EPFL

Homework II, Algorithms II 2024

Solutions to many homework problems, including problems on this set, are available on the Internet, either in exactly the same formulation or with some minor perturbation. It is *not acceptable* to copy such solutions. It is hard to make strict rules on what information from the Internet you may use and hence whenever in doubt contact Michael Kapralov or Ola Svensson. You are, however, allowed to discuss problems in groups of up to three students.

1 Additive-Error Cut Sparsifiers (30 points)

In this problem we are given an undirected graph $G = (V, E)$ with n vertices and m edges. Since the number of edges is potentially very large and we are running low on storage, we are interested in storing a "smaller" version of G. Concretely, we want to construct a weighted graph $G' = (V, E', w)$ defined on the same vertex set V and containing only a subset $E' \subseteq E$ of the original edges. The weight function $w : E' \to \mathbb{R}_0$ assigns each edge of G' a positive weight.

We require that G' approximately preserves the cut sizes of the original graph G. For any subset $S \subseteq V$ of vertices, called *cut*, let $E_G(S)$ be the total *number* of edges going from S to $V \setminus S$ in G, and let $E_{G'}(S)$ be the total *weight* of edges going from S to $V \setminus S$ in G' (w.r.t. the weight function w). For a fixed constant $0 < \varepsilon \leqslant 1$, we wish to guarantee that, for all cuts $S \subseteq V$:

$$E_G(S) - \varepsilon m \leqslant E_{G'}(S) \leqslant E_G(S) + \varepsilon m, \tag{1}$$

We propose a simple algorithm to construct a graph G' such that (1) holds with high probability for all cuts: Let $p = \frac{c}{\varepsilon^2} \cdot \frac{n}{m}$ for a suitable constant $c > 0$. (You are allowed to pick the constant c as large as you need, and you can also assume $0 < p \leqslant 1$ is a valid probability.) Iterate over the edges $e \in E$, and add each e to E' with probability p. Furthermore, if e is added to E', define the corresponding weight to be $w(e) = \frac{1}{p}$.

Your task is to prove that, using this algorithm: 1. G' contains in expectation a linear amount of edges, and 2. G' is very likely to preserve to preserve *all* cuts of G approximately. Concretely, prove that:

1. The expected number of edges we store in G' is $O\left(\frac{n}{\varepsilon^2}\right)$.

2. With probability at least $1 - d^n$ (where $d < 1$ is some constant), (1) holds for *all* cuts S of G.

Hint: You may use the more convenient variants of the Chernoff inequalities listed here: `https://en.wikipedia.org/wiki/Chernoff_bound#Multiplicative_form_(relative_error)` or other similar inequalities shown in the lecture.

2   Explaining via Sketching (30 points)

Alice and Bob are planning a cheese fondue party, but they need to make some decisions about which cheeses to buy.

- Alice's input: Alice has a matrix $A \in \mathbb{R}^{n \times d}$, where:

  - Each *column* represents a different type of cheese (there are d cheeses in total).
  - Each *row* corresponds to one of Alice's n guests.
  - Each entry $A[i, j]$ is a real number that quantifies how much guest i enjoys cheese j.

  The enjoyment level is approximately additive, meaning that if a subset of cheeses is chosen, the total enjoyment for a guest is correlated with the sum of their enjoyment levels for the selected cheeses.

- Bob's input: has a vector $b \in \mathbb{R}^n$, where $b[i]$ specifies the desired level of enjoyment Alice and Bob are aiming to provide for guest i.

Bob needs to determine which subset of the d cheeses to buy in order to meet the prescribed enjoyment levels b as closely as possible. Formally, Bob wants to find a subset $C' \subseteq [d]$ such that:

$$\|A\mathbf{1}_{C'} - b\| \leqslant \min_{C \subseteq [d]} (1 + \epsilon)\|A\mathbf{1}_C - b\| \tag{2}$$

where $\mathbf{1}_C \in \{0, 1\}^d$ is a binary indicator vector representing the subset C.

However, Bob is at the supermarket and the reception is not so good, so Alice and Bob can only communicate via a bandwidth-limited channel. The goal is to design a communication protocol that minimizes the amount of data exchanged while allowing Bob to find a subset C' of cheeses that satisfies the approximation requirement with a high probability of success. It is assumed that both Alice and Bob have access to a shared source of randomness consisting of *real* numbers drawn from $\mathcal{N}(0, 1)$. Your solution should satisfy the following:

- The total number of *real* numbers sent from Alice to Bob should be $O(d^2/\epsilon^2)$.
- Bob should be able to find a subset C' that satisfied 2, with probability at least $1 - \frac{1}{2^{O(d)}}$.

Hint. Consider a sketch using the Gaussian distribution.

3   Problem 3 - The ANNS Grinch (40 points)

The goal of this implementation assignment is to empirically show that, given query access to an Approximate Near Neighbor Search (ANNS for short) data structure, it is possible to *adaptively* construct a query on which the data structure fails.

