6.851: Advanced Data Structures Spring 2012

Lecture 21 — May 8, 2012

Prof. Erik Demaine

1 Overview

In the last lecture we introduced Euler tour trees [3], dynamic data structures that can perform link-cut tree operations in $O(\lg n)$ time. We then showed how to implement an efficient dynamic connectivity algorithm using a spanning forest of Euler tour trees, as demonstrated in [4]. This yielded an amortized time bound of $O(\lg^2 n)$ for update operations (such as edge insertion and deletion), and $O(\lg n/\lg\lg n)$ for querying the connectivity of two vertices. In this lecture, we switch to examining the lower bound of dynamic connectivity algorithms. Until recently, the best lower bound for dynamic connectivity operations was $\Omega(\lg n/\lg\lg n)$, as described by Fredman and Henzinger in [1] and independently by Miltersen in [2]. However, we will show below that it is possible to prove $\Omega(\lg n)$, using the method given by Pătraşcu and Demaine in [6].

2 Cell Probe Complexity Model

The $\Omega(\lg n)$ bound relies on a model of computation called the *cell probe complexity model*, originally described in the context of proving dynamic lower bounds by Fredman and Saks in [5]. The cell probe model views a data structure as a sequence of *cells*, or words, each containing a w-bit field. The model calculates the complexity of an algorithm by counting the number of reads and writes to the cells; any additional computation is free. This makes the model comparable to a RAM model with constant time access. Because computation is free, the model is not useful for determining upper bounds, only lower bounds.

We empirically assume that the size of each cell, w, is at least $\lg n$. This is because we would like the cells to store pointers to each of our n vertices, and information theory tells us that we need $\lg n$ bits to address n items. For the the following proof, we will further assume that $w = \Theta(\lg n)$. In this sense, the cell probe model is a *transdichotomous model*, as it provides a bridge between the problem size, n, and the cell or machine size, w.

3 Dynamic Connectivity Lower Bound for Paths

The lower bound we are trying to determine is the best achievable worst-cast time for a sequence of updates and queries to a path. We will prove the following:

Theorem 1. *Under the cell probe model, the lower bound worst-case cost is $\Omega(\lg n)$ per operation.*

It is possible to show that the lower bound is also $\Omega(\lg n)$ in the amortized case, but we will only examine the worst case cost.

3.1 Path Grid

We start by arranging the n vertices in a $\sqrt{n}$ by $\sqrt{n}$ grid, as shown in Figure 1. The grid has perfect matching between consecutive columns C_i and C_{i+1}. This results in a graph with $\sqrt{n}$ disjoint paths across the columns. The paths can be coded by the permutations $\pi_1, \pi_2, \ldots \pi_{\sqrt{n}-1}$, where π_i is the permutation of edges that joins column C_i and C_{i+1}.

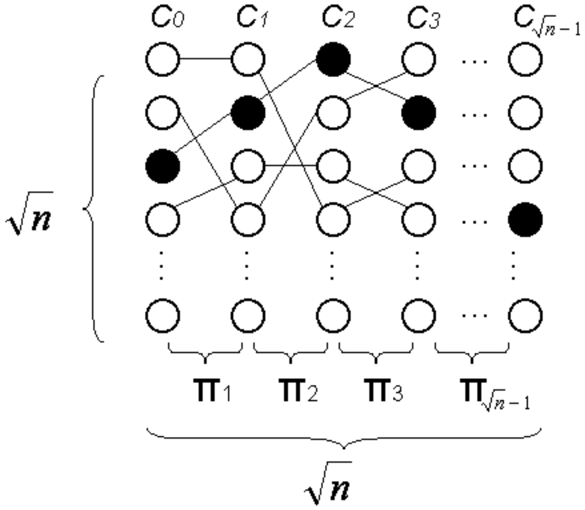

Figure 1: A $\sqrt{n}$ by $\sqrt{n}$ grid of vertices, with one of the disjoint paths darkened.

