them with whichever leaves were at the bottom level to decrease the cost. Finally, we can always move around leaves at the same level without affecting the cost, and therefore we can assume that the three lowest frequency characters are siblings on the bottom level of the tree.

At this point, the rest of the proof is nearly identical to the proof of optimality for binary Huffman encoding, but we present it here for completeness. We will proceed by induction on the number of characters. The base case, $n = 3$, is trivial (it is clearly optimal to assign each character a length 1 code, which is exactly what the algorithm above does). So assume for induction that n is an odd integer greater than 3 and ternary Huffman encoding is optimal when there are $n-2$ characters. Let $f_1, f_2, \ldots, f_n$ be a set of frequencies for n characters and let T be the tree constructed by ternary Huffman encoding from these frequencies. Let S be an optimal encoding tree for a ternary prefix free encoding of the n characters and assume that the three lowest frequency characters are siblings on the bottom level of S.

Let i, j and k be the indices of the three lowest frequency characters. Consider the set of characters that results from eliminating i, j and k and replacing them with a new character l with frequency $f_i + f_j + f_k$. Let

S' and T' denote the result of deleting i, j and k from S and T respectively. S' and T' can be thought of as encoding trees for this new set of characters. Observe also that T' is exactly the tree that ternary Huffman encoding would construct given this new set of characters, so by the inductive assumption T' is optimal.

For an encoding tree R, let $d_R(x)$ denote the depth of x in R and let $Cost(R)$ denote the expected length of text encoded using R. We have:

$$\begin{aligned} Cost(T) &= \sum_{x=1}^{n} f_x d_T(x) \\ &= \sum_{x \neq i,j,k} f_x d_T(x) + d_T(i)(f_i + f_j + f_k) \\ &= \sum_{x \neq i,j,k} f_x d_{T'}(x) + (d'_T(l) + 1)(f_i + f_j + f_k) \\ &= Cost(T') + (f_i + f_j + f_k) \end{aligned}$$

Similar calculations show that $Cost(S) = Cost(S') + (f_i + f_j + f_k)$. Since T' is optimal, $Cost(T') \leq Cost(S')$, which implies that

$$Cost(T) = Cost(T') + (f_i + f_j + f_k) \leq Cost(S') + (f_i + f_j + f_k) = Cost(S)$$

And so T is optimal.

Running Time (Not required for this problem) Adding all characters to the priority queue takes $O(n\log(n))$ time, assuming we use a standard binary heap. Then on each iteration of the while loop, removing three elements and adding one new one takes $O(\log(n))$ time. Furthermore, since the number of elements in the priority queue is reduced by two on each iteration (and started out odd), after $(n-1)/2$ iterations there is only one element left in the priority queue. So the algorithm takes at most $O(n\log(n))$ time.

4. (★★★★ level) Birthday Surprise

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 $V_i \geq 0$ for completing the task, and some operating cost $P_i \geq 0$ per day that accumulates for each day until the task is completed. It will take $R_i \geq 0$ 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 finishs 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: First, complete all tasks with $R_i = 0$. Sort the remaining tasks in order of decreasing P_i/R_i and allocate the first R_i available days to that task. Repeat until all N tasks have been completed.

Intuitively, this balances maximizing P_i for the next tasks to reduce future penalty and minimizing R_i to get it done quickly.

Pseudocode:

1. Let the set of all tasks be $T := \{1..n\}$
2. Let tasks with zero completion time be $U := \{i \mid R_i = 0\}$
3. Let remaining tasks be $S := T - U$
4. Sort S by P_i/R_i in descending order
5. Output U concatenated with S

Proof of correctness: Because interleaving the work for several tasks will delay all the tasks' completion dates, interleaving will necessarily have a higher penalty than a contiguous scheduling, and cannot be a part of any optimal solution. Therefore, we focus our attention only on orders where each task is scheduled on contiguous days: once you start a task, you complete it before starting any other task. Also, since we must complete all tasks, the V_i values are irrelevant: the base reward V_i is always received, no matter what order we schedule the tasks. Therefore, we can treat all the V_i as zero and just minimize the total penalties accrued, and this won't change the optimal solution.

To prove that our order is optimal, we will use the following swapping rule:

- Suppose the tasks are numbered $1,2,\ldots,n$ in the order they appear in the ordering, and suppose $P_j/R_j \leq P_{j+1}/R_{j+1}$. Then we can swap tasks j and $j+1$.

We'll first prove that this modification, if applied to any order, will never make things worse (it will never increase the penalty). Why? Well, the penalty for the days when tasks j and $j+1$ are being done is

$$\rho = R_jP_j + (R_j+R_{j+1})P_{j+1} + (R_j+R_{j+1})(P_{j+2}+P_{j+3}+\cdots+P_n),$$

before the swap. After the swap, the penalty for that time period becomes

$$\rho' = (R_j+R_{j+1})P_j + R_{j+1}P_{j+1} + (R_j+R_{j+1})(P_{j+2}+P_{j+3}+\cdots+P_n).$$

Notice that

$$\rho' = \rho + R_{j+1}P_j - R_jP_{j+1}.$$

Now if $P_j/R_j \leq P_{j+1}/R_{j+1}$, then $R_{j+1}P_j \leq R_jP_{j+1}$, so $\rho' \leq \rho$. The penalty for the other days is unaffected by the swap. This proves that the swapping rule above can never increase the penalty.

