

# On Dynamic Bit-Probe Complexity

Corina E. Pătraşcu<sup>1</sup> and Mihai Pătraşcu<sup>2</sup>

<sup>1</sup> Harvard University, [patrascu@fas.harvard.edu](mailto:patrascu@fas.harvard.edu)

<sup>2</sup> MIT, [mip@mit.edu](mailto:mip@mit.edu)

**Abstract.** This paper presents several advances in the understanding of dynamic data structures in the bit-probe model:

- We improve the lower bound record for dynamic language membership problems to  $\Omega\left(\left(\frac{\lg n}{\lg \lg n}\right)^2\right)$ . Surpassing  $\Omega(\lg n)$  was listed as the first open problem in a survey by Miltersen.
- We prove a bound of  $\Omega\left(\frac{\lg n}{\lg \lg \lg n}\right)$  for maintaining partial sums in  $\mathbb{Z}/2\mathbb{Z}$ . In the course of the proof, we show how to use the chronogram technique to obtain logarithmic bounds in the cell-probe model. This is an important methodological progress, even though such bounds were obtained recently through a different technique.
- We prove a surprising and tight upper bound of  $O\left(\frac{\lg n}{\lg \lg n}\right)$  for predecessor problems. We use this to obtain the same upper bound for dynamic word and prefix problems in group-free monoids.

## 1 Introduction

Bit-probe complexity can be considered a fundamental measure of computation. When talking about algorithms (branching programs), it is almost always preferred to cell-probe<sup>3</sup> complexity. In data structures, cell-probe complexity is used more frequently, but the machine independence and overall cleanness of the bit-probe measure have made it a persistent object of study since the dawn of theoretical computer science. Nonetheless, many of the most fundamental questions are not yet understood. In this paper, we address this on several fronts.

*Record Lower Bound.* We prove a lower bound of  $\Omega\left(\left(\frac{\lg n}{\lg \lg n}\right)^2\right)$  for dynamic connectivity. This problem asks to maintain an undirected graph, under insertion and deletion of edges, and queries asking whether two nodes are in the same connected component. The best upper bound is  $O(\lg^2 n \cdot (\lg \lg n)^3)$  [1], so our lower bound is optimal up to doubly logarithmic factors.

Our lower bound is the highest known bound for an explicit dynamic language membership problem. The previous record was  $\Omega(\lg n)$ , shown in [2]. A survey on cell probe complexity by Miltersen [3] lists improving this bound as the first open problem among three major challenges for future research. Our lower bound is

---

<sup>3</sup> Of course, the bit-probe model is an instantiation of the cell-probe model with one-bit cells. For conciseness, however, we shall use “cell-probe” in a more restricted sense, to refer to the cell-probe model with cells of  $\Theta(\lg n)$  bits.

based on the recent technique of Pătraşcu and Demaine [4], which proved the first  $\Omega(\lg n)$  bounds in the cell-probe model. While our contribution is mostly a series of technical tricks necessary for the bit-probe model, it should be noted that in no way is our  $\tilde{\Omega}(\lg^2 n)$  bound a mere echo of an  $\Omega(\lg n)$  bound in the cell-probe model. Indeed,  $\Omega(\frac{\lg n}{\lg \lg n})$  bounds in the cell-probe model have been known for one and a half decades (including for dynamic connectivity), but the bit-probe record has remained just the trivially higher  $\Omega(\lg n)$ . To our knowledge, our bound is the first to show a quasi-optimal  $\tilde{\Omega}(\lg n)$  separation between the cell-probe and bit-probe complexity, when the cell-probe complexity is superconstant.

*Lower Bound for Maintaining Partial Sums.* One of the most fundamental problems is maintaining partial sums. This problem asks to maintain an array  $A[1..n]$  under an  $\text{UPDATE}(k, x)$  operation, which changes  $A[k] \leftarrow x$ , and a  $\text{SUM}(k)$  operation, which asks for a partial sum  $\sum_{i=1}^k A[i]$ . Because we are concerned with the bit-probe model, it is most natural to consider the case when all operations are done in  $\mathbb{Z}/2\mathbb{Z}$ , i.e. we are interested in prefix parity. This can be generalized to any fixed cyclic group  $\mathbb{Z}/k\mathbb{Z}$ , without any change in the bounds discussed below. The only known upper bound is the classic  $O(\lg n)$ , based on a balanced binary tree. It is widely believed that this is optimal.

