CS 170 HW 5

Due **2020-10-05, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write none.

In addition, we would like to share correct student solutions that are well-written with the class after each homework. Are you okay with your correct solutions being used for this purpose? Answer "Yes", "Yes but anonymously", or "No"

2 Huffman Coding

This question is a solo question.

In this question we will consider how much Huffman coding can compress a file F of m characters taken from an alphabet of $n = 2^k$ characters x_0, x_1, ... , x_{n-1} (each character appears at least once).

(a) Let $S(F)$ represent the number of bits it takes to store F without using Huffman coding (i.e., using the same number of bits for each character). Represent $S(F)$ in terms of m and n.

(b) Let $H(F)$ represent the number of bits used in the optimal Huffman coding of F. We define the *efficiency* $E(F)$ of a Huffman coding on F as $E(F) := S(F)/H(F)$. Among all files with m characters and alphabet size n, describe the file F for which $E(F)$ is as small as possible.

(c) Among all files with m characters and alphabet size n, describe the file F for which $E(F)$ is as large as possible. How does the largest possible efficiency increase as a function of n? Give you answer in big-O notation.

Solution:

(a) $m\log(n)$ bits.

(b) The efficiency is smallest when all characters appear with equal frequency. In this case, $E(F) \approx 1$.

(c) Let F be x_0, x_1, ..., x_{n-2} followed by $m-(n-1)$ instances of x_{n-1}. This file has efficiency

$$\frac{m\log(n)}{(m-(n-1))\cdot 1 + (n-1)\cdot\log(n)}$$

This efficiency is best when m is very large, and thus $(m-(n-1))\cdot 1$ far outweighs $(n-1)\cdot\log(n)$. This results in the equation

$$\frac{m\log(n)}{(m-(n-1))\cdot 1 + (n-1)\cdot\log(n)} \approx \frac{m\log(n)}{(m-(n-1))\cdot 1} \approx \log(n)$$

So the efficiency is $O(\log(n))$.

3 Preventing Conflict

A group of n guests shows up to a house for a party, but m pairs of these guests are enemies (a guest can be enemies with multiple other guests). There are two rooms in the house, and the host wants to distribute guests among the rooms, breaking up as many pairs of enemies as possible. The guests are all waiting outside the house and are impatient to get in, so the host needs to assign them to the two rooms quickly, even if this means that it's not the best possible solution. Come up with a $O(n+m)$-time algorithm that breaks up at least half of the pairs of enemies. **Give a three-part solution. Solution:**

Main Idea: We add the guests one-by-one, each time adding a guest to the room that currently has less of their enemies (in the case of a tie, we choose one of the rooms arbitrarily).

Proof of Correctness: Let m_i be the number of the ith guest's enemies that have already been added to a room when we're adding the ith guest. We break up at least $m_i/2$ of these enemy pairs by adding this guest to the room with less of their enemies. So we break up at least $\sum_i m_i/2$ of the $\sum_i m_i$ total enemy pairs as desired.

Running Time: Each guest is iterated through once, and we can figure out which room to assign them in time proportional to the number of enemies they have (using e.g. an array tracking which room we've assigned each individual to so far), i.e. each pair of enemies is considered at most twice, so the total time is $O(n+m)$.

(Comment: This problem is equivalent to the max-cut problem: Given an undirected graph, in the max cut problem we want to split the vertices into two sets so that as many edges as possible have one endpoint in each set)

The remaining questions on this homework are optional and intended as additional resources for preparing for the midterm. They are not necessarily of an equivalent difficulty to the midterm problems, and will not be graded. Feel free to discuss these problems and their solutions publicly, including on Piazza.

4 (Optional) The Resistance

We are playing a variant of The Resistance, a board game where there are n players, k of which are spies. In this variant, in every round, we choose a subset of players to go on a mission. A mission succeeds if no spies are chosen to go on the mission, but fails if at least one spy goes on the mission, and when a mission fails we are not told who the spies are that went on the mission.

Come up with a strategy that identifies all the spies in $O(k\log(n/k))$ missions. Only a main idea and runtime analysis are needed.

Solution: Main Idea: We partition the n players into $2k$ groups, and send each group on a mission. For the missions that succeed, we remove those groups, and split the remaining groups in half to form a new set of groups. We repeat this procedure until each group has one person left, at which point we know they are a spy.

Runtime analysis: In the first iteration of this procedure we have $2k$ groups going on missions. After each group goes on a mission, since there are k spies, at most k of the missions fail, which means after splitting the groups that failed, we have at most $2k$ groups again.

