CSE 6512 Randomization in Computing. Fall 2023

Homework 2 Solutions

1. Let $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ be the input CNF Boolean formula on n variables. Let w_i be the weight of C_i, for $1 \le i \le m$, and $W = \sum_{i=1}^{m} w_i$. Let C be any clause in F with k literals. Give a random assignment to the variables. Under this assignment, Prob.[C is not satisfied] $= 2^{-k}$. This means that Prob.[C is satisfied] $= 1 - 2^{-k} \ge \frac{1}{2}$. As a result, the expected value of the sum of the weights of the satisfied clauses is $\ge \frac{1}{2}(w_1 + w_2 + \cdots + w_m)$. I.e., this expected value is $\ge \frac{W}{2}$.

2. We are interested in the probability of getting farther than d positions to the right. Then the probability of being at least d positions away from the origin will be twice that, because the case of going to the left is symmetrical.

   Let $X = B(n, 1/2)$ be the event that at any one step we go to the right. If we go a steps towards the right, and $n - a$ towards the left, and at the end we are farther than d away from the origin, towards the right, then $a > (n + d)/2$.

$$Prob\,[X > (n+d)/2] = Prob\left[X > \frac{n}{2}\left(1 + \frac{d}{n}\right)\right]$$
$$\epsilon = \frac{d}{n} \Rightarrow^{(Chernoff)}$$
$$Prob\,[X > (n+d)/2] < \exp\left(\frac{-d^2}{2n}\right)$$

   We want this $\le n^{-\alpha}$:

$$\exp\left(\frac{-d^2}{2n}\right) = n^{-\alpha} \Rightarrow \frac{d^2}{2n} = \alpha \ln n \Rightarrow d^2 = 2n\alpha \ln n \Rightarrow d = \sqrt{2n\alpha \ln n}$$

   In conclusion $d = \tilde{O}\left(\sqrt{n \log n}\right)$. □

3. It was shown in class that the maximum of n elements can be found in $O(1)$ time using n^2 common CRCW PRAM processors.

   Consider the case when $\epsilon = \frac{1}{2}$. Divide the elements into groups fo size $\sqrt{n}$. Assign the first $\sqrt{n}$ elements to the first n processors and the second $\sqrt{n}$ elements to the next n processors and so on. The maximum element in each group can be found in $O(1)$ time. At this stage, we have $\sqrt{n}$ elements and $n\sqrt{n}$ processors. Hence, the maximum of these elements can be found in $O(1)$ time. Total time $= O(1)$.

   Next, consider the case when $\epsilon = \frac{1}{3}$. Here, divide the elements into groups of size $n^{1/3}$. Assign the first $n^{1/3}$ elements to the first $n^{2/3}$ processors and the second $n^{1/3}$ elements to the next $n^{2/3}$ processors and so on. The maximum element of each group can be found in $O(1)$ time and using $n^{4/3}$ prceossors the maximum of these maximum elements can be found in $O(1)$ time.

For the general case, partition the input into groups with n^ϵ elements in each group. Find the maximum of each group assigning $n^{2\epsilon}$ processors to each group. This takes $O(1)$ time. Now the problem reduces to finding the maximum of $n^{1-\epsilon}$ elements. Again, partition the elements with n^ϵ elements in each group and find the maximum of each group. There will be only $n^{1-2\epsilon}$ elements left. Proceed in a similar fashion until the number of remaining elements is $\leq \sqrt{n}$. The maximum of these can be found in $O(1)$ time. Clearly, the run time of this algorithm is $O(1/\epsilon)$. This will be a constant if ϵ is a constant.

4. Let $X = k_1, k_2, \ldots, k_n$. Assume without loss of generality that the keys are distinct. Note that the right neighbor of any input key k_i is nothing but the minimum among all the input keys that are greater than k_i. Key k_i is assigned a group G_i of n processors, $1 \leq i \leq n$. The processors associated with k_i use an array $A_i[1:n]$. This array is initialized with all ∞'s. Processor j of group G_i writes k_j in $A_i[j]$ if $k_j > k_i$. After this write step that takes one parallel step, processors in G_i find the minimum of $A_i[1], A_i[2], \ldots, A_i[n]$ in $\widetilde{O}(1)$ time. This minimum is the right neighbor of k_i.