Fredman [5] proved a lower bound of  $\Omega(\frac{\lg n}{\lg \lg n})$  by considering the following greater-than problem. The problem asks to support a pair of update-query operations. In the update stage, the algorithm is given a number  $a \in [n]$ , which it must “remember”. In the query stage, the algorithm is given another number  $b \in [n]$ , and it must determine whether  $b > a$ . Observe that there is a trivial reduction from this problem to the partial sums problem, with one update and one query. It is quite tempting to believe that one cannot improve past the trivial upper bound  $T_u = T_q = O(\lg n)$ , since, in some sense, this is the complexity of “writing down”  $a$ . However, we show below that Fredman’s bound is optimal for the greater-than problem. Therefore, one needs a different strategy to improve the lower bound for maintaining partial sums.

It is natural to approach the problem in the framework of [4], which could prove an  $\Omega(\lg n)$  bound in the cell-probe model (of course, the group was  $\mathbb{Z}/n\mathbb{Z}$  in that case, matching the cell size). Applying the tricks we developed for dynamic connectivity will only reprove the old  $\Omega(\frac{\lg n}{\lg \lg n})$  bound. For reasons discussed below, obtaining  $\Omega(\lg n)$  seems quite difficult.

We can, nonetheless, prove an  $\Omega(\frac{\lg n}{\lg \lg \lg n})$  bound, which is only a triply-logarithmic factor away from the upper bound! The technique of this bound is at least as important as the result itself. The proof uses the chronogram technique of Fredman and Saks [6]. However, it is well known that the classic chronogram technique can only prove  $\Omega(\frac{\lg n}{\lg \lg n})$ . We present a small, yet very important variation, which brings a considerable strengthening of this technique: with this improvement, the chronogram technique can prove lower bounds of  $\Omega(\lg n)$  in the cell-probe model! To fully appreciate this development, one must remember that the chronogram technique was virtually the only available technique for proving dynamic lower bounds before the work of [4]. At the same time, obtaining

a logarithmic bound in the cell-probe model was viewed as one of the most important problems in data-structure lower bounds. It is now quite surprising to find that the answer has always been this close.

With this variation, the barriers between the chronogram method and the technique of [4] become rather weak, to the point where they can be considered the same technique. Nonetheless, it is important to achieve this strengthening, because it can be easier to work with the chronogram technique (our current lower bound is a living proof to this). On the other hand, the approach of [4] can be used in a fast-updates / slow-queries situation (like that required in our dynamic connectivity lower bound), whereas the chronogram ideas can only be applied to a fast-queries / slow-updates tradeoff.

*Upper Bound for Predecessor Problems.* As mentioned already, we can achieve an  $O(\frac{\lg n}{\lg \lg n})$  upper bound for Fredman’s greater-than problem. In fact, we can achieve the same bound for several predecessor-type problems. Consider for example the classic predecessor problem: maintain a dynamic set  $S$ , under queries to determine (some information about) the predecessor in  $S$  of a given number. A quite unfortunate fact is that we cannot determine the actual predecessor in  $o(\lg n)$ , because the output itself has this many bits of entropy. But we can recover some constant amount of information about the predecessor (a stored “color”), which proves to be enough for many purposes. For our classification of dynamic prefix problems, it is important to generalize this slightly to the colored  $k$ -predecessor problem: the query asks for the colors of the  $k$  predecessors of a given number. Here  $k$  is part of the definition of the problem, so it is a constant.

In the interest of dynamic prefix problems, we also study a more unusual stabbing problem. The problem itself is an interesting trick to circumvent finding an exact predecessor in many cases. We have to maintain a dynamic set  $S = \{b_1, b_2, \dots\}$  under the following query operation: given  $j \in (b_i, b_{i+1})$ , the query determines a value in  $(b_i, b_{i+1})$ , which is only a function of  $b_i$  and  $b_{i+1}$ , but not of  $j$  or  $i$ . Imagine that the elements of  $S$  break  $[n]$  into segments. The query must then produce a representative for the segment stabbed by  $j$ , which is inside the segment, but is independent of the actual choice of  $j$ . Adding or removing an element merges or splits segments. The representatives of these affected segments may change arbitrarily, but those of any other segments must remain the same (because they are only functions of the end-points). The segment representative has  $\lg n$  bits, so it is maybe surprising that one can be found in  $O(\frac{\lg n}{\lg \lg n})$  time. Of course, we do not ask the query to actually write down the representative – that would be impossible; all we need is that the representative is determined by the bit probes (and the query input).

