CS5234: Combinatorial and Graph Algorithms

Problem Set 1

Due: August 23th, 6:30pm

Instructions. The *exercises* at the beginning of the problem set do not have to be submitted—they are for your review. This problem set contains one main problem, which involves coming up with a fast algorithm to decide whether a 2-dimensional data set can be classified into two sets. The goal is to develop an efficient algorithm, and the solution relies on the sampling techniques we covered in class in Week 1. (I think it is really neat how much we can determine about a data set, even when we are only willing to look at a small part of it!)

- Please submit the problem set on LumiNUS in the appropriate folder. (Typing the solution using latex is recommended.) If you want to do the problem set by hand, please submit it at the beginning of class.
- Start each problem on a separate page.
- If you submit the problem set on paper, make sure your name is on each sheet of paper (and legible).
- If you submit the problem set on paper, staple the pages together.

Remember, that when a question asks for an algorithm, you should:

- First, give an overview of your answer. Think of this as the executive summary.
- Second, describe your algorithm in English, giving pseudocode if helpful.
- Third, give an example showing how your algorithm works. Draw a picture.

You may then give a proof of correctness, or explanation, of why your algorithm is correct, an analysis of the running time, and/or an analysis of the approximation ratio, depending on what the question is asking for.

Advice. Start the problem set early—questions may take time to think about. Come talk to me about the questions. Talk to other students about the problems.

Collaboration Policy. The submitted solution must be your own unique work. You may discuss your high-level approach and strategy with others, but you must then: (i) destroy any notes; (ii) spend 30 minutes on facebook or some other non-technical activity; (iii) write up the solution on your own; (iv) list all your collaborators. Similarly, you may use the internet to learn basic material, but do not search for answers to the problem set questions. You may not use any solutions that you find elsewhere, e.g. on the internet. Any similarity to other students' submissions will be treated as cheating.

Exercises and Review (*Do not submit.*)

Exercise 1. Imagine you are taking a poll of students at NUS. (Alas, you do not know exactly how many students there are at NUS.) Your goal is to determine what fraction of students like Starbucks. The polling error is allowed to be 2%, i.e., your answer should be within 2% of the correct answer. Your survey should be correct (i.e., within the desired polling error) with probability 90%. The university will give you a list of randomly students to call or e-mail. How big a list of students do you need to get a sufficiently precise answer? Give an exact (integer) numeric answer. (Notice that the total number of students at NUS does not matter!)

Exercise 2. Imagine you have an array $A[1\,..\,n]$. Each value in the array is an integer between 1 and M. Consider the following algorithm for finding the approximate sum of the values in the array: Fix $s = 1/\epsilon^2$. Your boss claims that this is a good algorithm, and gives the following proof:

Algorithm 1: Sum(A, n, s)

```
1 sum = 0
2 repeat s times
3     Choose a random i ∈ [1, n].
4     sum = sum + A[i].
5 return n(sum/s)
```

Let x_i be the value of the ith random sample, and $X = \sum(x_i)$. Let $A = \sum_i A[i]/n$, i.e., A is the average value and nA is the sum of the array (i.e., what we want to find). We know that $\mathrm{E}[X] = sA$. Then, using a Hoeffding Bound, we know that:

$$\Pr[|X - \mathrm{E}[X]| \geq \epsilon \mathrm{E}[X]] = \Pr[|X - sA| \geq \epsilon sA] \leq 2e^{-2(sA\epsilon)^2/s} .$$

We know that $A \geq 1$, so by choosing $s = 1/\epsilon^2$, we conclude that:

$$\begin{aligned} 2e^{-2(sA\epsilon)^2/s} &\leq 2e^{-2s\epsilon^2} \\ &\leq 2e^{-2} \end{aligned}$$

Thus, the probability of error is $\leq 1/3$. Thus, with probability at least $2/3$, $|X - sA| < \epsilon sA$, which implies that: $|nX/s - nA| < \epsilon nA$, i.e.,

$$nA(1-\epsilon) \leq sum \leq nA(1+\epsilon) .$$

We thus conclude that the algorithm returns a $(1 \pm \epsilon)$ estimate of the sum of the values in the array.

What is wrong with this algorithm and proof? Can you give an example of an array A where this algorithm will almost certainly give the wrong answer?

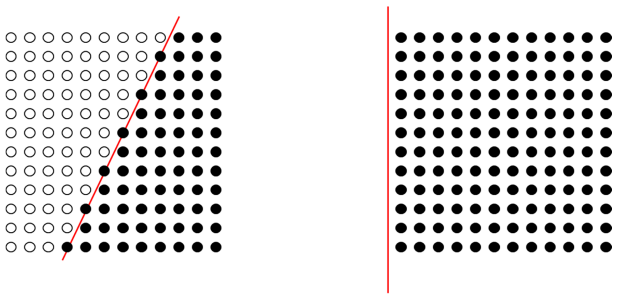
Standard Problems (to be submitted)

Problem 1. Classifying Pixels

A classic problem in data analysis is classification: given a set of points where each point is either a 0 or a 1, can we draw a line (or a hyperplane) that separates the points into two sets containing only (or mostly) 0's and only (or mostly) 1's.

