CS 573: Topics in Algorithms: Advanced Data Structures **Spring 2006**

Lecture 10 — February 16

Lecturer: Jeff Erickson *Scribe: Reza Zamani-Nasab*

10.1 Lower Bounds for Dynamic Connectivity

We refer to the problem of *dynamic connectivity* as the following problem:

> For every positive integer n, design a data structure to maintain an undirected graph on n vertices, supporting the following actions: inserting an edge (represented by insert(u,v)), deleting an edge (represented by delete(u,v)), and detecting whether two vertices are connected to each other in the graph (represented by connected(u,v)).

Our final goal here is to present a lower bound on *the worst case time complexity*[1] of any data structure that solves this problem in *the cell probe model* of computation.

The same idea, together with more careful analysis, can be used to establish a trade-off lower bound for this problem. For a data structure that solves dynamic connectivity (in cell probe model), assume *the amortized update*[2] *time* be $t_u(n)$ and *the amortized query time* be $t_q(n)$, where n indicates the order of graph. It can be shown that [3]

$$\min\{t_q, t_u\} \lg(\frac{\max\{t_q, t_u\}}{\min\{t_q, t_u\}}) = \Omega(\lg n)$$

This shows that both the solution of Thorup [1] to this problem with $t_u = O(\lg n(\lg\lg n)^3)$ and $t_q = O(\lg n/\lg\lg\lg n)$, and that of Holm et. al. [2] with $t_u = O(\lg^2 n)$ and $t_q = O(\lg n/\lg\lg n)$ can be considered as optimal solutions.

10.1.1 The Cell Probe Model

This is a rather general model of computation. The main object in this model is a *"permanent"* random access memory with 2^w words (which we call *cells*), each of size w bits[3]. When an algorithm in this model processes an input of size n, we assume that $w = \theta(\lg n)$ [4]. As you may expect, the algorithm is allowed to store anything it wants in the cells, in

[1] Actually it can be shown that the result is true in amortized sense and in the presence of randomization

[2] insertion/deletion

[3] Note that the number of words is limited to 2^w to make memory words address-accessible, since addresses must be storable in the cells

[4] $w = \Omega(\lg n)$ enables the algorithm to read and remember the input, and $w = O(\lg n)$ avoids having *weired* algorithms in cell probe model, like sorters with time bound $o(n \lg n)$

particular address of another cell to be used later in order to read/write from/to that cell (unlike models such as *pointer machine model* [5]).

The life cycle of a typical algorithm in the cell probe model can be divided into a number of epochs. At the start of every epoch, the algorithm accesses a number of cells of the "permanent" memory to read or write something. This follows by a processing step in which the algorithm never accesses the "permanent" memory. Instead it can use an unlimited amount of "transient" memory as scratch paper to do any computation that it wants. At the end of the epoch, the algorithm accesses some cells of the "permanent" memory to read/write something and then the infinite "transient" memory is *cleared* [6].

When an algorithm $\mathcal{A}$ in this model processes an input x, the total number of "permanent" memory accesses during the course of processing of x, is considered as the (time) complexity of $\mathcal{A}(x)$ [7].

10.1.2 The Approach

We present a random process which generates hard-to-handle inputs (requests of update/query). We will show that given a data structure that solves the dynamic connectivity problem, the average [8] time [9] to handle a hard input has a lower bound. This means that for a certatin input, the running time of the algorithm must be at least that lower bound, which is the desired result.

More precisely, fix a data structure $\mathcal{D}$ which solves the dynamic connectivity problem. Consider a graph on n (w.l.g. $n = 4^m$)[10] vertices where vertices are points of a $\sqrt{n} \times \sqrt{n}$ grid. Among vertices of every two adjacent column, there is a perfect matching, and these matchings are the only edges present in this graph. We will generate the hard requests using this graph, which specifically means that obtained lower bound will be true even if we restrict the problem to some special classes of graphs, like planar graphs. Label the columns/rows of the grid with $\{0, 1, 2, \cdots, \sqrt{n} - 1\}$ from left to right/top to bottom. Let π_i indicate the permutation of the rows induced by the perfect matching between column $i - 1$ and i. We define two actions on this graph:

- UPDATE(i,π) : Sets $\pi_i \leftarrow \pi$ and returns nothing.
- VERIFY(i,π) : Returns the result of this check as a boolean: $\pi_i \circ \pi_{i-1} \circ \cdots \circ \pi_1 \stackrel{?}{=} \pi$.