*Dynamic Word and Prefix Problems.* Dynamic prefix problems are defined like the partial sums problem, except that all additions take place in an arbitrary finite monoid. The word problem is identical to the prefix problem, except that queries only ask for the sum of the entire array, not an arbitrary prefix. The problem is defined by the monoid, so the monoid is considered fixed (and constants may depend on it). Our aim is to understand the complexity of the problem

in terms of the structure of the monoid. This line of research was inspired by the intense study of parallel word problems, which eventually led to a complete classification. Both in the parallel and in the dynamic case, it can be seen that many fundamental problems are equivalent to word and prefix problems for certain classes of monoids. Examples include partial sums modulo some value, the predecessor problem, and the priority queue problem. We can also show that existential range queries in one dimension are captured by a class of monoids. In general, we would expect any fundamental problem of a certain one-dimensional flavor to be represented, making word problems an interesting avenue for complexity-theoretic research.

The seminal paper of Frandsen, Miltersen and Skyum [7] achieved tight bounds for many classes of monoids, both in the bit-probe and in the cell-probe models, but the classification is incomplete in both cases. In this paper, we further the classification for the bit-probe model in two ways. First, our lower bound for the partial-sums problem in  $\mathbb{Z}/k\mathbb{Z}$  applies to the prefix problem in any monoid containing groups, and for the word problem in monoids containing a certain kind of externally noncommutative cycles [7, Theorem 2.5.1]. Second, we derive an upper bound of  $O(\frac{\lg n}{\lg \lg n})$  for the word and prefix problems in group-free monoids. This uses the same algebraic toolkit as used by [7] in the cell-probe model, but our application needs some interesting new ideas to handle the idiosyncrasies of the bit-probe model. An interesting open question is the complexity of the word problem in monoids containing only externally-commutative cycles. By analyzing examples of monoids in this class, we have come to believe that the answer is  $\Theta(\frac{\lg n}{\lg \lg n})$ , but a full proof of this seems to require progress on the algebraic side. Further details will be given in the full version of this paper.

## 2 Lower Bound for Dynamic Connectivity

**Theorem 1.** *The bit-probe complexity of dynamic connectivity is  $\Omega((\frac{\lg n}{\lg \lg n})^2)$ , even if amortization and randomization are allowed.*

We first describe the dynamic graph used in our lower-bound construction; refer to figure 2. The vertex set is roughly given by an integer grid of size  $\sqrt{n} \times \sqrt{n}$ . The edge set is given by a series of permutation boxes. A permutation box connects the nodes in a column to the nodes in the next column arbitrarily, according to a given permutation in  $S_{\sqrt{n}}$ . Notice that the permutations decompose the graph into a collection of  $\sqrt{n}$  paths. As the paths evolve horizontally, the  $y$  coordinates change arbitrarily at each point due to the permutations. In addition to this, there is a special test vertex to the left. This is connected to a subset of the vertices in the first column.

We now describe the hard sequence of operations. The shape of the graph allows us to implement a partial sums problem over  $S_{\sqrt{n}}$ . The partial sums macro-operations are implemented as follows:

**UPDATE( $i, \pi$ ):** sets  $\pi_i = \pi$ . This is done by removing all edges in permutation box  $i$  and inserting new edges corresponding to the new permutation  $\pi$ . This uses  $O(\sqrt{n})$  elementary operations.

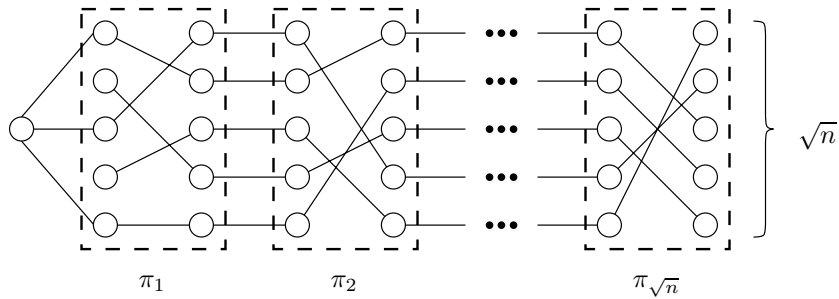


Fig. 1. Our graphs can be viewed as a sequence of  $\sqrt{n}$  permutation boxes.

SUM( $i$ ): returns  $\sigma = \pi_1 \circ \dots \circ \pi_i$ . We use  $O(\lg n)$  phases, each one guessing a bit of  $\sigma(j)$  for all  $j$ . Phase  $k$  begins by removing all edges incident to the test node. Then, we add edges from the test vertex to all vertices in the first column, whose row number has a one in the  $k$ -th bit. Then, we test connectivity of all vertices from the  $i$ -th column and the test node, respectively. This determines the  $k$ -th bit of  $\sigma(j)$  for all  $j$ . In total, SUM uses  $O(\sqrt{n} \lg n)$  elementary operations.