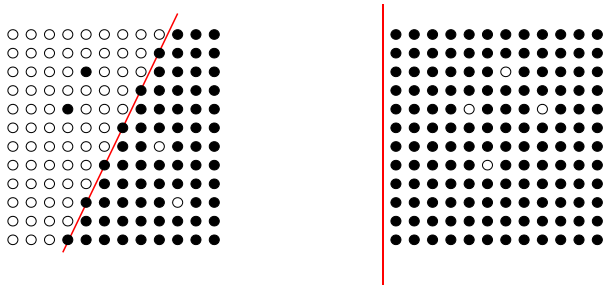
Here we look at a simplified version of the problem. Imagine you are given a two-dimensional n by n array A where $A[i, j]$ is either 0 or 1. If we think of the array A as a 2-dimensional display of pixels, we want to decide whether it is possible to draw a line separating all the 0's from all the 1's.

We say that array A is divisible if it is possible to draw such a line, where one side of the line is all 0's and the other side of the line is all 1's. Here are two examples of divisible arrays, where a white circle represents zero and a black circle represents one. The red line represents the separator.



Unfortunately, this might be very slow: we really have to look at all $\Theta(n^2)$ array locations to decide whether it is possible. Instead we ask whether an array A is either divisible or far from divisible. We say that array A if ϵ-close to being divisible if we only need to change $\epsilon \cdot n^2$ entries to make it divisible. (Equivalently, there are only $\epsilon \cdot n^2$ points on the wrong side of the separating line.)

In the following example, both arrays are 0.03-close to divisible: the array is 12 by 12 (i.e., 144 pixels), and each contains 4 mislabeled pixels. Since $4/144 < 0.03$, this is 0.03-close to divisible.



If array A is not ϵ-close to divisible, then it is ϵ-far from divisible. (Recall that it is only ϵ-far from divisible if it is not ϵ-close for any possible partition.)

Our goal in this problem is to develop an algorithm that has the following properties:

- If array A is divisible, then it returns TRUE.
- If array A is ϵ-far from divisible, then it returns FALSE.
- Otherwise, it can return either true or false.

The algorithm should return the correct answer with probability at least $2/3$. In the following parts, we will develop such an algorithm and prove it correct. We will measure the running time by counting how many times the array A is queried. All other computational costs will be ignored.

Problem 1.a. First, look at the four sides of the square: top, bottom, left, right. We say that a side of the square is *mismatched* if the endpoints are a different color. For example, in the first array given above (on the left), the top and bottom edges are mismatched while the left and right edges are not mismatched (i.e., they are matched).

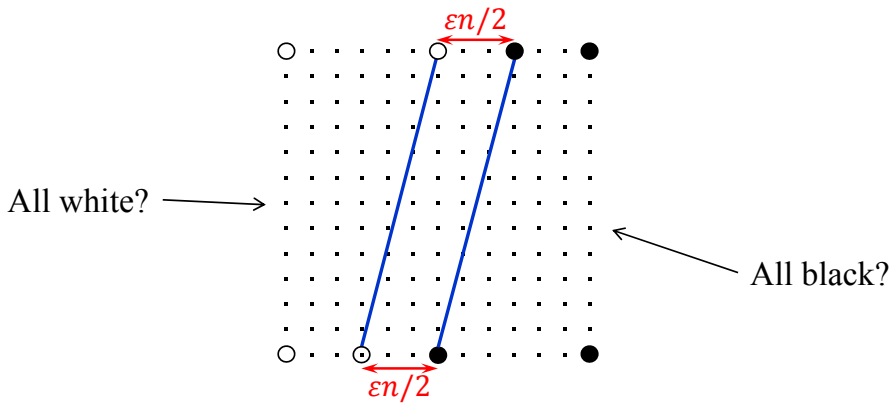
Prove that for every array A, one of the following three cases holds: (i) 0 edges are mismatched, (ii) 2 edges are mismatched, or (iii) 4 edges are mismatched. (That is, you cannot have only one edge mismatched or three edges mismatched.)

Problem 1.b. Prove that if there are four edges mismatched, then A is not divisible.

Problem 1.c. Assume A has zero edges mismatched and all the corners are zero. There is only ONE possible array A that is divisible and satisfies this condition! (Hint: think about where you would have to draw a separator if A is divisible.)

Give an algorithm such that if A has zero edges mismatched, then it returns TRUE if A is divisible and FALSE if A is ϵ-far from divisible. (Hint: you can use an algorithm we discussed in class.) Prove (briefly) that your algorithm is correct. Your algorithm should run in sublinear time that does not depend on n (only on ϵ).

Problem 1.d. Assume A has two edges that are mismatched. Give an algorithm to find two points on each mismatched side that are of opposite colors and are within distance $\epsilon n/2$ of each other, as in the picture below.



On a mismatched side with corners x_1 and x_2, your algorithm will find points y_1 and y_2. If the points appear in order x_1, y_1, y_2, x_2, then x_1 and y_1 should be the same color, and x_2 and y_2 should be the same color. See the diagram for an example.

(Hint: binary search.) What is the running time of your algorithm? (The running time of your algorithm should not depend on n.)

Problem 1.e. Assume A is divisible and has two mismatched edges. In the picture above, we have identified two points on each mismatched side that are of opposite colors and are within distance $\epsilon n/2$ of each other. Imagine we connect these lines as shown above. If A is divisible, then it must be the case that all the points to the left of the line connecting the white points are white, and all the points to the right of the line connecting the black points are black.

Give an algorithm for the situation where A has two mismatched edges, and we have already identified two nearby points for each mismatched edge as described above. The algorithm should guarantee that:

- If A is divisible, then it returns TRUE.
- If A is ϵ-far from divisible, then it returns FALSE.

Prove that your algorithm is correct. What is the running time of your algorithm?

Problem 1.f. Given all of the above, briefly sketch out the entire algorithm. What is the overall running time of the algorithm? (Remember, we only care about how many times array A is accessed; other computational costs can be ignored.)