6.851: Advanced Data Structures Spring 2010

Lecture 18 — April 14, 2010

Prof. Erik Demaine Scribe: Matthew Hofmann, Edited by: Morteza Zadimoghaddam

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]. In fact, we prove this lower bound even if the connected components of our graph are always some paths. We use a weak assumption here that the word size is $O(\log n)$. This is equivalent to assuming that the space of our algorithm (our problem) is polynomial in the size of the input which is a reasonable assumption. Note that handling super-polynomial space probably needs a super-polynomial time algorithm.

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 $\sqrt{n} \times \sqrt{n}$ grid, as shown in Figure 1. For each $1 \leq i < \sqrt{n}$, 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}. So every node in column i is in the same connected component with exactly one node in another column j for $j \neq i$. For example node x in column 1 is in the same connected component with node $\pi_1(x)$ in the second column, node $pi_2(pi_1(x))$ in the third column, and so on. So using the composition of these permutations, we can see which pair of nodes in different columns are in the same connected component.

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 can be done by $O(\sqrt{n})$ edge deletions and insertions. We just need to delete all edges between columns i and $i+1$, and insert a new set of edges between them that represents the new permutation.
- *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 given permutation π. This query can be answered with $O(\sqrt{n})$ connectivity queries.

We can now describe 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* $\Omega(\sqrt{n} \times \sqrt{n} \lg n) = \Omega(n \lg n)$ *accesses to 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, UPDATE(i, π) and VERIFY_SUM(i, π). 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). Choosing the input permutation uniformly random makes the sum functions (the composition of the consecutive permutations) also uniformly random. Of course the consecutive sequence should contain the updated permutation in order to have this randomness property.
- 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 original bit sequence is

$$(000_2, 001_2, 010_2, 011_2, 100_2, 101_2, 110_2, 111_2) \rightarrow$$

Therefore, 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. Note that $\pi_{correct}$ in each query is the correct permutation. In other words, these permutations are chosen in a way that the results of all queries are true. So if we are asking a query for column i the input permutation we give to the algorithm is $\sum_{j=1}^{i} \pi_i$ where π_j is the current permutation of column j, and summing means composition of these permutations as some functions.

3.4 Tree of Time

To keep track of our position in the bit reversal sequence, we can store each number in the sequence in a leaf node of a perfect tree. The resulting tree has the interesting property being ordered by time from left to right. We refer to this tree as the *tree of time*. The tree of time for $w = 3$ is shown in Figure 2.

Because we used the bit reversal sequence to order the keys in the tree of time, it has the additional property that the left and right subtrees of any non-leaf node interleave perfectly. This interleaving

is the reason that the bit reversal sequence is useful in our lower bound proof. Each node in the right subtree needs to be aware of the UPDATE operations performed on the nodes in the left subtree. In other words, the right subtree must read the writes made in the left subtree. The resulting transfer of information between the interleaved left and right nodes is what makes the access sequence hard.

Claim 3. *For every node v in the tree, let ℓ be the number of leaves in its subtree, corresponding to ℓ units of time. Then the right subtree of v is expected to execute $\Omega(l\sqrt{n})$ cell probes that read from cells last written during the left subtree.*

Because ℓ maps onto the columns $C_0, C_1, \ldots, C_{\sqrt{n}-1}$ of our grid of vertices, we know that $\ell = \sqrt{n}$ when v is the root of the tree of time. By summing over all levels of the tree, we end up summing over $\Omega(\lg n)$ levels of v; however, to avoid double-counting read-write pairs across subtrees, we must make sure to count the read of r (in the right subtree) of the write w (in the left subtree) only at $v = LCA(r, w)$. Finally, linearity of expectation gives us the bound from Claim 2:

$$\Omega(l\sqrt{n}) \text{ per operation} \cdot \Omega(\lg n) \text{ operations} = \Omega(\sqrt{n}\sqrt{n}\lg n) = \Omega(n \lg n)$$

3.5 Proof Idea using Information Theory

To prove Claim 2, we need to first prove Claim 3. To accomplish this, we will assume that Claim 3 is false, meaning that we assume we can beat $\Omega(l\sqrt{n})$ per cell probe and $\Omega(l\sqrt{n}\lg n)$ overall. Showing that this assumption leads to a contradiction is sufficient to prove Claims 3 and 2.

