CS 170 Homework 13

Due **5/2/2022, at 10:00 pm (grace period until 11:59pm)**

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".

2 Duplicate Streams

We have two streams of bits, $A = a_1, a_2, \ldots, a_n$ and $B = b_1, b_2, \ldots, b_n$ (that is, each a_i, b_i is either 0 or 1). We want to design an algorithm that streams A, then streams B, and we want to detect if $A = B$ (that is,$a_1 = b_1, a_2 = b_2, \ldots$ are all true) or not. Your algorithms know when A ends and B begins, and you may assume that the index of each bit is provided to the algorithm along with the bit (i.e., it doesn't need to spend additional memory keeping track of the current index).

(a) Give an algorithm using $O(1)$ bits of memory that outputs "True" if $A = B$ and "False" if A, B differ in an odd number of positions (if A, B differ in an even number of positions, we don't care what it outputs).

(b) A hash function family from $[n]$ to $[m]$, is k-wise independent if for any k distinct inputs $x_1, x_2, \ldots x_k$ to the hash function and h chosen uniformly at random from this family, $h(x_1), h(x_2), \ldots, h(x_k)$ is equally likely to be any of the m^k possible k-tuples of values in $[m]$.

Give a streaming algorithm using a hash function sampled from a k-wise hash function family from $[n]$ to $[m]$ (for an m of our choice) and $O(1)$ additional bits of memory that outputs "True" if $A = B$ and outputs "False" with probability at least $1/2$ if A, B differ in at most k positions. (Hint: Use your approach in part a on a subset of the stream).

(c) Give a randomized algorithm using $O(\frac{n}{k} \log n)$ bits of memory that outputs "True" if $A = B$ and outputs "False" with at least constant probability if A, B differ in at least k positions.

(d) Suppose it takes $O(k \log n)$ bits of memory to store a hash function from a k-wise independent family from $[n]$ to $[m]$. By combining the previous two parts, what memory requirement do we get for distinguishing $A = B$ and $A \neq B$ with at least constant probability, regardless of how many positions they differ in?

Solution:

(a) Count the total number of 1s in the stream, mod 2. This takes 1 bit of memory per stream. Two streams that differ in an odd number of positions differ in the number of 1s by an odd amount, so this will always distinguish two streams differing in an odd number of positions.

(We can actually get this down to 1 bit by just counting the total number of 1s in A and B. This will be even if they are the same and odd if they differ in an odd number of positions)

(b) We sample a hash function h from a k-wise independent family from $[n]$ to $\{0,1\}$. We keep track of the index of the element we're currently processing. We count the total $\mod 2$ of a_i, b_i over all indices such that $h(i) = 1$. This is effectively running the algorithm from part a, but only on indices such that $h(i) = 1$. Given A, B that differ in at most k indices, consider these indices. By the k-wise independent property, each independently has a $1/2$ chance of having $h(i) = 1$. In turn, the number of these indices that is considered by the algorithm is $Binom(k, 1/2)$, which is odd with $1/2$ probability, at which point we can reuse the analysis from part a.

(If one does not know $Binom(k, 1/2)$ is odd with $1/2$ probability, you can instead use the following argument: Consider all but one of the indices in which in A, B differ. Regardless of whether the number of these indices that hashed to 1 is even or odd, the last index in which A, B differ will hash to 1 with probability $1/2$, so the total number of these indices that hashes 1 to is odd with $1/2$ probability).

(c) We sample n/k independent and uniformly random values in $i_1, i_2, \ldots i_{n/k} \in \{1, 2, \ldots, n\}$ before processing either stream and store them (which takes $O(n/k \log n)$ space). We store all a_{i_j}, b_{i_j} and check if they are equal at the end of the stream. With probability at least k/n each a_{i_j}, b_{i_j} pair is not equal. So the overall failure probability is at most $(1 - k/n)^{n/k}$ (which is $\approx 1/e$).

(d) We can run part b and c in parallel for the same arbitrary value of k, using $O(k \log n + \frac{n}{k} \log n)$ bits of memory. Choosing $k = \sqrt{n}$, we get a space requirement of $O(\sqrt{n} \log n)$.