More concretely, you are asked to solve the problem `cbfgweiuhui`. This problem is interactive, in that it provides an API to query an ANNS data structure. Your submission should issue a sequence of few, carefully constructed queries to the ANNS data structure, in order to output a query point to which the data structure answers incorrectly.

3.1 Setting

For some parameters r, c, the data structure you are given access to implements the (r, c)-ANNS data structure with Manhattan distance seen in Lecture 20 of the course This data structure is initialized on a dataset P of n d-dimensional points in the hypercube $\{0, 1\}^d$. When your program issues a query $q \in \{0, 1\}^d$, the data structure gives an answer $a(q)$ which is either a point in P or the special element $\perp$. As seen in the course, the guarantee is that for any *fixed* query $q \in \{0, 1\}^d$ we have the following:

- If there exists $p \in P$ such that $\|q - p\|_1 \leqslant r$, then we have $a(q) = p'$ for some $p' \in P$ such that $\|q - p'\|_1 \leqslant cr$ with probability at least $1 - 1/n$;

- If for every $p \in P$ we have $\|q - p\|_1 > cr$, then we have $a(q) = \perp$ with probability 1.

In each test case, the underlying dataset P has the property that there exists a point $z \in P$ (called a *center point*) such that for every $p \in P \setminus \{z\}$ we have $\|z - p\|_1 \geqslant 2\lceil cr \rceil + 2$.

Let us define a *bad* query point for the ANNS as a query $q^* \in \{0, 1\}^d$ such that $\|z - q^*\|_1 \leqslant r$ but $a(q^*) = \perp$.

Goal. Your program is given as input the dimension d, the ANNS parameters r, c, the number of points in the dataset n, the number of queries N it is allowed to make, the center point z, and it is given query access to the ANNS data structure. Your submission should output a *bad* query point $q^* \in \{0, 1\}^d$, while making at most N queries to the ANNS data structure.

3.2 Algorithm

In this section we describe the algorithm that your submission should implement.

Algorithm Idea. The algorithm first samples a point q which is sufficiently close to z (i.e., inside the ball of radius r around z). If $a(q) = \perp$, the algorithm has found a query point for which there is a close point in the dataset, and yet the data structure thinks that there is no point within a radius of cr. Otherwise, $a(q) = z$, since, by the ANNS data structure, we know that $\|z - q\|_1 \leqslant r$ and $\|p - q\|_1 \geqslant \|z - p\|_1 - \|z - q\|_1 > cr$. The algorithm then tries to iteratively find a coordinate to flip in q (hence moving it away from z by one bit at a time), so that this process reaches a point where $a(q) = \perp$ before "leaving" the ball of radius r around z. Once this point is reached, the algorithm terminates.

How do we decide which coordinate to flip each time? Suppose we are at a point q such that $a(q) = z$ and $\|z - q\|_1 < r$. Let

$$M(z, q) = \{i \in [d] : q_i = z_i\}$$

be the set of coordinates in which z and q match. Then, randomly sample $w = \lceil cr \rceil + 1 - \|z - q\|_1$ coordinates $I = \{i_1, \ldots, i_w\}$ from $M(z, q)$. For $j \in [w]$, let u^j be the point defined as $u_i^j = 1 - q_i$ for all $i \in \{i_1, \ldots, i_j\}$ and $u_i^j = q_i$ otherwise. For convenience, let $u_0 = q$ and $\tilde{q} = u_w$.

Note that $\|p - \tilde{q}\|_1 \geqslant \lceil cr \rceil + 1$ for every $p \in P$, since we defined $\tilde{q}$ to be at a distance of exactly $\lceil cr \rceil + 1$ from z, and by the structure of P, we know $\|p - \tilde{q}\|_1 \geqslant \|z - p\|_1 - \|z - \tilde{q}\|_1 > cr$ for every $p \in P \setminus \{z\}$. Therefore, querying $\tilde{q}$ returns $\perp$.

We then have two points $q, \tilde{q}$ such that $a(q) = z$ and $a(\tilde{q}) = \perp$. Consider the path $(u_0, \ldots, u_w)$ of points on the hypercube connecting q and $\tilde{q}$. There must be a point on this path where we transition from $a(u_j) = z$ to $a(u_j) = \perp$. Let $j^* \in [w]$ be that point, i.e., $a(u_{j^*}) = z$ and

$a(u_{j^*+1}) = \perp$. Hence, $i_{j^*} \in I$ is the coordinate where u_{j^*} and u_{j^*+1} differ. Note that i_{j^*} is also a coordinate where z and u_{j^*+1} differ, since $i_{j^*} \in I \subseteq M(z,q)$. Thus, we update q by flipping its i_{j^*}-th coordinate. [1] We then repeat.

The above process is randomized, and we may reach the boundary of the ball of radius r around z before finding a point q with $a(q) = \perp$. To increase the success probability, we repeat the entire process for several trials.

Formal Description. The algorithm steps are summarized below, using the notation introduced above. The parameter T is a constant representing the number of trials. Step 2.b.ii should be implemented by making queries to the data structure.