To uncover the contradition we will use an encoding argument from information theory. The idea is that there must be information transferred from the left subtree to the right tree, because every UPDATE in the left subtree must be known by the right subtree during its VERIFY_SUM operations. Under the cell probe model, information can only be transferred via cell reads. Therefore, we need to determine a lower bound on the number of bits required to encode the permutations each read operation may need to recover. Kolmogorov complexity provides us with the following theorem:

Theorem 4. *Encoding a single random permutation on n elements requires $\Omega(n \lg n)$ bits.*

If v has ℓ leaves, then its left subtree has $\frac{\ell}{2}$ leaves corresponding to $\frac{\ell}{2}$ random permutations on $\sqrt{n}$ elements. Applying Theorem 4 yields the lower bound we are looking for.

Lemma 5. *Encoding $\frac{\ell}{2}$ random permutations on $\sqrt{n}$ elements requires $\Omega(\ell\sqrt{n}\lg n)$ bits.*

3.6 A Simpler Proof using SUM

Our task is now to assume that Claim 3 is false and show that this assumption leads to a contradiction with Lemma 5. To do this, we start by assuming we know the entire history of permutations before v's subtree. Then, we encode the verified sums in v's right subtree such that we can recover the sequence of random permutations, $S(\pi_i)$, from the left subtree. If this encoding requires $\Omega(\ell\sqrt{n}\lg n)$ bits as decribed by Lemma 5, then we know this must be the lower bound for Claim 3. However, this is a difficult proof, so we will start with a simpler case. Instead of encoding the verified sums, we will instead encode the sums of the permutations.

Query: SUM(i): $\sum_{j=1}^{i}(\pi_j)$, or the composition of the first i permutations.

Setup: We know the entire past, including the permutations that occurred before v's subtree.

Encoding: Let $R = \{\text{cells read during right subtree}\}$, and $W = \{\text{cells last written during left subtree}\}$. The set P contains the cells read in the right subtree which were last written to earlier, in the left subtree. Note that P might be a bit different from $R \cap W$. Now, we will try encoding P. If the size of each cell is $w = \Theta(\lg n)$, then it will require $2w$ bits to encode addresses and contents of each cell in P. The total encoding size is therefore $|P| \cdot \lg n$ bits.

Decoding: To decode, we first simulate all possible SUM(i) queries in the right subtree. This generates P. Decoding now depends on where the cell was last written:

- *Right subtree* – If the cell was last written in the right subtree, we have access to the permutations performed, and decoding is trivial.
- *Left subtree* – We can use the information encoded in the simulated P set to determine what permutations were performed by the left subtree's UPDATE operations.
- *Past subtrees* – Because we assume we know the entire past, we know the permutations caused by past subtrees.

Result: We know by Lemma 5 that our encoding must be at least $\Omega(\ell\sqrt{n}\lg n)$ bits. This maps onto to set we are encoding, $|P| \cdot \lg n$. By cancelling out the $\lg n$ terms, we discover that $|P| = \Omega(\ell\sqrt{n})$, which agrees with Claim 3.

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): $\sum_{j=1}^{i}(\pi_j) =?\pi$. 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. We also know that, for any i, only one π will result in TRUE. Furthermore, recall that we are using the worst case π inputs for VERIFY_SUM operations. Because the worst case π is the "correct" permutation, we know that 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 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 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 VERIFY_SUM queries, we encode a *separator*.

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 $|S| \leq 2^{O(m+\log\log U)}$.*

Theorem 7 states that 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.

- $|R| + |W| = \Omega(\ell\sqrt{n}\lg n)$ – 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