CS 270: Combinatorial Algorithms and Data Structures Spring 2023

Lecture 26 — April 27, 2023

Prof. Jelani Nelson *Scribe: Jelani Nelson*

1 Lower bounds and the cell probe model

Typically in this class we have assumed the *word RAM* model of computation. That is, our machine has some constant number of registers and runs a program stored in memory. The first instruction of the program is stored at some particular memory address M_0, the next one at M_1, etc. (and some instructions can be `JUMP`s, which set the program counter to some other value). These instructions all operator on *words*. A word is a basic unit of storage, which we assume is w bits. For example, the `ADD` instruction can add two words at a time (modulo 2^w), and memory addresses are w bits long (and hence the total memory of the machine is at most 2^w), etc.

How do we prove lower bounds on data structures and algorithms in this model? One of the most robust ways is to only count `LOAD` and `STORE` instructions, since these are instructions that pretty much any real machine has, whereas some of the other instructions are not necessarily universal (e.g. `POPCOUNT` and `MostSignificantBit` instructions). That is, we only count memory reads and writes. This model of complexity, where all computation is free and only reads/writes from/to memory are counted, is known as the *cell probe model* of Yao [Y78]. In the remainder of this note, we will be proving lower bounds in this model.

2 Partial sums lower bound

Consider the dynamic partial sums problem over some group G with operation '+' on an n-dimensional array A. Initially the array has all entries set to 0 (the identity element of the group).

- `update`(i, b): $A[i] \leftarrow b$
- `query`(i): return $\sum_{j=1}^{i} A[j]$

Working with a word of size w, it is natural to consider G to be the cyclic group $\mathbb{Z}_{2^w}$. There is a data structure solving this problem with $O(\lg n)$ query time and update time, by building a complete binary search tree with $[n]$ as leaves. For a node u a group element g_u is stored, maintaining the invariant that the answer to `query`(i) is the sum of all node-associated group elements on the root-to-leaf path to leaf i. This invariant can be maintained with $O(\log n)$ update and query. Note that the group does not *have* to be $\mathbb{Z}_{2^w}$. Below we show a lower bound for an even simpler problem: where the group is $\mathbb{Z}_2$ (note this problem is easier, since it is equivalent to saying we only want to find the least significant bit of the answer for $G = \mathbb{Z}_{2^w}$).

We show a lower bound for dynamic partial sums over $\mathbb{Z}_2$ due to Fredman and Saks [FS89] of $t_q = \Omega(\lg n / \lg(t_u w))$, where t_u is the update time and t_u is the query time. In particular, this implies $\max\{t_u, t_q\} = \Omega(\lg n / \lg \lg n)$. The optimal lower bound of $\max\{t_u, t_q\} = \Omega(\lg n)$ was not shown until 15 years later by Pătraşcu and Demaine [PD04] — we will not show that today.

The lower bound works as follows, and is known as the *chronogram technique*. The way we describe the chronogram technique will be in the family of what we call encoding techniques. Essentially what we say is that if we have a data structures with a bound t_q that is too small, then we could use that data structure an an encoding scheme to compress elements of some set S into $\ll \lg |S|$ bits. Clearly this is impossible by the pigeonhole principle, so t_q must be large. We now give the details. Almost all of my understanding of how this works is due to a conversation with Kasper Green Larsen. In particular, the presentation of the chronogram technique below *is* slightly different than both the original [FS89] and the treatment in Miltersen's survey [M99]. I personally find the explanation given below slightly more intuitive.

Consider a data structure $\mathcal{D}$ that works on operation sequences that look as follows. The operation sequence has n updates, followed by one uniformly random query. We group these n updates together into what we call *epochs*. Epoch 1 is the last epoch of updates (right before the query), and epoch 2 comes right before it, etc. Epoch i will be a sequence of β^i updates for some β we will choose later. Thus the number of epochs is $\lg_\beta n$.

Recall that there is an n-dimensional array A being updated in dynamic partial sums. In the updates of epoch i, we update the set of array entries with index of the form $j \cdot (n/\beta^i)$ for $j = 1, \ldots, \beta^i$. The entries are assigned independent uniform random bit values. We use $b_{i,1}, ..., b_{i,\beta^i}$ to denote these random bit values, where $b_{i,j}$ is the value assigned to the entry $j \cdot (n/\beta^i)$ during the updates of epoch i.

At the end of epoch 1, we execute a single uniformly random partial sum query, i.e. $\texttt{query}(k)$ for a uniformly random $k \in [n]$.

Now, we color the memory cells of the data structure (there are at most 2^w memory cells) after the entire sequence of updates has been processed. A memory cell c is colored with color i if its contents were changed during epoch i but not in any epoch $j < i$. That is, i is the *last* epoch in which c was updated. Let C_i denote the set of cells colored with color i. If the worst case update time is t_u, then clearly $|C_i| \leq t_u \beta^i$ for all i.

