

Homework 2

Lecturer: Ronitt Rubinfeld

Due Date: March 5, 2020

Homework guidelines: You may work with other students, as long as (1) they have not yet solved the problem, (2) you write down the names of all other students with which you discussed the problem, and (3) you write up the solution on your own. No points will be deducted, no matter how many people you talk to, as long as you are honest. If you already knew the answer to one of the problems (call these "famous" problems), then let me know that in your solution writeup – it will not affect your score, but will help me in the future. It's ok to look up famous sums and inequalities that help you to solve the problem, but don't look up an entire solution.

The following problem is **NOT** to be turned in.

- Consider an algorithm that returns a *random satisfying assignment* to any given DNF with n variables and $\text{poly}(n)$ clauses. Every satisfying assignment must be returned with non-zero probability (not necessarily uniform). Given an input DNF formula ϕ on n variables, the algorithm must satisfy the following requirements:

- Every satisfying assignment $\{x_1, x_2, \dots, x_n\} \in \{0, 1\}^n$, must have non-zero probability of being output (need not be uniform).
- Assignments that do not satisfy ϕ , have zero probability of being output

Does there exist such an algorithm that uses at most \sqrt{n} random bits?

The following problems are to be turned in.

1. Suppose you are given n boxes, each containing between 1 and k balls, where $n \gg k$. However, you do not know the number of balls in each box. All you can do is pick one of the boxes, and count the number of balls inside (assume this takes $O(1)$ time). The goal of this problem is to design an algorithm to choose an *uniformly random ball*.
 - (a) The naive algorithm of viewing each box uses $O(n)$ run-time. Show that this task can be performed with $O(k)$ expected run-time.
 - (b) What if counting the number of balls in a box that contains d balls, actually takes $O(d)$ time, instead of $O(1)$? In this case, what is the best run-time you can achieve?

2. **(Pairwise independence)** A *pairwise independent* space on n variables is a subset $S \subseteq \{-1, +1\}^n$ such that for every $i \neq j \in \{1, \dots, n\}$, if \mathbf{x} is uniformly-random element of S then $(\mathbf{x}_i, \mathbf{x}_j)$ is a pair of independent bits each drawn uniformly from $\{-1, +1\}$. The size of the pairwise independent space is $|S|$.
 - (a) In class, we stated without proof, a construction which generates $n = 2^l - 1$ pairwise independent bits, from $l = \log(n + 1)$ truly random bits¹. In other words, this is a

¹Note that we used $\{0, 1\}$ random bits in class, and considered the *parity* of all n possible non-empty subsets. Here, we will use $\{-1, +1\}$ random bits, and consider the *product* of the elements in all possible subsets. You can easily check that this leads to the same outcome when replacing 0 with -1 .

pairwise independent space of size $2^l = n + 1$. Prove that the bits are indeed pairwise independent.

Any algorithm that generates n pairwise independent random bits, will sample from a pairwise independent space $S = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(s)}\}$, where $s = |S|$.

Arrange the vectors in S into a matrix S , with rows corresponding to $\mathbf{x}^{(i)}$.

(b) Interpret the condition of pairwise independence in terms of the columns of S . Using this, prove that S must contain at least n vectors (think about the rank of S).

(c) Show that this implies that construction is optimal, in the sense that every pairwise independent space on n variables, requires at least $\log n$ truly random bits.

You can solve this part by assuming the statement of (b).

3. Let $\{M_1, M_2, \dots, M_k\}$ be subsets of $[1 \dots n]$, that is $M_i \subseteq [n]^2$. Given a set S containing N integers, we randomly assign weights $w_x \in_{\mathcal{R}} S$, to each $x \in [n]$. We also define the weight of M_i to be the sum of the weights of its elements i.e. $w(M_i) = \sum_{x \in M_i} w_x$. The goal of this problem is to prove that the minimum weight set is likely unique. To clarify, a unique minimum weight M_i is one, such that, for every other M_j , we have $w(M_j) > w(M_i)$. We will prove that

$$\mathbb{P}[\text{There is a unique } w(M_i) \text{ of minimum weight}] \geq 1 - \frac{n}{|S|}.$$

The astonishing fact is that this holds, even if the collection of sets $\{M_1, M_2, \dots, M_k\}$ comprises of all 2^n subsets of $[n]$. We begin by defining the quantity $\alpha(x)$ for all $x \in [n]$.

$$\alpha(x) = \min_{i \in [k], x \notin M_i} w(M_i) - \min_{i \in [k], x \in M_i} w(M_i \setminus \{x\})$$

Note that above, we use the set-minus notation $M_i \setminus \{x\}$ to denote the set M_i with the element x removed. In other words, $w(M_i \setminus \{x\}) = w(M_i) - w_x$.