This construction differs from the one used in the cell-probe model. There, it was possible to prove a lower bound for a VERIFY operation, which verifies a given partial sum, rather than compute one from scratch. This does not seem possible in the bit-probe model. The reason is quite technical: the “separators” used for the cell-probe proof are too large. Unfortunately, known lower bounds show that those separators are actually optimal, so the approach fails altogether. In the bit-probe model, we are forced to revert to the SUM operation. Interestingly, we can still prove a good lower bound even if we blow up the queries by a  $\lg n$  factor: the cost is just one of the  $\lg n$  factors in the denominator.

To fully specify the hard sequence, we must still say how UPDATES and SUMS are generated. We choose  $m = n^\epsilon$  macro-operations randomly and independently ( $\epsilon > 0$  can be an arbitrary small constant). Because queries are  $O(\lg n)$  time more expensive, each operation is chosen to be a query with probability  $\frac{1}{\lg n}$ . The arguments of each operation ( $i$ , possibly  $\pi$ ) are picked uniformly at random. Let  $t$  be the running time of a connectivity operation. The total expected running for  $m$  operations is  $O(t\sqrt{n}(1 - \frac{1}{\lg n})m + t\sqrt{n} \lg n \cdot \frac{1}{\lg n}m) = O(mt\sqrt{n})$ . We shall prove an  $\Omega(m\sqrt{n}(\frac{\lg n}{\lg \lg n})^2)$  lower bound, implying the desired bound on  $t$ .

*Proof of the Lower Bound.* Following [4], we consider a tree whose leaves represent the entire sequence of operations in time order. Any bit probe is characterized by when it occurs and when the bit was last written. These times are given by two leaves in the tree. For every node in the tree, we will lower bound the number of bit probes with the write time in the subtree of the node, and the read time in a subtree of a right sibling of the node. We then add up these

lower bounds for all nodes, to obtain the overall bound. The correctness of this strategy follows by two observations. First, we never double count a bit probe: it is only counted for the node immediately under the lowest common ancestor of the read and write times. Second, we can sum the lower bounds because they all hold in the average case, under the same distribution.

We choose the branching factor of the time-tree to be  $\Theta(\lg n)$ , as suggested by the following intuition. Queries are fewer than updates by a logarithmic factor (because they are expensive). For maximal hardness, we must show that essentially all information in a series of updates is extracted. To do this, we must reuse most queries  $\Theta(\lg n)$  times. Most nodes have  $\Theta(\lg n)$  right siblings. If the node has  $L$  leaves under it, we expect  $\Theta(L)$  updates in its subtree, and  $\Theta(\lg n \cdot L \frac{1}{\lg n}) = \Theta(L)$  queries in the subtrees of its right siblings, which allows all information to be extracted. Thus, a query is effectively reused  $\Theta(\lg n)$  times on each level – in the lower bound for all left siblings of its ancestor.

**Lemma 2.** *Consider two adjacent intervals of operations, the left one of size  $L$ , and the right one of size  $\Theta(L \lg n)$ . Let  $r$  be the number of bit probes executed in the right interval, and  $c$  the number of probes from the right interval, which access a bit last changed during the left interval. Then  $E[\lg \binom{r}{c}] = \Omega(L\sqrt{n} \lg n)$ .*

*Proof.* Assume that we are told the entire sequence of macro-operations, except the ones in the left interval. Given this information, the sequence of answers to the SUM queries from the right interval has Kolmogorov complexity  $\Omega(L\sqrt{n} \lg n)$ ; see [4, Lemmas 5.3 and 5.4]. In essence, we expect  $\Theta(L)$  updates in the left interval, and  $\Theta(L)$  queries in the right interval. Further, we expect these to interleave almost perfectly, causing the answers to  $\Omega(L)$  queries to be independent.

We now propose the following encoding for the answers to the queries: encode  $r$ ,  $c$ , and the subset of the  $r$  bit probes which touch a bit changed during the left interval, and not yet changed in the right interval. This takes  $E[\lg \binom{r}{c}] + O(\lg n)$  bits on average. Note that we only encode a  $c$ -subset of  $[r]$ ; this identifies the relevant bit probes, but doesn't, for example, identify the addresses they probe. To decode, we first simulate the data structure's behavior before the left interval (we know everything from the past). We then simulate it for the right interval; implicitly we will recover the answers to the queries. This simulation needs to recover the values of all probed bits. If the bit location was written during the simulation of the right interval, we already know the value. Otherwise, we check whether the bit probe is in the set of  $c$  special bit probes. In this is true, the value is the negation of what it was before the left interval began; otherwise, it is the old value from before the left interval.  $\square$