3.2 Block Operations

There are two operations that we define on the grid:

- *UPDATE(i, π)* – Replaces the i^{th} permutation π_i in the grid with permutation π. This is equivalent to $O(\sqrt{n})$ edge deletions and insertions.
- *VERIFY_SUM(i, π)* – Check if $\sum_{j=1}^{i}(\pi_j) = \pi$. In other words, check if the first i permutations $\pi_1, \pi_2, \ldots \pi i$ is equivelent to a single permutation π. This is equivalent to $O(\sqrt{n})$ connectivity queries.

We can now described the dynamic connectivity problem in terms of these operations and cell probe operations. Thus, we make the following claim:

Claim 2. *Performing $\sqrt{n}$ UPDATE(i, π) operations and $\sqrt{n}$ VERIFY_SUM(i, π) operations requires $\Omega(\sqrt{n}\sqrt{n}\lg n) = \Omega(n \lg n)$ cell probes.*

3.3 Construction of Bad Access Sequences

To prove the above claim, we need to use a family of random, “bad” sequences so as to ensure that we achieve the true lower bound. Otherwise, it would be possible for the data structure to tune to the input sequences, allowing it to take advantage of patterns in the sequences and run faster than it would in the worst case. We want to come up with the most difficult sequence of operations and sequences possible to ensure a worst case lower bound.

First, we will alternate the update and query operations, $\mathsf{UPDATE}(i, \pi)$ and $\mathsf{VERIFY\_SUM}(i, \pi)$. Then, we will carefully select the arguments to these operations as follows:

- π *for UPDATE* – The permutation π for each UPDATE operation will be uniformly random, which will randomly change the result of each SUM operation (where SUM computes the sum of i permutations, which must also be performed by VERIFY_SUM).
- π *for VERIFY_SUM* – Using uniformly random permutations is not appropriate, because it is easy for the operation to check that a permutation *doesn't* match the sum; it only needs to find one path in the sum that doesn't satisfy π. The worst case permutation is actually the "correct" permutation, $\pi = \sum_{j=1}^{i}(\pi_j)$, because this forces the operation to check *every* path before returning TRUE.
- i – For both operations, the i's follow the *bit reversal sequence.* To generate this sequence, take all n-bit numbers n_i, write them in binary, then reverse (or "mirror") the bits in each number n_i to create a new number m_i. This results in a new sequence of numbers with some special properties. For example, the sequence maximizes the WILBER_1 function, as it chooses a path with the maximal number of non-preferred children at each step.

Example Using the argument selection rules above, we will alternate between update and query operations using worst case arguments, resulting in a "bad" access sequence. For example, with 3-bit cells, the bit reversal 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 access sequence we would give would then be

$$QUERY(0, \pi_{correct}), UPDATE(0, \pi_{random}), QUERY(4, \pi_{correct}), UPDATE(4, \pi_{random}), \ldots$$

Notice that the sequences of queries defined above interleaves betwen adjacent blocks perfectly.

3.4 Tree of Time

To keep track of the order of the order of the sequence of acesses, we can create a binary tree in which each leaf represents one $QUERY/UPDATE$ pair of operations, and the leaves are in chronological order. We will refer to this as the tree of time. Note that for all h and m, the mth node at height m has all leaves with i such that i's last h digits are the bit reversal of m as descendents. As a result, the leaves in the left and right subtrees of any non-leaf node in this tree are interleaved. This implies that one can determine exactly what updates were performed in the left subtree of any node by making the right queries on values of i represented by leaves in the right subtree. The need to store and then retrieve this information is what makes this access sequence hard.

