# Lecture 7 — February 24, 2005

*Lecturer: Mihai Pătraşcu* *Scribe: Tim Abbott*

## 1 Overview

In the last lecture we saw how to solve the dynamic connectivity problem in $O(\log^2 n)$ time per operation. To recall, here we want a data structure to be able to dynamically handle queries as to whether two elements are in the same connected component in a graph. The data structure should support insert and delete operations on both edges and isolated vertices.

In this lecture we show a lower bound on dynamic connectivity, that it is impossible to do better than $\Omega(\log n)$ time per operation [1]. This is the first lower bound to be covered in this class.

## 2 A Lower Bound on Dynamic Connectivity

### 2.1 The Cell Probe Model

In order to prove a lower bound, we need a model of computation. It is best for the model of computation to be as general as possible, since then the lower bound is as meaningful as possible. Here we work in the cell probe model. In this model, we have a memory consisting of cells, each with $w$ bits. It is assumed that $w = \Theta(\log n)$, so that we can store in a cell a pointer to an arbitrary cell, since we typically only have memory polynomial in $n$. We have complete access to read or write any of the cells. However, the runtime of an algorithm in this model is the number of cell probes (reads or writes) that it carries out during execution of the algorithm.

All of our work today will be in the cell probe model. Notice that this model is more general than the pointer machine model, since one can implement a pointer machine in this model. Our goal will be to prove the following theorem of Pătraşcu and Demaine:

**Theorem 1.** *Dynamic connectivity requires $\Omega(\log n)$ time per operation in the worst case, in the cell probe model.*

The following more general result is also true, though we will not prove it here.

**Theorem 2.** *Dynamic connectivity requires $\Omega(\log n)$ time per operation in the cell probe model, even if amortized and randomization are allowed.*

### 2.2 The Construction

We consider the following data structure problem. We have a graph on an $\sqrt{n} \times \sqrt{n}$ grid, where the only edges connect vertices in adjacent columns. Further, we require that the edges between

column $i$ and column $i+1$ form a permutation $\pi_i$. Thus, each node not on the first or last column has exactly two neighbors in the graph.

We require that the data structure support two distinct operations:

- Update$(i, \pi)$, which replaces the $i$th permutation $\pi_i$ by $\pi$.
- Verify-Sum$(i, \pi)$, which checks whether the composition of the first $i$ permutation is equal to $\pi$, i.e. whether

$$\pi_i \circ \pi_{i-1} \circ \ldots \circ \pi_2 \circ \pi_1 = \pi$$

Now, we can give an efficient implementation for this data structure using dynamic connectivity on the graph of this data structure. The algorithm handles Update operations by doing $\sqrt{n}$ inserts and $\sqrt{n}$ deletes for the $\sqrt{n}$ edges that are changed. It handles Verify-Sum operations by checking for each $j$ whether $(1, j)$ is connected to $(i, \pi(j))$ is connected in the graph. They are connected if and only if the composition of the permutations, evaluated at $j$, gives $\pi(j)$.

Each of the operations on our grid data structure uses $O(\sqrt{n})$ dynamic connectivity operations. The theorem then follows from the following proposition, which we will spend the rest of the lecture proving.

**Proposition 3.** *Performing $\sqrt{n}$ Update and $\sqrt{n}$ Verify-Sum operations requires $\Omega(n \log n)$ operations in the worst case.*

**Proof of Theorem** Assume the last proposition holds. Now, as we showed above, we can perform the $O(\sqrt{n})$ operations using a total of $O(n)$ dynamic connectivity operations. It follows that dynamic connectivity must require $\Omega(\log n)$ time in the worst case.

### 2.2.1 The Input Sequence

We need only find a hard input sequence. We construct the input sequence by starting with a permutation $\sigma$ of the columns indices. Then, we iterate through the permutation $\sigma$, and in each step $i$ we first Update the column $\sigma(i)$ and then Verify-Sum on that same column $\sigma(i)$. The permutation $\sigma$ is constructed by taking the list of the binary numbers between 0 and $\sqrt{n} - 1$, and reversing the order of the bits. In each Update we change the permutation to a random permutation, while on the Verify-Sum queries we always give the correct permutation $\pi$.