In each iteration, at least half of the players are removed from groups. So we identify the spies after $O(\log(n/k)$ iterations. So the total number of missions needed is $O(k\log(n/k))$.

5 (Optional) Cyclic Shifts

Given an n-element vector u, the jth cyclic shift of u is the vectors $u^{(j)}$ defined as $u_i^{(j)} = u_{(i+j) \mod n}$ for some j (using 0-indexing). For example, the cyclic shifts of $u = (0, 1, 2)$ are $u^{(0)} = (0, 1, 2), u^{(1)} = (1, 2, 0)$, and $u^{(2)} = (2, 0, 1)$.

Given two n-element vectors u, v, give an efficient algorithm that computes the n dot products $u^{(0)} \cdot v, u^{(1)} \cdot v, \ldots, u^{(n-1)} \cdot v$. Just the main idea and runtime analysis are needed.

Hint: Recall the Protein Matching problem from HW3. Is there a short vector u' such that every cyclic shift of u is contained in u'?

Solution:

Main idea: Write the $2n$-element vector u', which is u repeated twice, and let v' be the reversal of v. Note that every "subvector" of u' is a cyclic shift of u.

We use FFT to multiply the polynomials $u'_0 + u'_1 x + \ldots u'_{2n-1} x^{2n-1}$ and $v'_0 + v'_1 x + \ldots v'_{n-1} x^{n-1}$. Multiplying these polynomials to get P, the coefficient of x^i in P for $n - 1 \leq i \leq 2n$ in c is $\sum_{j=0}^{n-1} u'_{i-j} v'_j = \sum_{j=0}^{n-1} u^{(i-n+1)}_{n-1-j} v_{n-1-j}$, so we can output these coefficients.

Runtime analysis: We can multiply the polynomials in $O(n \log n)$ time.

6 (Optional) Finding Ancestors

You are given a tree $T = (V, E)$ with a designated root node r. For each vertex v, let $a(v)$ be the kth ancestor of v in the tree (so for $k = 1$ this is the parent of v, for $k = 2$ this is v's parent's parent, etc.). We follow the convention that the root node, r, is its own parent. Modify DFS so that it takes r, k in as input, and computes $a(v)$ for all vertices in $O(|V|+|E|)$ time. Just a main idea is needed.

Solution:

Main Idea: We run DFS from r, and modify DFS to keep track of the list L of current ancestors of a vertex. When we pre-visit v, we add add v to L, and when we post-visit v, we remove it from L. Since $pre(u) < pre(v) < post(v) < post(u)$ if and only if u is an ancestor of v, this guarantees that before we add v to the list, the list contains exactly the ancestors of v, starting with the root and ending with v's parent.

Then, each time we pre-visit a vertex, before adding it to the list, we set $a(v)$ to be the k-th to last element in the list, or r if the list has size less than k.

7 (Optional) Shortest Path Between Sets

Given a undirected weighted graph G with non-negative edge weights, let $d(s, t)$ be the shortest path length from s to t. Give an algorithm that takes as input two subsets of vertices S and T and outputs $\min_{s \in S, t \in T} d(s, t)$, i.e. the shortest path from any vertex in s to any vertex in t.

Solution:

Solution 1:

Main idea: Take G and add a new vertex s^*, and add a weight 0 edge from s^* to every vertex in S. Run Dijkstra's starting from s^*, and then given the output $dist$, return $\min_{t \in T} dist(t)$.

Correctness: For any path from a vertex S to a vertex T, there is a path of the same length from s^* to the same vertex in T, in vice-versa. $\min_{t \in T} dist(t) = \min_{s \in S, t \in T} d(s, t)$.

Runtime: Running Dijkstra's takes $O((|V| + |E|) \log |V|)$.

Solution 2:

Main idea: We modify Dijkstra's, so that rather than initializing $dist(s) = 0$ for a single vertex, it initializes $dist(s) = 0$ for all $s \in S$, and also updates $dist(v)$ for all v that are neighbors of vertices in S. Our priority queue is initialized with all vertices in $V \setminus S$ instead of $V \setminus s$. Then, we run Dijkstra's as normal.

Correctness: This can be seen as equivalent to what Solution 1 does, since Solution 1 will start by setting $dist(s) = 0$ for all $s \in S$ anyway.

Runtime: Running Dijkstra's takes $O((|V| + |E|) \log |V|)$.