[5] A model of computation whose memory consists of an unbounded collection of registers, or records, connected by pointers. Each register may contain an arbitrary amount of additional information. No arithmetic is allowed to compute the address of a register. The only way to access a register is by following pointers[4].

[6] This, in particular, means that an epoch can access information of a previous epoch just by reading something from "permanent" memory

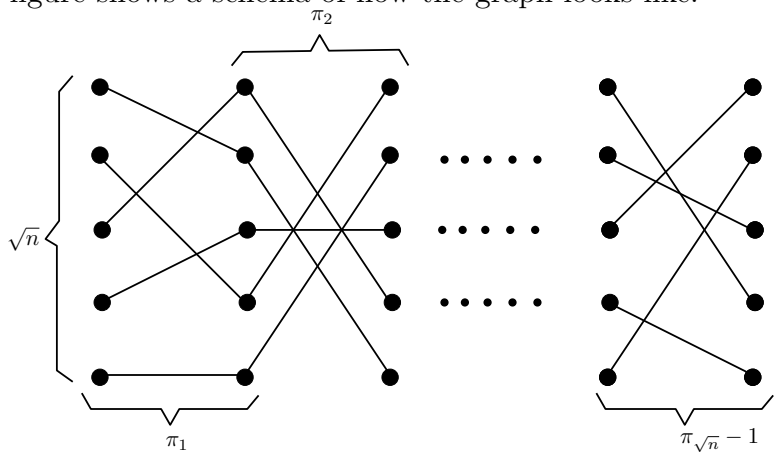
[7] So this means that we can assume "the permanent" memory is a far and slow storage device, like a central database which must be kept updated, and the transient memory is a local and fast storage media, so we charge the algorithm just for accessing the slow one

[8] over the random bits of the process that generates the hard input

[9] that is complexity in cell probe model

[10] w.l.g. we can assume $n = 4^m$, to keep $\sqrt{n}$ and bit-reversal operation well-defined

The following figure shows a schema of how the graph looks like:



We notice that to have a data structure to store this kind of graph and supports both UPDATE and VERIFY operations, we can use data structure $\mathcal{D}$ to encode the graph and then each of VERIFY and UPDATE boils down to a sequence of $O(\sqrt{n})$ updates and queries from corresponding $\mathcal{D}$. So it suffices to establish a lower bound on the efficiency of a data structure that stores this kinds of graphs such that it can handle both UPDATE and VERIFY. Therefore from now on, we will try to put a lower bound on the efficiency of such data structures.

The previously mentioned randomized process is:

for $i \leftarrow 1$ **to** $\sqrt{n} - 1$ **do**
  UPDATE($\beta(i, \lceil \lg \sqrt{n} \rceil)$, RandomPerm($\sqrt{n}$))
  VERIFY($\beta(i, \lceil \lg \sqrt{n} \rceil)$)
end for

where $\beta(i, k)$ indicates the bit-reversal transform of i when it is represented by k bits[11] and RandomPerm(i) is considered as a function that returns a random permutation of $\{0, 1, 2, \cdots, i-1\}$, uniformly and independently.

Actually we will use a somewhat different randomized process, that makes the analysis easy. Then we will show that almost the same analysis works for the above process. The modified process uses a function SUM(i) which actually returns $\pi_i \circ \pi_{i-1} \circ \cdots \circ \pi_1$.

for $i \leftarrow 1$ **to** $\sqrt{n} - 1$ **do**
  UPDATE($\beta(i, \lceil \lg \sqrt{n} \rceil)$, RandomPerm($\sqrt{n}$))
  SUM($\beta(i, \lceil \lg \sqrt{n} \rceil)$)
end for