Now consider an arbitrary level of the tree. We apply the lemma by setting the left interval to the subtree of a node, and the right interval to the union of the subtrees of its right siblings. We do this only for nodes which are in the first half of their siblings, so that the right interval is  $\Theta(\lg n)$  times bigger. For some node  $i$ , we obtain  $E[\lg \binom{r_i}{c_i}] = \Omega(L_i \sqrt{n} \lg n)$ . Let us sum this up for all nodes on a level. Clearly,  $\sum L_i = \frac{m}{2} = \Theta(m)$ . By linearity of expectation, the left hand

side becomes  $E[\lg \prod \binom{r_i}{c_i}]$ . It is well known that  $\prod \binom{r_i}{c_i} \leq \left(\sum \frac{r_i}{c_i}\right)$ . Now,  $\sum r_i = \Theta(T \lg n)$ , where  $T$  is the total number of bit probes. This is because each bit probe is counted at most once for every sibling of its ancestor on the current level. Let  $\sum c_i = C_\ell$ , where  $\ell$  is the current level. We can now obtain  $E[\lg \binom{\gamma T \lg n}{C_\ell}] = \Omega(m\sqrt{n} \lg n)$ , for an appropriate constant  $\gamma$ . Now let us sum over all levels  $\ell$ . As explained in the beginning,  $\sum C_\ell \leq T$ . Then, we obtain  $E[\lg \binom{\gamma T \lg n \cdot \lg n / \lg \lg n}{T}] = \Omega(m\sqrt{n} \lg n \frac{\lg n}{\lg \lg n})$ , which implies  $E[T]O(\lg \lg n) = \Omega(m\sqrt{n} \frac{\lg^2 n}{\lg \lg n})$ .

This completes the proof of our  $\Omega\left(\left(\frac{\lg n}{\lg \lg n}\right)^2\right)$  lower bound. As explained already, one of the  $\lg \lg n$  factors is lost because we need to increase the complexity of queries by a  $\lg n$  factor, which translates into a higher branching factor for the tree, and a depth of just  $O\left(\frac{\lg n}{\lg \lg n}\right)$ . The other  $\lg \lg n$  factor is lost from our encoding. Intuitively, Lemma 2 should state that  $c = \Omega(L\sqrt{n} \lg n)$ . This is because the right interval needs  $\Omega(L\sqrt{n} \lg n)$  bits of information, and, intuitively, this can only come from so many bit probes. Unfortunately, the fact that a bit was not modified during the left interval is also information, even if it is not clear how such information could ever be used in an upper bound. The factor of  $\lg \binom{r}{c} \approx c \lg n$  seems very hard to avoid.

### 3 Lower Bound for Partial Sums in Cyclic Groups

Let us review the ideas behind the chronogram method, at an intuitive level. First, we generate a sequence of random updates, ended by one random query. Looking back in time from the query, we partition the updates into exponentially growing epochs: for a certain  $t$ , epoch  $i$  contains the  $t^i$  operations immediately *before* epoch  $i - 1$ . We then argue that for all  $i$ , the query needs to read at least one cell from epoch  $i$  with constant probability. This is done as follows. The first  $i - 1$  epochs contain  $O(t^{i-1})$  updates. Let  $t_u$  be an upper bound on the number of cell probes performed by an update. Then, the “information” that subsequent epochs learn about epoch  $i$  is  $O(t^{i-1}t_u)$ . If  $t = \Omega(t_u)$  for a sufficiently large constant,  $t^i$  exceeds this by a constant factor, so we only have information about a fraction of the updates. Clearly, information about epoch  $i$  cannot be reflected in higher epochs (those occurred back in time). If a random query forces us to learn information about a random update from epoch  $i$ , we are forced to read a cell from epoch  $i$ , because this information is not available in any other epoch. This gives  $\Omega(1)$  probes in expectation to every epoch, so the lower bound on the query time is given by the number of epochs that we can construct, i.e.  $t_q = \Omega(\lg n / \lg t_u)$ . A tradeoff of this form was indeed obtained by [8], and is the highest tradeoff obtained by the chronogram method. Of course, these calculations only work when the output of the query matches the cell size.

We now describe the idea needed to improve the  $\lg t_u$  factor in the denominator. The analysis done by the chronogram technique is overly pessimistic, in that it assumes all cell probes in the first  $i - 1$  epochs access cells from epoch  $i$ , gathering a maximum amount of information. In the setup from above, this may actually be tight, because the data structure knows the division into epochs, and

can build a strategy based on it. However, we can randomize the construction of epochs to foil such strategies. We generate a random number of updates, followed by one query; since the data structure cannot anticipate the number of updates, it cannot base its decisions on a known epoch pattern. Due to this randomization, we expect each update to contribute  $O(t_u/t_q)$  probes into a random epoch (as before,  $t_q$  is roughly the number of epochs). Thus, we can obtain bounds of the form  $t_q \lg \frac{t_u}{t_q} = \Omega(\lg n)$ , implying  $\max\{t_u, t_q\} = \Omega(\lg n)$ .