5. We will show that we can stably sort n integers in the range $[1, \sqrt{n}]$ in $O(\sqrt{n})$ time using $\sqrt{n}$ CREW PRAM processors. Using the idea of radix sorting it will follow that we can sort n integers in the range $[1, n^c]$ (for any constant c) in $O(\sqrt{n})$ time using $\sqrt{n}$ processors.

   Let $X = k_1, k_1, \ldots, k_n$ be the input sequence. Assign $\sqrt{n}$ keys per processor. In particular, the first processor gets the keys $k_1, k_2, \ldots, k_{\sqrt{n}}$; the second processor gets the keys $k_{\sqrt{n}+1}, k_{\sqrt{n}+2}, \ldots, k_{2\sqrt{n}}$; and so on.

   (a) Each processor sorts its keys using bucket sorting. This takes $O(\sqrt{n})$ time. Let $N_{i,j}$ be the number of keys of value j that processor i has, for $1 \leq i, j \leq \sqrt{n}$.

   (b) All the $\sqrt{n}$ processors perform a prefix sums computation on $N_{1,1}, N_{2,1}, \ldots, N_{\sqrt{n},1}$, $N_{1,2}, N_{2,2}, \ldots, N_{\sqrt{n},2}$, $\cdots$, $N_{1,\sqrt{n}}, N_{2,\sqrt{n}}, \ldots, N_{\sqrt{n},\sqrt{n}}$.

   (c) Each processor now uses these prefix sums values to output its keys in the sorted order.

   Since each of the above three steps takes $O(\sqrt{n})$ time, the run time of the algorithm is $O(\sqrt{n})$.

6. Assume that A and B are in common memory in successive cells. In particular, assume that A is in $M[1:n]$ and B is in $M[n+1:m+n]$.

   (a) Sort B, i.e., sort $M[n+1:n+m]$. This can be done in $\widetilde{O}(\log m)$ time using m arbitrary CRCW PRAM processors.

   (b) Assign one processor per element of A. Processor i performs a binary search in $B[n+1:n+m]$ to check if $M[i]$ is in B, for $1 \leq i \leq n$. This binary search takes $O(\log m)$ time.

   (c) In this step, we'll use an array $Q[1:2m]$. Each element of A that is also in B will be placed in a unique cell of Q. Each element of A is assigned one processor. If an element of A is in $A \cap B$, the corresponding processor will try to place the element in Q. If an element of A is not in $A \cap B$, the corresponding processor goes to sleep. If a processor π has an element that has to be placed in Q, π proceeds in rounds. It takes as many rounds as needed to successfully place its key.

In a round, π picks a random cell in Q; If this cell is occupied, it waits for the next round; If this cell is empty, it tries to write its key in the cell; Processor π reads from this cell to check if its key is there; If so, the processor goes to sleep; If not, it moves to the next round.

Probability that π succeeds in any round is $\geq 1/2$. Thus the number of rounds needed to place π'key successfully in Q is $\widetilde{O}(\log m)$, for any processor π.

(d) Use a prefix computation to compress the array $Q[1 : 2m]$ (and get rid of the empty cells). This can be done in $O(\log m)$ time using $\frac{2m}{\log m} \leq n$ processors.

The compressed array Q is $A \cap B$.

We could do steps (c) and (d) in a different way as follows. We use an array $Q[1 : m]$ initialized to all zeros. Each element of A is assigned a processor. Processor i goes to sleep if k_i is not in $A \cap B$, $1 \leq i \leq n$. Otherwise, processor i writes a 1 in $Q[j]$ if $M[i] = M[n + j]$. After this parallel write step, we assign one processor per element of B. These processors empty the cells of B that are not in $A \cap B$. A prefix sums computation is done on Q in $O(\log m)$ time using $\frac{m}{\log m}$ processors. These prefix sums are used to write the elements of $A \cap B$ in successive cells in common memory.