3 Multiway Cut

In the multiway cut problem, we are given a graph $G(V, E)$ with k special vertices $s_1, s_2 \dots s_k$. Our goal is to find the smallest set of edges F which, when removed from the graph, disconnect the graph into at least k components, where each s_i is in a different component. When $k = 2$, this is exactly the min s-t cut problem, but if $k \geq 3$ the problem becomes NP-hard.

Consider the following algorithm: Let F_i be the set of edges in the minimum cut with s_i on one side and all other special vertices on the other side. Output F, the union of all F_i. Note that this is a multiway cut because removing F_i from G isolates s_i in its own component.

(a) Explain how each F_i can be found in polynomial time.

(b) Let F^* be the smallest multiway cut. Consider the components that removing F^* disconnects G into, and let C_i be the set of vertices in the component with s_i. Let F_i^* be the set of edges in F^* with exactly one endpoint in C_i. How many different F_i^* does each edge in F^* appear in? Which is larger: F_i and F_i^*?

(c) Using your answer to the previous part, show that $|F| \leq 2|F^*|$. (Challenge: How could you modify this algorithm to output F such that $|F| \leq (2 - \frac{2}{k})|F^*|$?)

(As an aside, consider the minimum k-cut problem, where we want to find the smallest set of edges F whose removal disconnects the graph into at least k components. The following greedy algorithm for minimum k-cut gets a $(2 - \frac{2}{k})$-approximation: Initialize F to the empty set. While $G(V, E - F)$ has less than k components, find the minimum cut within each component of $G(V, E - F)$, and add the edges in the smallest of these cuts to F. Showing this is a $(2 - \frac{2}{k})$-approximation is fairly difficult.)

Solution:

(a) Consider adding a vertex t to the graph and connecting t to all special vertices except s_i with infinite capacity edges. Then F_i is the minimum s_i-t cut, which we know how to find in polynomial time.

(b) Each edge in F^* appears in exactly two of the sets F_i^*.

Note that F_i^* is the set of edges in a cut which disconnects s_i from the other special vertices. Then by definition F_i has fewer edges than F_i^* since F_i is the minimum cut disconnecting s_i from all other special vertices.

(c) We combine the answers to the previous part and note that F's size is at most the total size of all F_i to get:

$$|F| \leq \sum_i |F_i| \leq \sum_i |F_i^*| = 2|F^*|$$

To get the $(2 - \frac{2}{k})$-approximation, after computing all F_i, we instead output F as the union of all F_i except for the one with the most edges. Let this be F_j. This is still a multiway cut because each s_j is still disconnected from all other s_i. Then:

$$|F| \leq \sum_{i \neq j} |F_i| \leq (1 - \frac{1}{k}) \sum_i |F_i| \leq (1 - \frac{1}{k}) \sum_i |F_i^*| = (2 - \frac{2}{k})|F^*|$$

4 $\sqrt{n}$ coloring

(a) Let G be a graph of maximum degree Δ. Show that G is $(\Delta + 1)$-colorable.

(b) Suppose G is a 3-colorable graph. Let v be any vertex in G. Show that the graph induced on the neighborhood of v is 2-colorable. *Clarification: the graph induced on the neighborhood of v refers to the subgraph of G obtained from the vertex set V' comprising vertices adjacent to v (but not v itself) and edge set comprising all edges of G with both endpoints in V'.*

(c) Give a polynomial time algorithm that takes in a 3-colorable n-vertex graph G as input and outputs a valid coloring of its vertices using $O(\sqrt{n})$ colors. Prove that your algorithm is correct and also analyze its runtime.
Hint: think of an algorithm that first colors "high-degree" vertices and their neighborhoods, and then colors the rest of the graph. The previous two parts might be useful.

Solution:

(a) Let the color palette be $[\Delta + 1]$. While there is a vertex with an unassigned color, give it a color that is distinct from the color assigned to any of its neighbors (such a color always exists since we have a palette of size $\Delta + 1$ but only at most Δ neighbors).