One case in which this reasoning works is the partial sums problem in the cell-probe model (with the group  $\mathbb{Z}/n\mathbb{Z}$ ), for which the tradeoff is tight. Our bit-probe lower bound from below contains all ideas needed by this proof. As mentioned already, this is the largest cell-probe tradeoff known, and a different proof was described recently in [4]. In the bit-probe model, we only achieve a slightly weaker bound, because  $O(t_u/t_q)$  bit probes into an epoch give more than this number of bits of information. As for the connectivity bound, we must also account for the choice of the relevant  $O(t_u/t_q)$  probes from the total of  $t_u$ .

**Theorem 3.** *The bit-probe complexity of maintaining partial sums is  $\Omega(\frac{\lg n}{\lg \lg n})$ , even if amortization and randomization are allowed. In general,  $t_q(\lg \frac{t_u}{t_q} + \lg \lg t_q) = \Omega(\lg n)$ .*

Let  $M = n^\varepsilon$ , for any small  $\varepsilon > 0$ . We generate  $2M$  operations  $\text{UPDATE}(i, x)$ , for random  $i \in [n], x \in \mathbb{Z}/2\mathbb{Z}$ . Now pick  $m$  uniformly at random from  $\{M, \dots, 2M\}$ . After  $m$  operations, we insert a query to a random index. Looking back in time from the query, divide that last  $M$  operations into epochs. Epoch  $i$  has the  $t^i$  operations preceding epoch  $i - 1$ , where  $t$  will be specified later ( $t \geq 2$ ). The number of epochs is  $\Theta(\lg_t M) = \Theta(\lg_t n)$ . Let  $P_i$  be the number of bit probes from epochs 1 through  $i - 1$ , touching bits written during epoch  $i$ .

**Lemma 4.** *If  $i$  is chosen uniformly at random,  $E[\frac{P_i}{t^{i-1}}] = O(\frac{t_u}{\lg_t n})$ .*

*Proof.* We say that a bit probe has span  $j$  if  $t^{j-1} \leq w - r < t^j$ , where  $r$  and  $w$  are the read and write times (i.e. the index of the operation during which they occur). Let  $C_j$  be the total number of bit probes of span  $j$  that the  $2M$  updates would execute in the absence of an interleaving query. Fix  $i$  to some arbitrary value, and let  $f$  be the last operation in epoch  $i$ . Observe that  $f$  is a random variable depending on  $m$ . We are interested in bit probes with  $w \in (f - t^i, f], r \in [f, m]$ . All such bit probes must have span at most  $i + 1$ , because  $m - f < 2t^{i-1}$ . For all  $j$ , if  $r \geq f + t^{j-1}$ , the bit probe must have span at least  $j$ . We count all probes which satisfy these necessary conditions, thus upper bounding  $P_i$ .

Fix the  $2M$  updates arbitrarily. Because  $m$  is not known to the data structure, the first  $m$  updates behave as if no query ever happened. As  $m$  is chosen randomly, we are interested in the probes of span  $j$  from a random segment of size  $t^j$ . There are  $\Omega(M)$  choices for the position of this segment, and we choose uniformly at random between them. Then, the expected number of span- $j$  operations that are relevant is  $O(t^j \frac{C_j}{M})$ . For  $j = i$  or  $j = i + 1$ , we can upper bound the segment

size by  $m - f = O(t^{i-1})$ . Then,  $E_m[P_i] = O(t^{i-1} \frac{C_i + C_{i+1}}{M}) + \sum_{j < i} O(t^j \frac{C_j}{M})$ , so  $E_m[\frac{P_i}{t^i - 1}] = O(\frac{C_i}{M} + \frac{C_{i+1}}{M} + \sum_{j < i} \frac{C_j}{M \cdot t^{i-j}})$ .