- (a) Calculate the probability (over the choice of weights) that $\alpha(x)$ is equal to $w(x)$ for a *specific* $x \in [n]$. Next, upper bound the probability that this happens for *some* $x \in [n]$. Specifically, show that:

$$\mathbb{P}[\exists x \text{ such that } \alpha(x) = w(x)] \leq \frac{n}{|S|}$$

- (b) Now, we show that it is unlikely to that two distinct M_j and M_l have the same minimum weight (compared to all other $w(M_i)$ where $i \in [k]$). Apply the result from part (a) for some suitable x , to obtain the main result:

$$\mathbb{P}[\text{There is a unique } w(M_i) \text{ of minimum weight}] \geq 1 - \frac{n}{|S|}.$$

²This notation $[n]$ stands for the set of integers $\{1, 2, \dots, n\}$.

4. **Parallel Perfect Matchings:** In class, we used PIT to check whether a bipartite graph G contains a perfect matching. In this problem, we will think about the problem of actually finding a perfect matching. Sequentially, the problem of finding a perfect matchings in bipartite graphs can be solved using flows, but as we mentioned in class, we are interested in a parallel algorithm. The goal of this problem is to show that a perfect matching can be computed in parallel, by a polynomial number of processors, in $poly(\log n)$ time. For the purpose of this assignment, you may assume that there is a shared memory, and locations within this memory can be read and written by any of the processors in $\mathcal{O}(1)$ time.

Importantly, you can use the fact that arithmetic operations on n bit numbers, can be performed in $poly(\log n)$ time, using $poly(n)$ processors.

- (a) To begin, let us consider a black box algorithm \mathcal{A} (we don't know how the internals of \mathcal{A} work), that checks whether a given graph G contains a perfect matching not,

$$\mathcal{A}(G) = \begin{cases} 1, & \text{if } G \text{ contains a perfect matching.} \\ 0, & \text{otherwise,} \end{cases}$$

and runs in time $T_{\mathcal{A}}^{seq}(G)$. Use this to construct a sequential algorithm \mathcal{B} , that finds a perfect matching in time $\mathcal{O}(m \cdot T_{\mathcal{A}}^{seq}(G))$.

Next, we recap the algorithm from class. Given a bipartite graph $G = (L \cup R, E)$, where $|L| = |R| = n$, $E \subseteq L \times R$ (all edges go between left and right set), and $|E| = m$, we constructed a $n \times n$ matrix A with entries:

$$A_{u,v} = \begin{cases} X_{u,v}, & \text{if } (u,v) \in E \\ 0, & \text{otherwise} \end{cases}$$

Subsequently, for each $(u,v) \in E$, we assigned *random values* to variables $X_{u,v} \leftarrow x_{u,v}$, and evaluated the determinant of the resulting matrix:

$$\text{Det}(A) = \sum_{\sigma \text{ is a permutation of } [n]} \text{sign}(\sigma) \cdot \prod_{u \in L} A_{u,\sigma(u)}$$

- (b) Interpret the value of the above expression in terms of the perfect matchings of G . What is the value when there is exactly one perfect matching? What if there are several? This should be similar to what we did in class, but note that this time, we are asking about the actual value (as opposed to whether it is just zero or non-zero). [There is no “wrong answer” to this part, but it will help you think about the rest of the problem.](#)
- (c) Assume that the graph G has *exactly one perfect matching*, and also assume that the determinant of a $n \times n$ matrix, can be computed in parallel by $poly(n)$ processors, in $poly(\log n)$ time. Combine parts (a) and (b) to obtain a parallel algorithm that finds this perfect matching in $poly(\log n)$ time.

Now, for every edge $e = (u,v) \in E$, we assign a *random integer weight* $w_{u,v}$, and define the weight of a matching to be the sum of the weights on all its edges. As we proved in the previous Problem 3, even if G contains many perfect matchings (there could be exponentially many), it probably contains only one of minimum weight.

- (d) If we assign values $2^{w_{u,v}}$ to the variables $X_{u,v}$, what can you conclude about the value of the determinant? Specifically, if the weight of the minimum weight perfect matching is w^* , how does the value of the determinant relate to 2^{w^*} ?
There is no “wrong answer” to this part, but it will help you think about the rest of the problem.
- (e) Does this property (from the previous part) still hold, if the random weights $w_{u,v}$ result in more than one minimum weight perfect matching?
- (f) How can you detect whether a specific edge e is in the *minimum weight* perfect matching or not?
- (g) Now, you are ready to put all of the pieces together, and present a *randomized* parallel algorithm for finding a perfect matching in *poly*($\log n$) time.