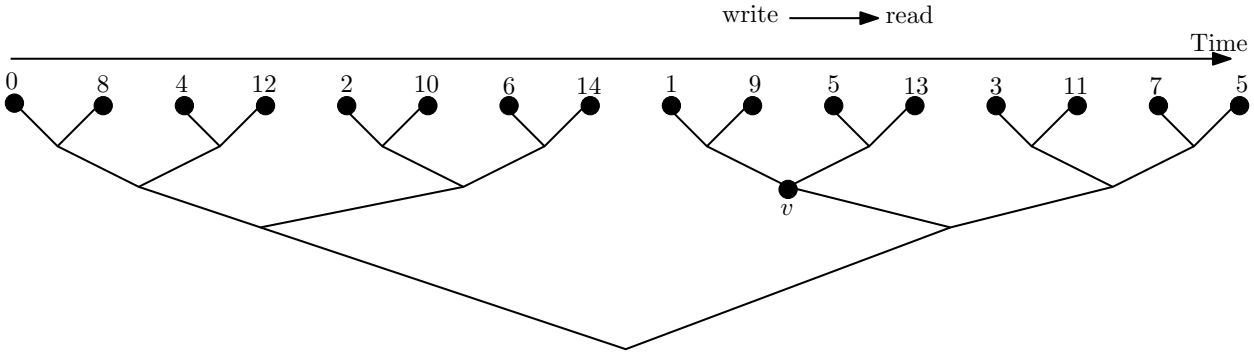
In the rest of this lecture we will show that this modified version of randomized process, generates a distribution of sequences of requests, such that any data structure that handles UPDATE and SUM will spend at least $\Omega(n \lg n)$ amount of time on at least one of those sequences. In particular, this together with the fact that we can prove the same lower bound for the original version of randomized inputs, means that $\mathcal{D}$ (our data structure that solves dynamic connectivity and is used to *simulate* UPDATE and VERIFY) must spend

[11] For example, $\beta(1, 4) = 8$, i.e. $0001 \rightarrow 1000$

$\Omega(n \lg n/(\sqrt{n} \times \sqrt{n})) = \Omega(\lg n)$ on at least one operation in at least one sequence, which proves the desired lower bound on the worst case time of dynamic connectivity problem.

10.1.3 Analysis of The Modified Hard Inputs

Consider the sequence of UPDATE and SUM 's of the modified version of hard inputs, in the order of their occurrence in time, written from left to right. We want to lower bound the amount of cell accesses for an algorithm in cell probe model that can handle the permutation problem correctly. For the sole purpose of analysis, we assume a balanced binary tree on this sequence. Whenever the algorithm reads some cell, we will charge a vertex of this tree, so this tree organizes counting of read operations. An illustration follows.



Now consider a reading of a cell which happens at some point of time (figure shows a read over the node labeled 13). The content that is read, must be written by some write operation previously, at some point of time, inside the operation denoted by one of the leaves of the binary tree. We charge this read operation, to the least common ancestor of these two nodes (in the figure, the read is charged to the node v). Note that every read operation will be assigned to a unique node of the tree, so to count the number of read operations we just need to count the charges over all the nodes of the tree. Since the sequence of reads is generated by a random process, and since expectation of a random variable is a linear function, we can calculate the expected charge of a node and sum those expectations over all nodes, to have expectation of the total number of read operations.

So now we concentrate on a certain node v of this binary tree, to compute its expected charge. We just need a lower bound. Let W be the set of all write operations performed in the leaves of the left subtree of v (both address of the cell and the content written to the cell, is kept inside a write operation in W) and similarly R be the set of all read operations performed in the leaves of the right subtree of v. Since $R \supseteq R \cap W$, it suffices to just lower bound $|R \cap W|$. To achieve this we will show that having $R \cap W$ at hand, together with some additional information provided by a magic LITTLE BIRDIE[12] will enable us to compute all the random permutations used in the left subtree leaves of the node v. Specifically we assume that the LITTLE BIRDIE tells us all the state of the permanent memory

[12] The information provided by the LITTLE BIRDIE can not hurt a lower bound, since the algorithm has the option to ignore the additional information

before left subtree, all updates in the right subtree, and all queries (i.e SUM results) in the right subtree together. Now we just note that perfect interleaving provided by bit-reversal order of the modified process, together with this information, enables us to compute all the permutations of the left subtree of v. But now assuming that v has l leaves, there are $\Omega(l)$ of those random permutations in the left subtree. Every representation of those will need $\Omega(l \lg(\sqrt{n}!)) = \Omega(l\sqrt{n} \lg n)$ bits *in expectation*. Since our special representation has size $|R \cap W|\theta(\lg n)$ (our cells are $\theta(\lg n)$ bits), we must have $\mathbb{E}(|R \cap W|) = \Omega(l\sqrt{n})$. So this means that the expected charge on the node v is $\Omega(l\sqrt{n})$. Using linearity of expectation, the total of expected charges on every level of the tree is $\Omega(n)$, and the total in whole tree is $\Omega(n \lg n)$, which is the desired result.