Observe that  $\sum_i E_m[\frac{P_i}{t^i - 1}] \leq \sum_i (2 + \sum_{k=1}^{\infty} t^{-k}) \cdot O(\frac{C_i}{M}) = \sum_i O(\frac{C_i}{M})$ . By definition,  $\sum_i \frac{C_i}{M}$  is the amortized running time of an update. When updates are chosen uniformly at random,  $E[\sum_i \frac{C_i}{M}] \leq t_u$ . Thus, if  $i$  is chosen uniformly at random,  $E[\frac{P_i}{t^i - 1}] = O(\frac{t_u}{\lg_t n})$ .  $\square$

Now pick an epoch  $i$  at random. We show that with constant probability, the query needs to probe a bit from epoch  $i$ . Then, by linearity of expectation,  $t_q = \Omega(\lg_t n)$ . By the Markov bound, there is a  $1 - \delta$  probability that for the chosen  $i$ ,  $E[\frac{P_i}{t^i - 1}] = O(\frac{t_u}{\lg_t n})$ . Here the expectation is over the choice of  $m$ , and the random updates, but  $i$  is already fixed;  $\delta > 0$  is an arbitrarily small constant. If this relation does not hold, we make no claim. Otherwise, let  $p$  be the probability that a random query accesses a cell from epoch  $i$ . We want to prove that  $p$  is bounded away from 0. Then, be the union bound, there is a constant probability that a query needs a bit probe into epoch  $i$ .

Pick  $t^i$  random and independent queries, and imagine that all are run independently, starting with the state of the data structure after  $m$  updates. Given the choice of the queries, and the random updates from all epoch except  $i$ , the Kolmogorov complexity of the answers to all queries is  $\Omega(t^i)$ ; see [4, Lemmas 5.3 and 5.4]. This is rather intuitive, since the  $t^i$  queries are expected to interleave with the  $t^i$  updates from epoch  $i$ , so each query recovers one more bit of information about the unknown updates. We now propose an encoding for the answers to these queries.

First, we encode which bit probes executed in epochs  $[1, i - 1]$  read bits that were changed in epoch  $i$ . Let  $R_{i-1}$  be the total number of bit probes made by epoch  $[1, i - 1]$ . Since  $m$  is chosen randomly,  $E[R_{i-1}] = O(\frac{t^{i-1}}{t} t_u)$ . This part of the encoding takes  $O(\lg P_i + \lg R_{i-1}) + \lg \binom{R_{i-1}}{P_i} = O(P_i \lg \frac{R_{i-1}}{P_i})$  bits. Since  $(x, y) \mapsto x \lg \frac{y}{x}$  is convex, the expected size is  $O(E[P_i] \lg \frac{E[R_{i-1}]}{E[P_i]}) = O(t^{i-1} \frac{t_u}{\lg_t n} \lg \frac{t_u}{t_u / \lg_t n}) = O(t^{i-1} \frac{t_u}{\lg_t n} \lg \lg_t n)$ .

The second part of the encoding identifies which of the  $t^i$  queries reads at least one bit-probe from epoch  $i$ . If this number is  $Q$ , this part takes  $\lg \binom{t^i}{Q} = O(Q \lg \frac{t^i}{Q})$  bits. By convexity, the expected size is  $O(p t^i \lg \frac{t^i}{p t^i}) = O(t^i p \lg \frac{1}{p})$ . For the queries which read a bit from epoch  $i$ , we explicitly encode their answers; this takes  $p t^i$  bits in expectation.

It is easy to see that the encoding actually works. We first simulate the data structure up until the beginning of epoch  $i$ . Then, we simulate it after epoch  $i$ . This can be done because we know which bit probes read later get a changed bit because of epoch  $i$ . At the end, we simulate all queries which do not read bits from epoch  $i$ . For the rest of the queries, we know the answers.

The total size of the encoding is  $O(t^i p \lg \frac{1}{p}) + O(t^{i-1} \frac{t_u}{\lg_t n} \lg \lg_t n)$ . This must be  $\Omega(t^i)$ , from which we obtain  $p \lg \frac{1}{p} = \Omega(1) - O(\frac{1}{t} \cdot \frac{t_u}{\lg_t n} \lg \lg_t n)$ . Now choose  $t$  satisfying the equation  $t = C \cdot \frac{t_u}{\lg_t n} \lg \lg_t n$ , for a large enough constant  $C$ . For

this  $t$ , we obtain  $p \lg \frac{1}{p} = \Omega(1)$ , hence  $p = \Omega(1)$ . The lower bound is  $t_q = \Omega(\lg_t n)$ , from which we obtain  $t_q = \Omega(\lg n / (\lg \frac{t_q}{t_q} + \lg \lg t_q))$ .