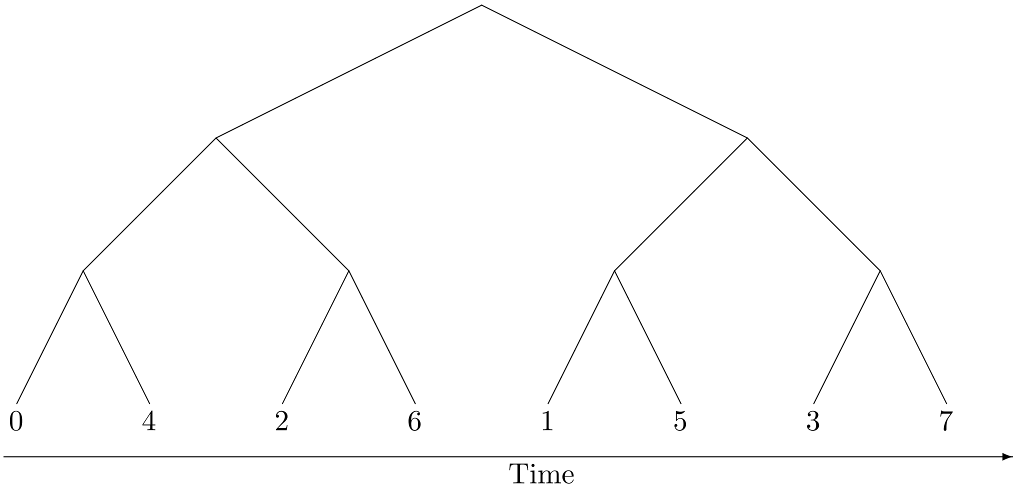

Figure 2: The tree of time for $w = 3$

Claim 3. *For any non-leaf node of the tree of time, v, let ℓ be the number of leaves in each of its subtrees. Then the series of operations represented in the right subtree of v must execute $\Omega(\ell\sqrt{n})$ probes of cells that were last modified during operations represented in the left subtree of v.*

For any given cell probe, there is at most one leaf that represents the step when the cell was last modified, and one leaf representing the step in which this probe occurs. So, the only node that includes this probe in its $\Omega(\ell\sqrt{n})$ probes is the least common ancestor of these two leaves.

There are $\sqrt{n}$ leaves total, so the sum of the values of ℓ corresponding to all of the nodes on a given level is $\sqrt{n}$. There are $\Omega(\log n)$ levels; therefore, this claim implies that the total number of cell probes used by this sequence of operations is $\Omega(\log n) \cdot \Omega(\sqrt{n} \cdot \sqrt{n}) = \Omega(n \log n)$. Thus, this claim implies claim 2.

3.5 Plan for proving claim 3

In order to prove Claim 3, we will show that for any node, v, in order to answer the queries represented in v's right subtree the program will need to use $\Omega(\ell\sqrt{n}\log n)$ bits of information that were set during operations represented in the left subtree of v.

Assuming that we knew the exact configuration of the paths prior to the operations represented in v's subtrees, it is possible to determine exactly what permutations were used by each $UPDATE$ operation represented in v's left subtree by executing the right series of $VERIFY\_SUM$ operations using i with leaves in v's right subtree. There are $\sqrt{n} = 2^{\theta(\sqrt{n}\log n)}$ possible permutations of $\sqrt{n}$ elements, so it takes $\Omega(\ell\sqrt{n}\log n)$ bits of information to track the set of modifications performed in the left subtree of v.

3.6 Simplified proof using SUM

Rather than doing the relevant proof immediately, we will start with a warmup. So, imagine that instead of using $VERIFY\_SU <$ as our queries, we are using

$$SUM(i) = \sum_{j=0}^{i} \pi_j$$

In other words, $SUM(i)$ returns the composition of the first i permutations.

Let R be the set of all cells accessed during the right subtree of v and W be the set of all cells written during the left subtree of v. Assuming that the size of each cell is $w = \theta(\log n)$, $R \cap W$ can be encoded in $O(|R \cap W|w)$ bits, and I claim that the full series of operations performed in the right subtree of v can be simulated using only a recording of the states of the cells immediately before the left subtree of v began, and an encoding of $R \cap S$.

Each time our simulation needs to know the information in a cell, there are three possible cases, based on where in the tree of time that cell was last modified.

1. v's right subtree: Its current value was set by another operation in the simulation, so we can easily determine it.

2. v's left subtree: This cell must be in $R \cap W$, so we can get its value from the encoding of $R \cap W$.

3. past subtrees: We can simply look its value up in the recording of the cells' states before v's subtree started executing.