Now let's prove that our algorithm generates an optimal ordering. If we start from any order, then we can repeatedly apply the swapping rule above until the tasks are ordered by decreasing P_i/R_i-value, i.e., until $P_1/R_1 \geq P_2/R_2 \geq \cdots \geq P_N/R_N$. Basically, we just apply bubble sort: take the task with largest P_i/R_i and swap it forward until it is at the start of the order; then take the task with second-largest P_i/R_i and repeatedly swap to move it forward until it as second in the order; and so on. We can see that each step, the swap moves a task with larger P_i/R_i-value forward in front of a task with smaller P_i/R_i-value, so is allowed by the swapping rule. Therefore, the swapping rule above allows us to transform any order into the one output by our algorithm, without increasing the total penalty. As a consequence, the order produced by our algorithm must be optimal.

Running Time (Not required for this problem): The proof of correctness uses bubblesort (comparing two adjacent elements) to show that the sorted order is optimal. In reality we can use a more efficient sorting algorithm, to get a runtime of $O(N \log N)$.

5. (★★★★ level) Finding Maul

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.

The basic idea here is that we iteratively remove non-crime lords until we reach a fixed point. Any node whose degree is below 20 is certainly a non-crime lord. We keep a worklist (implemented as a set) of pending non-crime lords. In each iteration we remove a non-lord from the worklist, delete it, adjust the degree of its neighbors, and add them to the worklist if their degree has fallen below 20.

Pseudocode.

1. Set $d[v] := 0$ for each $v \in V$.
2. For each edge $\{u, v\} \in E$:
3. Increment $d[u]$. Increment $d[v]$.
4. Initialize a set W as $W := \{v \in V : d[v] < 20\}$.
5. While W is non-empty:
6. Remove a vertex from W; call it v.
7. For each $\{v, w\} \in E$:
8. Decrement $d[w]$.
9. If $d[w] < 20$ and $w \notin W$, add w to W.
10. Remove v from the graph.
11. Output the set of nodes left in the graph (they are the crime-lords).

Proof of correctness.

Lemma: At each iteration of the while loop, for each vertex v that has not yet been deleted, $d[v]$ = the degree of v (in the graph that remains).

Proof: This invariant can be easily proven by induction on the number of iterations of the loop. Whenever we delete a vertex, we decrement each of its neighbors to reflect the change in their degree.

Claim: After the algorithm ends, we've identified a subgraph G' such that every user in G' has degree at least 20, in that subgraph. In other words, the algorithm outputs a valid set of vertices who satisfy the requirements to be criminal crime-lords.

Proof: At the beginning of every iteration of the loop, every node either has degree at least 20, or is in the worklist W. Nodes in the worklist could not possibly be crime-lords. This is true before the first iteration because that's how line 4 initialized the worklist. After that, a node's degree can only change when one of its neighbors is removed, and whenever a node is deleted we check all of its neighbors' degrees. Therefore, after the loop, when the worklist is empty, every remaining vertex [if any] will have degree ≥ 20, and therefore can be validly labeled a lord.

Claim: Everyone who can be validly labeled as a lord is included in the output of the algorithm.

Proof: Let S be the largest set of vertices that can be validly labeled as crime-lords. Then it is an invariant that $d[s] \geq 20$ and $s \notin W$, throughout the algorithm, for all $s \in S$. This is true before the first iteration because of how line 4 initialized the worklist. Also, if it is true before one iteration of the loop, it remains true after that iteration. In particular, consider an iteration where we remove v from the worklist. By the contrapositive of the inductive hypothesis and using the fact that we had $v \in W$ at the start of this iteration, it follows that $v \notin S$. Also, for each neighbor w of v, if $w \in S$, by the inductive hypothesis w had an edge to ≥ 20 other members of S; since $v \notin S$, after deleting v it still has an edge to ≥ 20 other members of S. Therefore, the invariant remains true after this iteration of the loop, and the claim follows by induction.

The first claim shows that the algorithm's output is not "too large", and the second claim shows that the algorithm's output is not "too small". This implies that our algorithm finds the largest possible set of criminal crime-lords, as desired.

Running time. $O(|V| + |E|)$. We examine each vertex twice (in lines 1 and 4), and do a constant amount of work per vertex in each case. Each vertex is inserted into W at most once, so the number of iterations of the loop in lines 5–10 is at most $|V|$, and each iteration takes $O(1)$ time, ignoring the inner loop at lines 7–9. (Notice that using a HashSet you can insert, remove and lookup items in $O(1)$ time.) Also, the inner loop at lines 7–9 examines each edge at most once, when we remove one of its endpoints, and we we do a constant amount of work per edge in that case. This accounts for all of the work done in this algorithm, and we can see that we do $O(1)$ work per vertex plus $O(1)$ work per edge. Hence, the running time is $O(|V| + |E|)$.