As previously mentioned, we have proved the lower bound for the modified version of input requests, which can not be simulated using a data structure that solves dynamic connectivity and so this does not give a lower bound for dynamic connectivity problem. To get a lower bound for dynamic connectivity problem, it suffices to establish the same lower bound for the original sequence of random requests (those that use VERIFY instead of SUM).

The idea to overcome this difficulty is to (kind of) simulate SUM operations using VERIFY operations. Namely, we notice that it seems that using the information that LITTLE BIRDIE provides together with elements of $R \cap W$, we can obtain the result of all the SUM operations in the right subtree just by *simulating* a lot of VERIFY requests for *all the possible permutations*, the idea is that the correct SUM value will return a "True" from VERIFY function. Unfortunately, some other wrong permutations may also get a "True" value from VERIFY function. The reason is that simply when we are simulating VERIFY for a permutation, that is not the correct value which SUM returns, it may read some cells which are written in left subtree but *are not* in $R \cap W$ and the simulation goes wrong from this point, and it might result in a "True" from VERIFY.

The idea for removing this difficulty is that it suffices that we know whether a given read request is in W or not. If our simulation has a tool to detect this, then first we can simulate the VERIFY of the correct permutation simply because it never reads some cell in the right subtree from $\overline{R \cap W}$ (by definition of R and W), and second as soon as a simulation tries to read something from $\overline{W}$, we can abort the simulation (since we know the correct permutation never asks for such a read).

The simplest way to implement this idea is to add W to our previous information (which are $R \cap W$ and LITTLE BIRDIE informations). This approach doesn't work because we need a lot of bits to encode the whole W. The workaround for this is to consider a separating family [13] $\mathcal{C}$, for all the sets which may potentially be $R - W$ and $W - R$. Then it is easy to see that having the LITTLE BIRDIE informations, $R \cap W$ and the index of the set in $\mathcal{C}$ that separates $R - W$ and $W - R$, we can rebuild all the random permutations used in the left subtree of v. So like the previous result, we must have $\mathbb{E}(\lg |\mathcal{C}| + |R \cap W| \lg n) = \Omega(l\sqrt{n} \lg n)$.

It is easy to prove the following lemma using standard probabilistic approach:

[13] A separating family for all subsets of U which are of size at most m, is a collection $\mathcal{C}$ of subsets of U such that for any two disjoint subsets of U, A and B, where $|A|, |B| \leq m$, there is at least one set $X \in \mathcal{C}$ such that $A \subseteq X$ and $B \subseteq \overline{X}$

Lemma 10.1. *There is a separating family of size $2^{O(m+\lg\lg|U|)}$ for the subsets of set U which are of size at most m.*

Combine the lemma with $\mathbb{E}(\lg|\mathcal{C}| + |R \cap W| \lg n) = \Omega(l\sqrt{n}\lg n)$ to obtain

$$\mathbb{E}(|R \cap W|)O(\lg n) + O(\mathbb{E}(|R| + |W|) + \lg\lg n) = \Omega(l\sqrt{n}\lg n)$$

Now if $\mathbb{E}(|R| + |W|) = o(l\sqrt{n}\lg n)$ then the above immediately results that $\mathbb{E}(|R \cap W|) = \Omega(l\sqrt{n})$, which is the desired result. Otherwise if $\mathbb{E}(|R| + |W|) = \Omega(l\sqrt{n}\lg n)$, then since the subtree of v has $O(l)$ leaves and in each leaf we don't have more than $O(\sqrt{n})$ dynamic connectivity operations, at least one of those operations must take $\Omega(\lg n)$ time (for at least one sequence of requests), which is again the desired result.

Bibliography

[1] M. Thorup., *Near-optimal fully-dynamic graph connectivity.* In Proc. 32nd ACM Symposium on Theory of Computing (STOC), pp. 343-350, 2000.

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

[3] Patrascu, Mihai and Erik D. Demaine, *Lower bounds for dynamic connectivity*, Proc. 36th Symp. Theory of Computing (STOC 2004), ACM, 2004

[4] http://www.nist.gov/dads/HTML/pointermachn.html

[5] http://theory.lcs.mit.edu/classes/6.897/spring05/lec/lec07.pdf