Let T denote the (random) set of memory cells probed by the uniformly random query after the n updates. Our goal is to show that:

$$\forall i,\ \mathbb{E}\,|T \cap C_i| = \Omega(1).$$

Since the C_i are disjoint, we get from linearity of expectation that

$$\mathbb{E}\,|T| \geq \sum_{i=1}^{\lg_\beta n} \mathbb{E}\,|T \cap C_i|,$$

and a lower bound of $t_q = \Omega(\log_\beta n)$ would follow.

To prove this, assume for contradiction that there exists some i such that $\mathbb{E}\,|T \cap C_i| \leq \alpha$ for some very small constant α to be determined. We will use this assumption to create an impossible encoding of $b_{i,1}, ..., b_{i,\beta^i}$. Notice that the update values for epoch i are independent of epochs $j \neq i$. Thus $H(b_{i,1}...b_{i,\beta^i}|(b_{i',\cdot})_{i' \neq i}) = \beta^i$. This means that an encoder and a decoder can share knowledge of all updates in all epochs other than i, and we would still have to write down β^i bits in an encoding. In addition to this, let $R_1, \ldots, R_{\beta^i}$ be a sequence of random variables such that R_j gives a uniform random index between $j \cdot (n/\beta^i)$ and $(j+1) \cdot (n/\beta^i) - 1$. Clearly we can also condition on these as they are independent of the update bits. The encoding argument now goes as follows:

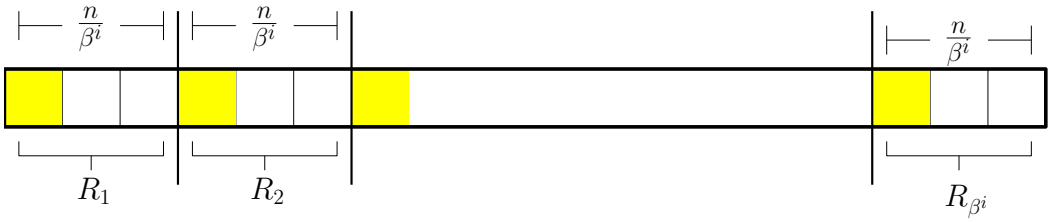

Figure 1: Above is the array A, and its entries are broken into contiguous blocks of size n/β^i. R_j is then chosen to be a uniformly random index in the jth block, i.e. R_j is chosen uniformly at random in the interval $[j \cdot (n/\beta^i), (j+1) \cdot (n/\beta^i))$. The entries in A colored in yellow correspond to the entries updated in epoch i.

We examine $R_1, \ldots, R_{\beta^i}$ and run the corresponding queries on $\mathcal{D}$. For each query, we check whether it reads a cell in C_i. If we pick a uniform random R_j, then the distribution of the query is uniform random. This means that $\mathbb{E}_{j,R_1,\ldots,R_{\beta^i}} |T(R_j) \cap C_i| \leq \alpha$ (here $T(R_j)$ is the set of cells probed by the query R_j). Thus

$$\mathop{\mathbb{E}}_{R_1,\ldots,R_{\beta^i}} \sum_{j=1}^{\beta^i} |T(R_j) \cap C_i| \leq \alpha\beta^i.$$

Thus by Markov's inequality and a union bound, we get that with probability $3/4$, the number of queries R_j which read in C_i is upper bounded by $4\alpha\beta^i < \beta^i/100$ for $\alpha = 1/400$. If this is not the case, our encoding consists of a 0-bit, follows by a naive encoding of $b_{i,1}, \ldots, b_{i,\beta^i}$, costing $\beta^i + 1$ bits.

Otherwise, we start by writing a 1-bit followed by a description of which queries amongst $R_1, \ldots, R_{\beta^i}$ read inside C_i. For those queries, we also write down their answer. This costs $\lg \binom{\beta^i}{\beta^i/100} + \beta^i/100 < \beta^i/4$ bits. Next, we write down all memory cells in $C_{i-1}, \ldots, C_1$. This costs $\sum_{j<i} O(|C_j|w) = O(\beta^{i-1} t_u w)$ bits. If we choose $\beta = (t_u w)^2$, this is $o(\beta^i)$ bits.

The expected size of the encoding is less than $1 + (3/4)(\beta^i/4 + o(\beta^i))(3/4) + (1/4)\beta^i$ bits, thus by Shannon's source coding theorem, the entropy of the encoded message, M, is $H(M) < \beta^i/2$ bits.