(b) In an induced graph, we only care about the direct vertices adjacent to v and v itself. In any 3-coloring of G, v must have a different color from its neighborhood and therefore the neighborhood must be 2-colorable.

(c) While there is a vertex of degree-$\geq \sqrt{n}$, choose the vertex v, pick 3 colors, use one color to color v, and the remaining 2 colors to color the neighborhood of v (2-coloring is an easy problem). Never use these colors ever again and delete v and its neighborhood from the graph. Since each step in the while loop deletes at least $\sqrt{n}$ vertices, there can be at most $\sqrt{n}$ iterations. This uses only $3(\sqrt{n} + 1)$ colors. After this while loop is done we will be left with a graph with max degree at most $\sqrt{n}$. We can $\sqrt{n} + 1$ color with fresh colors this using the greedy strategy from the solution to part a. The total number of colors used is $O(\sqrt{n})$.

5 (Optional) One-Sided Error and Las Vegas Algorithms

An RP algorithm is a randomized algorithm that runs in polynomial time and always gives the correct answer when the correct answer is 'NO', but only gives the correct answer with probability greater than 1/2 when the correct answer is 'YES'.

(a) Prove that every problem in RP is in NP (i.e., show that $RP \subseteq NP$). *Hint: it may be helpful to view a randomized algorithm $R(x)$ as a deterministic algorithm $A(x, r)$ where x is the input and r is the result of the 'coin flips' which the algorithm uses for its randomness.*

(b) A *Las Vegas Algorithm* is a random algorithm which always gives the right solution, but whose runtime is random. A ZPP algorithm is a Las Vegas algorithm which runs in expected polynomial time (ZPP stands for Zero-Error Probabilistic Polytime). Prove that if a problem has a ZPP algorithm, then it has an RP algorithm. *Hint: Since we have not told you anything about the structure of the ZPP algorithm, your RP algorithm can only use it by running it somehow. What can you do to make sure the runtime is bounded? Use Markov's inequality.*

Solution:

(a) Assume we have some problem that is in RP and let A be an algorithm that solves this problem. If A is given an input x such that the correct answer given input x is 'YES', then A will return 'YES' will probability greater than $1/2$. We want to use A to construct an NP algorithm for x.

Randomized algorithms run like regular algorithms, but will occasionally use random coin-flips to make decisions. For any randomized algorithm B, we can build a deterministic algorithm B' that takes the outcome of those coin-flips beforehand and runs the same computation as B. Let A' be that deterministic algorithm for A.

Since A is an RP algorithm, there must be a poly-size sequence of coin-flip outcomes $\vec{b} = b_1, \ldots, b_k \in \{0,1\}^k$ such that A returns 'YES' given input x and outcomes $\vec{b}$. If we treat $\vec{b}$ as the 'solution' to x, we can see that A' is an NP algorithm for the given problem. Thus every problem in RP is in NP.

(b) Let A be any ZPP algorithm and assume A runs in expected time at most n^k for some constant k. We construct an RP algorithm A' that given input x of size n, does the following:

- Run A for $2n^k$ steps, then stop.
- If A returns 'YES' in that time, return 'YES'
- If A returns 'NO' in that time, return 'NO'
- Otherwise, return 'NO'

How do we know A' is an RP algorithm? First of all, if the correct answer on input x is 'NO', then either A will stop and A' will say 'NO' or A will not stop and A' will still say 'NO'. So A' will always give the right answer when the right answer is 'NO'.

No assume the correct answer is 'YES'. If A stops in time, A' will give the correct answer. But if A does not stop in time, then A' will give the wrong answer. So we want to bound the probability that A does not stop in time.

Let X be a random variable representing the number of steps A will take on input x. We know that $E[X] \leq n^k$. By Markov's inequality this gives us

$$Pr[X \geq a] \leq \frac{n^k}{a}$$

If we set $a = 2n^k$, this gives us

$$Pr[X \geq 2n^k] \leq \frac{n^k}{2n^k} = \frac{1}{2}$$

Thus if the correct answer on input x is 'YES', A' will give the correct answer with probability greater than $1/2$.

So A' is an RP algorithm.