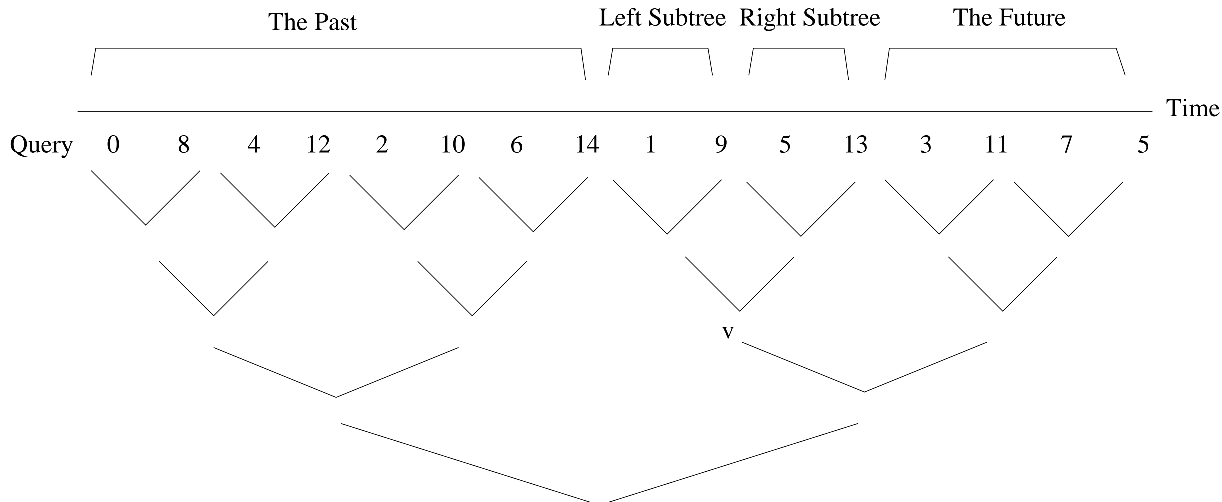
**Example** For $\sqrt{n} = 8$, the sequence is

$$(000_2, 100_2, 010_2, 011_2, 001_2, 101_2, 011_2, 111_2) = (0, 4, 2, 6, 1, 5, 3, 7)$$

The input sequence we would give would then be

$$U(0), Q(0), U(4), Q(4), U(2), Q(2), U(6), Q(6), U(1), Q(1), U(5), Q(5), U(3), Q(3), U(7), Q(7)$$

where $U(i)$ is a change of the $i$th column to a random permutation, and $Q(i)$ is a Verify-Sum query on the $i$th column, with the correct answer. Notice that the sequences of queries defined above interleaves betwen adjacent blocks perfectly.




### 2.2.2 Interleaving

A key feature of the input sequence constructed is that the columns queried perfectly interleave. This is best illustrated by considering a complete binary tree drawn over the time axis such that each node has in its right subtree the leaves occuring after it, and in the left subtree the leaves occuring before it. The subtree of a node forms a complete interval of time, so the nodes to the left of it form its past and the nodes to the right of it form its future.

The intuition here is that the difficulty in resolving these queries relates to the fact that at each node in this binary tree, there is a lot of work forced on the right subtree by the interleaving with the left subtree. Thus, each node has a considerable amount of work done with it, and it is possible that each level could do the same amount of work, leading to the log factor in the bound we're trying to prove.