To derive the contradiction, we need to show that this encoded information is enough to (almost) fully recover $(b_{i,j})_{j=1}^{\beta^i}$. We will show that a $(1-\beta)$-fraction of the bits $b_{i,1}, \ldots, b_{i,\beta^i}$ are completely determined from M: If we wrote a 0-bit first, this follows trivially (in fact then all the bits are recovered). If we write a 1-bit first, we first obtain the set of queries that read inside C_i and their corresponding answers. From this, we can also deduce the queries that do not read inside C_i. Since the epochs $j \neq i$ are known to the decoder, he can execute the update algorithm of $\mathcal{D}$ on all updates preceeding epoch i. Now, since he knows which queries do not read in C_i, he can run $\mathcal{D}$'s query procedure on those queries. Whenever $\mathcal{D}$'s query procedure requests a memory cell, he checks whether the cell is included in the sets $C_{i-1}, \ldots, C_1$ (which is in the encoding). If so, he has the contents and can continue. Otherwise, he knows they are not in sets $C_i, ..., C_1$ and therefore the contents must be the same as they were just before epoch i. But these contents are known because he just ran all updates preceeding epoch i. He can thus finish the query algorithm and now has the answer to every query $R_1, \ldots, R_{\beta^i}$.

Now, let us remember which cells are updated in epoch i (the yellow cells in Figure 1). By the end of epoch i, the first yellow cell contains the value $b_{i,1}$, the second yellow cell contains $b_{i,2}$, etc. All the cells which are not yellow in between were written in previous epochs, and thus the decoder knows them. Therefore, by knowing all the answers to the $R_1, \ldots, R_{\beta^i}$, the decoder can recover all the bits written in the yellow positions (since the answer to query R_j is the XOR of all bits up to and including R_j, and the decoder knows all of those bits except for one, the one in yellow!). There is a slight catch: epochs $i-1, i-2, \ldots 1$ do overwrite some of the yellow cells with new bits, and thus for the blocks j where this overwriting happens, the decoder will not learn $b_{i,j}$ from the answers to the queries $R_1, \ldots, R_j$. However, note that the only indices that are overwritten in future epochs are the indices $0, n/\beta^{i-1}, 2n/\beta^{i-2}, \ldots = 0, \beta n/\beta^i, 2\beta n/\beta^i, \ldots$. In other words, only a β-fraction are overwritten, so the decoder does learn the $(1-\beta)$-fraction of the bits that weren't overwritten. But these are still $(1-\beta)\beta^i$ uniformly random bits, and any encoding to recover them should have expected encoding length at least $(1-\beta)\beta^i$, but we are achieving total expected encoding length $\mathbb{E}\,|M| < \beta^i/2$, which is still a contradiction. In other words, one can view this entire scheme as a compression scheme not for *all* the bits $b_{i,1}, \ldots, b_{i,\beta^i}$, but just the $(1-\beta)$-fraction that the decoder actually learns. Alternatively, the encoder can just always write $b_{i,1}, b_{i,\beta+1}, b_{i,2\beta+1}, \ldots, b_{i,(\beta^{i-1}-1)\beta+1}$ as part of the encoding as well, which only adds $\beta^{i-1} = o(\beta^i)$ to the encoding length, so that the decoder can in fact learn all the bits at the end of the day.

3 Lower bound for dynamic connectivity

Given the lower bound of $\max\{t_u, t_q\} = \Omega(\lg n/\lg\lg n)$ of the previous section for dynamic partial sums over $\mathbb{Z}_2$, we can obtain the same lower bound for dynamic connectivity via a reduction due to [MSVT94]. In fact the reduction holds even if the underlying graph is promised to always be a collection of two vertex-disjoint trees, which implies optimality of the link-cut trees of Sleator and Tarjan [ST83]. In dynamic connectivity we wish to support the following operations, where the graph G starts as the empty graph on n vertices:

- `insert`(u, v): insert the edge (u, v) into G
- `delete`(u, v): delete the edge (u, v) from G, if it exists
- `connected`(u, v): return `True` if u, v are in the same connected component, and false otherwise

The reduction works as follows. Recall we want to solve dynamic partial sums over $\mathbb{Z}_2$, where there is some underlying array $A[0 \ldots n-1]$ and the answer to `query`(i) is $A[0] \oplus A[1] \oplus \cdots \oplus A[i]$. We maintain a graph G on vertex set $[2n+3]$ with the following edge set E:

$$E = \left(\bigcup_{i:A[i]=0} \{(2i+1, 2i+3), (2i+2, 2i+4)\} \right) \cup \left(\bigcup_{i:A[i]=1} \{(2i+1, 2i+4), (2i+2, 2i+3)\} \right).$$

Note that a change to some entry $A[i]$ corresponds to deleting and inserting $O(1)$ new edges in G (to be precise: two edge removals and two edge insertions into G). One can show that $A[0] \oplus A[1] \oplus \cdots \oplus A[i]$ equals 0 iff `connected`$(1, 2n+3)$ is `True` after inserting the edge $(2i+, 3, 2n+3)$ into G. Thus `query` in partial sums can be implemented by one edge insertion, one `connected`