1. Let $\mu = \min\{r, \lceil 2\mathrm{e}^2 \cdot (\ln n + 1) \rceil\}$.

2. For $t = 1, \dots, T$, do the following.

   (a) Sample q uniformly from $\{x \in \{0,1\}^d : \|x - z\| = r - \mu\}$.
   (b) While $a(q) \neq \perp$ and $\|z - q\|_1 < r$, do the following.
       i. Sample a set $I = \{i_1, \dots, i_w\} \subseteq M(z,q)$ of size $w = \lceil cr \rceil + 1 - \|z - q\|_1$.
       ii. Find $j^* \in [w]$ such that $a(u_{j^*}) = z$ and $a(u_{j^*+1}) = \perp$.
       iii. Update q by flipping its i_{j^*}-th coordinate.
   (c) If $a(q) = \perp$ and $\|z - q\|_1 \leqslant r$, then return q and halt.

Example. Consider running the algorithm above on a very simple example with $d = 10$, $r = 2$, $c = 1.5$, $n = 3$, with $z = (0,0,0,0,0,0,0,0,0,0)$ (in such a simple case, the number N of allowed queries will be larger than n, say $N = 60$). Then P consits of z plus two more 10-dimensional points at distance at least 8 from z. Since in this setting $\mu = r$, the first point q must be z, so for sure $a(q) \neq \perp$ in the beginning. The set I will be of size 4. After finding j^* along the path between q and $\tilde{q}$ (i.e. between u_0 and u_w) and flipping the i_{j^*}-th coordinate of q, we obtain a new point q at distance 1 from z. Since we have not reached the boundary of the ball of radius 2 around z yet, we have room to flip one more bit in q. We then repeat the while loop once more, hoping that after this we will have a point with $a(q) = \perp$.

3.3 Theory question: is this a contradiction? (5 points)

Part 1 (2 points). Prove that for any *fixed* sequence of N queries $Q = \{q_1, q_2, ..., q_N\}$ to the ANNS, the probability that the data structure fails to answer correctly any of the queries in Q is at most N/n.

Part 2 (3 points). On the other hand, the algorithm you are asked to implement in this question should find a *bad* query (as defined in Section 3.1) after querying the ANNS data structure on N queries. Explain why this is not in contradiction with Part 1 above even when $N \ll n$.

---

[1] Recall the construction of the ANNS data structure seen in Lecture 20, where we have ℓ hash tables, each defined by a different function $h^t : \{0,1\}^d \to \{0,1\}^k$ that maps any x to $(x_i)_{i \in C_t}$ for some $C_t \subseteq [d]$ of size k. The idea is that by flipping the i_{j^*}-th coordinate in q, we reduce the number of hash tables where $h^t(z) = h^t(q)$. Note that by construction, $h^t(u_{j^*+1}) \neq h^t(z)$ for each h^t, while there exists an $h^{\hat{t}}$ such that $h^{\hat{t}}(u_{j^*}) = h^{\hat{t}}(z)$. It must then be the case that $i_{j^*} \in C_{\hat{t}}$, so letting q' be q with the i_{j^*}-th coordinate flipped, we have $h^{\hat{t}}(q') \neq h^{\hat{t}}(z)$. Since $h^{\hat{t}}(u_{j^*}) = h^{\hat{t}}(z)$ implies $h^{\hat{t}}(q) = h^{\hat{t}}(z)$ and $h^{\hat{t}}(q') \neq h^{\hat{t}}(z)$, we reduce the number of hash tables where the query point hashes to the same bucket as z.

3.4 Implementation question (35 points)

Part 1 (20 points). Solve the problem `cbfgweiuhui-pt-1` on Codeforces. To do so, it is sufficient to implement step 2.b.ii of the algorithm from Section 3.2 by scanning the entire path linearly.

Part 2 (15 points). Solve the problem `cbfgweiuhui-pt-2` on Codeforces. This is the same problem as above, except that the number of allowed queries N is smaller. Therefore, scanning the entire path linearly is insufficient to pass all test cases. Instead, you should implement step 2.b.ii of the algorithm from Section 3.2 using a well-known divide-and-conquer algorithm.

Please see the PDF for instructions on how to submit your solution to the online judge and for other important remarks regarding interactive problems and grading.

By no means you are required to code an ANNS data structure, and you can debug your code by unit-testing each subroutine that you implement.

However, if you want to be able to see your entire algorithm running on your machine, you can find on Moodle a sample C++ source code that implements an ANNS data structure like the one seen in Lecture 20. When running this code on your machine, you should use the class `offlineANNS` for the ANNS data structure. In this case, the input format is the same as the one on Codeforces, with the difference that the center point z is then followed by $n - 1$ more lines containing the rest of the points in the dataset. You can also find a sample input on Moodle. If you are coding in Python and wish to translate the sample code provided, you should represent points by objects other than lists, since these are not hashable. You can use tuples. Alternatively, you can use `numpy` arrays and then use the method `tobytes()` when hashing them. It is not an issue to have `numpy` arrays because the part of your code implementing the ANNS data structure is not to be submitted to Codeforces.