**Theorem 4.** *For any node $v$, let $\ell$ be the number of leaves in its subtree. Then the right subtree of $v$ must execute $\Omega(l\sqrt{n})$ cell probes reading from the left subtree, in expectation.*

To see why this should imply our overall lower bound, note that a cell probe is characterized by the time $r$ when it is read, and by the time $w$ when the data being read was written. In our tree, the two uniquely determine a least common ancestor. The least common ancestor is then the node which that cell probe will be counted against, and we can thus be sure we are not double-counting any cell probes. Now, this alone would not be sufficient to add these bounds together. If they were worst-case bounds, for example, then perhaps the data structure achieves the worst case in different instances for different parts of the tree. However, since this bound is in expectation, by linearity of expectation, the expected total number of operations done is at least the sum of these values over the branches. Now, given this, we can add the bounds together to determine that the total number of operations done at each level of the tree is $\Omega(n)$, so that we spend $\Omega(n \log n)$ cell probes in total, as desired.

It suffices then to prove the theorem, so assume it is false. We use an encoding argument. The idea is that in order for the right subtree to correctly calculate queries, it must read enough information to compute the random permutations given in the inputs to the left subtree updated. The left

subtree consists of $\ell$ updates, each of which has a random permutation in $S_{\sqrt{n}}$. The following lemma is a very simple result from information theory, which reader can easily check:

**Lemma 5.** *Any encoding of $\ell$ permutations in $S_{\sqrt{n}}$ must use $\Omega(\ell\sqrt{n}\log n)$ bits in expectation.*

## 2.3 A Simpler Model

We will now attempt to use the assumption that we have a data structure violating Theorem 4 to contradict Lemma 5. First, let us step into an easier problem. Suppose that we replaced the Verify-Sum query with a Sum query, that in fact gave the value of the $i$th permutation rather than simply checking correctness of a guess. We will show the result in this modified model, and then shift it back.

The setup here is that we know the past (before the left subtree), and we are trying to encode the permutations passed to updates from the left subtree. Because of the perfect interleaving of the bit-reversal permutation, this is equivalent to encoding the sums returned by all queries from the right subtree. Define the following sets:

$$R = \{\text{Cells read in the right subtree}\}, W = \{\text{Cells written in the left subtree}\}$$

We encode the set $R \cap W$, and with each cell both its address and its contents. Given this, we can decode to obtain the permutations in the updates during the left subtree as follows.

For every cell read in the right subtree, it is one of either:

1. Cells written in the right subtree. These cells we know as they were written in this phase (by induction on time, we know everything that the data structure does in the right subtree).
2. Cells written in the past. These cells are known since we are supposing that our data structure knows the past.
3. Cells written in the left subtree. The address and contents of these cells are available as they are in $R \cap W$; one can check whether they were written in the left subtree by checking the address. Thus the data in these cells are known as well.

By induction on time, we can simulate the data structure during the right subtree; in particular, the simmulation recovers the answers of the queries. Because of the interleaving property of our input, we can use this data to decode the random permutations passed to updates in the left subtree, since for each update we can compute the sum before and after the change. Thus, we must have in our encoding at least $\Omega(\ell\sqrt{n}\log n)$ bits. The size of our encoding is $|R \cap W| O(\log n)$, since a word and address have $O(\log n)$ bits. Thus, we must have that

$$|R \cap W| = \Omega(\ell\sqrt{n})$$

which completes the proof of the Theorem 4. Note that this result is in expectation, since the number of bits of information needed to encode our random permutations is only a lower bound on the expected number of bits required.

Now, we have shown the theorem is true if our operations are Sum and Update; but in fact we do not have Sum, we only have Verify-Sum. We thus have somewhat less information than we had in the previous setting – we know that all the Verify-Sum queries returned true, by our construction, but we do not know what the values of $\pi$ they queried were. The past is still fixed, and we still do not know what the updates that were performed during the left subtree.

One is led to consider the following algorithm: Once again, we encode the address and contents of each element of $R \cap W$. To decode, we simulate the algorithm running on all possible Verify-Sum queries for each of the Verify-Sum queries on the right subtree. We know that one of these queries returns true, that being the one corresponding to the correct value of $\pi$, and all the other should return false. Thus, the data structure already has encoded in it the correct value of $\pi$.