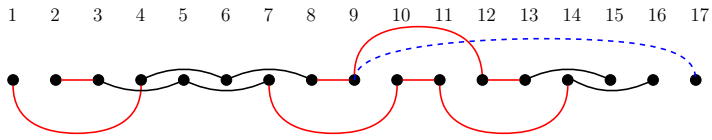

Figure 2: Example reduction from dynamic partial sums over $\mathbb{Z}_2$ to dynamic connectivity. Numbers in the top are vertex ID's. Red edges correspond to 1's in the array A, and black edges are 0's. The underlying array in this example is $A[0 \ldots 6] = [1, 0, 0, 1, 1, 1, 0]$. Thus $n = 7$, so the total number of vertices in G is $2n + 3 = 17$. The blue dashed line is inserted before querying `connected`$(1, 17)$ to test whether $A[0] \oplus A[1] \oplus A[2] \oplus A[3] = 0$.

query, then one edge deletion in G. Thus the maximum operation time in dynamic connectivity must be at least that for partial sums over $\mathbb{Z}_2$, which is $\Omega(\lg n / \lg \lg n)$ as shown in the last section.

The intuition behind this reduction is the following. If all array entries A had been zero, then E would be $\{(1, 3), (3, 5), (5, 7) \ldots\} \cup \{(2, 4), (4, 6), (6, 8), \ldots\}$. That is, there would be an "even path" (only touching vertices with even ID's) and an "odd path" (only touching vertices with odd ID's), such that after every index i in A, the even path is always one step ahead of the odd path. These are the black edges in Figure 2. Whenever there is a 1 at some entry in A, it introduces the red edges into G which swap the even and odd paths. That is, it makes the even path the odd path, and the odd path the even path. Note that 1 and $2n + 3$ are both odd numbers, so had all entries of A been 0, they would both be in the odd path together. However, due to the paths swapping at 1's, we essentially want to check that the number of swaps from the beginning up to the edges added by $A[i]$ is even.

4 What's happened since?

- $\max\{t_q, t_u\} = \Omega(\lg n)$ for both dynamic connectivity and partial sums (over $\mathbb{Z}_{2^w}$) [PD04].
- $\max\{t_q, t_u\} = \Omega((\lg n / \lg \lg n)^2)$ for dynamic weighted orthogonal range counting in $2D$, but only when the weights are $\lg^{2+\epsilon} n$-bit integers [P07].
- $t_q = \Omega((\lg n / \lg(w t_u))^2)$ for dynamic range counting in $2D$ with $\lg n$-bit weights [L12].
- $\max\{t_u, t_q\} = \tilde{\Omega}(\lg^{1.5} n)$ for parity dynamic range counting in $2D$ [LWY17].
- Many lower bounds known for other problems, e.g. union-find, predecessor, dynamic min spanning forest, planarity testing, dynamic marked ancestor, several computational geometry problems, approximate nearest neighbor in ℓ_p for various p, etc. See [P11] and also the older survey of Miltersen [M99].

References

[FS89] Michael L. Fredman, Michael E. Saks. The Cell Probe Complexity of Dynamic Data Structures. *STOC*, pages 345–354, 1989.

[L12] Kasper Green Larsen. The cell probe complexity of dynamic range counting. *STOC*, pages 85–94, 2012.

[LWY17] Kasper Green Larsen, Omri Weinstein, Huacheng Yu. Crossing the Logarithmic Barrier for Dynamic Boolean Data Structure Lower Bounds. *CoRR* abs/1703.03575, 2017.

[M99] Peter Bro Miltersen. Cell probe complexity — a survey. *In 19th Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 1999. Advances in Data Structures Workshop.

[MSVT94] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, Roberto Tamassia. Complexity Models for Incremental Computation. *Theor. Comput. Sci.* 130(1), pages 203–236, 1994.

[P07] Mihai Pătraşcu. Lower bounds for 2-dimensional range counting. *STOC*, pages 40–46, 2007.

[P11] Mihai Pătraşcu. Unifying the Landscape of Cell-Probe Lower Bounds. *SIAM J. Comput.* 40(3), pages 827–847, 2011.

[PD04] Mihai Pătraşcu, Erik D. Demaine. Tight bounds for the partial-sums problem. *SODA*, pages 20–29, 2004.

[ST83] Daniel Dominic Sleator, Robert Endre Tarjan. A Data Structure for Dynamic Trees. *J. Comput. Syst. Sci.* 26(3), pages 362–391, 1983.

[Y78] Andrew Chi-Chih Yao. Should Tables Be Sorted? *FOCS*, pages 22–27, 1978.