In every case, we can give the simulation the data it needs to continue correctly, so it can give the correct outputs in response to all queries. Having the values of $SUM(i)$ for each leaf in v's right subtree provides enough information to determine exactly what permutations were performed in v's left subtree, so the output of the simulation must be different for each possible set of permutations that could be performed there. The only part of the simulation's input that depends on which permutations were performed in v's left subtree is $R \cap W$, so there must be at least as many possible values of $R \cap W$ as there are possible sets of permutations. Thus, $|R \cap W| = \Omega(\ell\sqrt{n}\log n)/\log n = \Omega(\ell\sqrt{n})$.

3.7 Proof using VERIFY_SUM

We have shown that Claim 3's bound of $\Omega(l\sqrt{n})$ cell probes is necessary to transfer the information from the UPDATE operations on the left subtree to the SUM operations on the right. Now we will try to prove a similar result using VERIFY_SUM.

Query: VERIFY_SUM(i, π): $\sum_{j=1}^{i}(\pi_j) \overset{?}{=} \pi$. Returns TRUE if the composition of the first i permutations is equivalent to permutation π, and returns FALSE otherwise.

Setup: Again, we assume we know the entire past. However, this time, we will not assume that we know updates in the left subtree or queries in the right subtree. We do know though, that for any i, only one π will result in TRUE. Furthermore, recall that we are using the worst case π inputs for $\mathsf{VERIFY\_SUM}$ operations. Because the worst case π is the "correct" permutation, we know that $\mathsf{VERIFY\_SUM}$ will always return true!

Encoding: As before, define R, W, and P. Instead of just having to encode the sums, however, we also have to encode the input π's. This is avoided by noting that the input π must always be the permutation that causes $\mathsf{VERIFY\_SUM}$ to return true, as described in the setup. Instead of encoding π, we can just recover during decoding by trying all possible π_i permutations and selecting the one which matches the encoded P permutation.

Decoding: We start by simulating all possible input permutations to $\mathsf{VERIFY\_SUM}$ so as to recover π, as described above. As before, the decoding algorithm relies on the knowledge of where the cell was last written. Unfortunately, this is no longer easy to discern. Because we simulated all possible input permutations, we queried cells in set R' as well as in R. Let R' be the set of cells read by these incorrect queries. If we read cell $r \in R' \setminus R$, then the permutation π must be incorrect. However, there is a chance that r will intersect with W in cells not in P, so that the state data needed to evaluate these read operations will get the value from the past, instead of getting 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, resulting in a false positive.

Result: We could get around this problem by encoding all of R or W, allowing us to check whether $r \in W \setminus R$ or $r \in \text{past} \setminus P$. However, this requires a large number of bits; W could be as large as $\ell\sqrt{n}\log n$ cells, or $\ell\sqrt{n}\log^2 n$ bits, prevent our encoding from fitting in the desired $\ell\sqrt{n}\log n$ space. Instead, we will try to find a way to cheaply encode whether we are in $R \setminus W$ or $W \setminus R$.

3.8 Separators

To solve the decoding problem described for $\mathsf{VERIFY\_SUM}$ queries, we encode a *separator*.

Definition 4. *A* separator family *for size m sets is a family $S \subset 2^U$ with the property that for any $A, B \subset U$, with $(|A|, |B| \leq m)$ and $(A \cap B = \varnothing)$, there exists a $C \in S$, such that $(A \subset C)$ and $(B \subset U \setminus C)$.*

Theorem 5. *There exist separator families S such that $|S| \leq 2^{O(m+\log\log U)}$.*

Theorem 5 results from the fact that we can have a perfect hash family H with $|H| \leq 2^{O(m+\lg\lg|U|)}$, which maps $A \cup B$ to an $O(m)$-size table with no collisions. Each of these table entries stores two bits indicating if that element is in A or B. Storing the perfect hash function takes $\lg|H|$ bits, and the table entries take $2O(m)$ bits, so overall $\lg|H| + 2O(m) = O(m + \lg\lg|U|) + O(m) = O(m + \lg\lg|U|) = \lg|S|$.