However, that algorithm is **incorrect**. The issue here is somewhat subtle, and causes this method to return false positives. Let $R'$ be the set of cells read by these incorrect queries. The difficulty here is that the read operations on $R'$ might intersect with $W$ in cells not in $R \cap W$, so that the state data needed to evaluate these read operations will get the value from the past, when in fact it should get data written during the left subtree. Since the algorithm is reading incorrect bits on its cell probes when the query has an answer that should be no, it might return yes instead.

## 2.4 Separators

One possible fix for this algorithm would be to encode all of $W$. However, assuming that our theorem holds, $W$ could be as large as $\ell\sqrt{n}\log n$ cells, or $\ell\sqrt{n}\log^2 n$ bits. Then our encoding would not fit in the desired $\ell\sqrt{n}\log n$ space. However, we do not actually need to encode the set $W$. Instead, we must be able to detect whether a cell probe is in $W \setminus R$, since we know that any cell probe to a cell in that space must come from a Verify-Sum query with an argument that should give the answer false, and our algorithm can stop.

**Definition 6.** *A* separator family *for size $m$ sets is a family $S \subset 2^U$ with the property that for any $A, B \subset U, |A|, |B| \leq m$, there exists $C \in S$ such that $A \subset C, B \subset U \setminus C$.*

**Theorem 7.** *There exist separator families $S$ such that $|S| \leq 2^{O(m+\log\log U)}$.*

Thus given sets $A, B$ to separate, we require $\log |S| \leq O(m + \log\log U)$ bits to indicate a certificate of separation (a set $C$ with $A \subset C, B \subset U \setminus C$). This theorem follows from the known bounds for Bloomier filters, or can be proved via the probabilistic method.

Aside from the information in our encoding of $R \cap W$, we also need to encode a separator for $R \setminus W$ and $W \setminus R$. Now, to decode the sums in the left subtree, we simulate all possible values for the permutation in each Verify-Sum query in our input sequence. Now, the Verify-Sum query with the correct sum only makes cell probes to good cells (those in $R \cap W$), thus our simulation will give the correct response of true to that query. But suppose we are simuating an incorrect Verify-sum query. The cells read in this simulation are either

1. All from $R$, in which case it can be simmulated and returns false correctly (we can detect cells from $R \cap W$ since we have the addresses for these cells, and all other cells have known values)

2. At least one comes from the $W \setminus R$ side of the separator, in which case the separator detects this. Since no cell on that side of the separator is read if we have a correct query, we can abort the simulation with return value false.

3. A cell probe from the $R \setminus W$ side is fine, since that was written in the past, which we are assumed to have complete knowledge of. Thus, we can simmulated the query, and it will return false correctly.

Thus, we have a method for decoding the values of all the permutations in the left subtree updates, given both $R \cap W$ and a separator of size $O(|R| + |W| + \log\log n)$. Then,

$$|R \cap W| O(\log n) + O(|R| + |W| + \log\log n) = \Omega(\ell\sqrt{n}\log n).$$

We have two possibilities here. If $|R| + |W| = \Omega(\ell\sqrt{n}\log n)$, then we have not proven anything, because our estimate is then trivially true. However, in this case, there were a total of $O(\ell)$ operations handled by each of the left and right subtrees, each of which does $\sqrt{n}$ dynamic connectivity operations. Then at least one of those dynamic connectivity operations used $\Omega(\log n)$ time, and the main theorem holds.

Otherwise, we have that $|R| + |W| = o(\ell\sqrt{n}\log n)$, and we've thus shown that $|R \cap W| = \Omega(\ell\sqrt{n})$. This implies that there were at least that many cell probes were forced on the right subtree by the left subtree, which is exactly what we were trying to show.

# References

[1] M. Pătraşcu, E. Demaine, *Lower bounds for dynamic connectivity*, Proc. 36th Symp. Theory of Computing (STOC 2004), ACM, 2004. Full version available as *arXiv:cs.DS/0502041*.