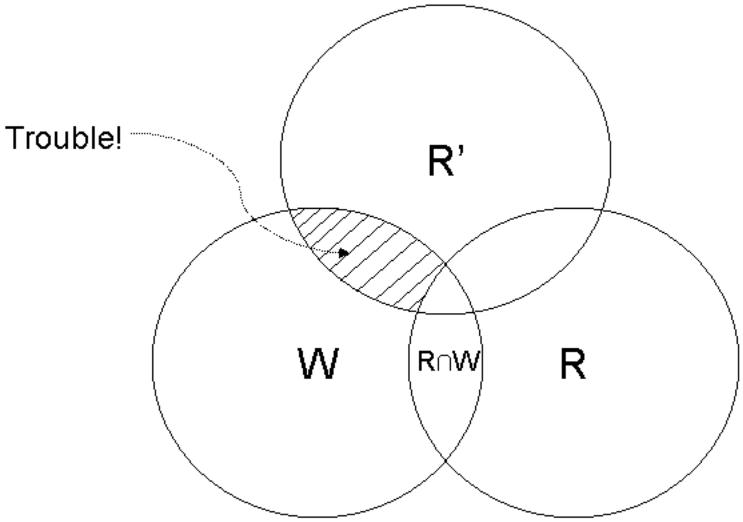

Figure 3: Venn diagram showing the intersections of R, W, and R'.

Therefore, we can encode a separator S in $O(|R| + |W| + \lg\lg n)$ bits. The separator will have the property that $R \setminus W \subseteq S$ and $W \setminus R \subseteq \overline{S}$. By encoding the separator S along with P, we can successfully decode each write in the following manner.

Decoding (with Separator): To decode, we first simulate all input permutations to VERIFY_SUM so as to recover π. Then, we determine when the cell being read was last written:

- *Right subtree* – As before, we have knowledge of the permutations performed on the right subtree, so decoding here is trivial.
- P – We again use the information encoded in the simulated P set to determine what permutations were performed.
- S – If the cell is in S but not $R \setminus W$, then it must have last been written in the past, and we assume we know the effects of past permutations
- $\overline{S}$ – If the cell is not in S, then it must not be in R, meaning that this can't be the correct π. So we abort this decoding and look elsewhere for the appropriate π.

3.9 Conclusion

We have shown that we can decode the permutations caused by the left subtree UPDATE operations by using an encoded representation of P and a separator S of size $O(|R| + |W| + \lg\lg n)$, giving us a lower bound of

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

There are two possibilities to consider.

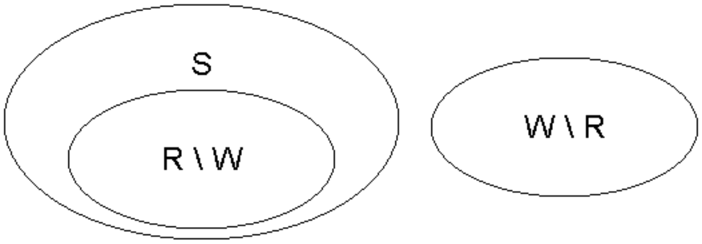

Figure 4: The separator S distinguishing between $R \setminus W$ and $W \setminus R$.

- $|R| + |W| = \Omega(\ell\sqrt{n} \lg n)$ – If $|R| + |W|$ is large, 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 $\Omega(\lg n)$ time, and the main theorem holds.
- $|R| + |W| = o(\ell\sqrt{n} \lg n)$ – 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. L. Fredman and M. R. Henzinger. *Lower bounds for fully dynamic connectivity problems in graphs.* Algorithmica, 22(3):351362, 1998.

[2] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia. *Complexity models for incremental computation.* Theoretical Computer Science, 130(1):203236, 1994.

[3] M. R. Henzinger, V. King, *Randomized dynamic graph algorithms with polylogarithmic time per operation.* STOC 1995: 519-527

[4] J. Holm, K. Lichtenberg, M. Thorup, *Poly-logarithmic deterministic fullydynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity.* J. ACM 2001: 48(4): 723-760

[5] M. Fredman, M. Saks, *The cell probe complexity of dynamic data structures.* STOC 1989: 345-354

[6] M. Pătraşcu, E. D. Demaine, *Lower bounds for dynamic connectivity.* STOC 2004: 546-553