## 4 Upper Bounds for Predecessor Problems

We only sketch the solution for the segment representatives problem. The other problems are discussed in Appendix A. We construct a trie with branching factor  $B = \Theta(\frac{\lg n}{\lg \lg n})$ , in which each element is represented by a root-to-leaf path. A node is marked active if any leaf under it is in the current set. Among the children of the node, the minimum and maximum active nodes are specially marked. In the appendix, we described how to maintain these markings in  $O(\frac{\lg n}{\lg \lg n})$  per insertion and deletion. It remains to implement queries. The main idea is to find the lowest common ancestor of the predecessor and the successor of the query point, and the children corresponding to the predecessor and the successor. Once this is known, it is easy to determine a canonical representative. First, take the bits corresponding to the LCA, which are common to the query, the predecessor, and the successor. Then, take the  $\lg B$  bits giving the child corresponding to the successor. Finally, pad with zeroes. This computation of the representative takes zero time, since all elements are already known.

To find the lowest common ancestor of the predecessor and successor, we proceed as follows. Traverse the leaf-to-root path from the query until an active node is found. Scan the children of the node. If there is both an active node to the left and one to the right of the query, we have found the answer. Otherwise, assume by symmetry there is an active node to the left, so we have only found the predecessor. We continue searching up the tree, until we find a node which is *not* marked as maximum among its siblings. At that point, we scan the right siblings, finding the next active one, which corresponds to the successor.

## 5 Group-Free Monoids

**Theorem 5.** *The dynamic prefix problem in any group-free monoid has bit-probe complexity  $\Theta(\frac{\lg n}{\lg \lg n})$ .*

We use a corollary of the Krohn-Rhodes decomposition [9], which was also used in the cell-probe case [7]. Unfortunately, our application of it is considerably more involved due to the idiosyncrasies of the bit-probe model: we go through considerable lengths to avoid an exact predecessor query.

**Theorem 6 ([9]).** *Let  $M$  be a finite nontrivial group-free monoid. One of the following holds:*

1.  $M \setminus \{1\} = \langle a \rangle = \{a, a^2, \dots, a^k = a^{k+1}\}$ ;
2.  $M$  is left simple, i.e.  $\forall a, b \in M \setminus \{1\}, ab = a$ ;
3.  $M = V \cup T$ , with  $V, T$  proper submonoids of  $M$ , and  $T \setminus \{1\}$  a left ideal of  $M$  (i.e.  $(\forall a \in M, b \in T : ab \in T)$ ).

We prove our upper bound by induction on the size of  $M$ . However, we need a stronger induction hypothesis, which assumes a solution for a slightly harder problem. We call this the prefix problem with breakpoints. Consider an array  $A[1..n]$ , in which an element can either be an element of  $M$ , or a breakpoint, denoted  $\mathbb{D}$ . The update operation can change any position of  $A$  to any element from  $M \cup \{\mathbb{D}\}$ . A query on an arbitrary position  $i$  must return the composition of  $A[j + 1], \dots, A[i]$ , where  $j$  is the predecessor of  $i$  in the set of breakpoints. For uniformity, we say  $A[0] = A[n + 1] = \mathbb{D}$ . Also, we assume breakpoints do not appear in consecutive positions. This can easily be arranged by doubling the array, and inserting an identity at every other position.

Assume by induction that the predecessor problem with breakpoints in all group-free monoids of size less than  $|M|$  has complexity  $O(\frac{\lg n}{\lg \lg n})$ . Now apply the decomposition theorem. Cases 1 and 2 are discussed in appendix B.

*Case 3* –  $M = T \cup V$ . Since  $|T|, |V| \leq |M| - 1$ , we have, by induction, solutions for the prefix problem with breakpoints, both running in  $O(\frac{\lg n}{\lg \lg n})$ . To obtain a data structure for  $M$ , we use the following components:

- a prefix structure for  $V$ , built over the array  $A^V[1..n]$ . The values of  $A^V$  are defined as follows: if  $A[i] \in V$ ,  $A^V[i] = A[i]$ ; otherwise ( $A[i] \in T \setminus V$  or  $A[i] = \mathbb{D}$ ), let  $A^V[i] = \mathbb{D}$ .
- a structure for finding segment representatives with the dynamic set being  $B = \{i \mid A^V[i] = \mathbb{D}\}$ . Denote the elements  $B = \{b_1, b_2, \dots\}$ .
- a prefix structure for  $T$ , built over the array  $A^T$ . If  $A[i] = \mathbb{D}$ , let  $A^T[i] = \mathbb{D}$ . The other elements are represented in a less straightforward way. For any segment  $(b_i, b_{i+1})$ , let  $A^T[\text{repr}(b_i + 1)] = \bigoplus_{j=b_i+1}^{b_{i+1}} A[j]$ . Note that all but  $A[b_{i+1}]$  are elements from  $V$ , but the composition is in  $T$  because of the ideal property. All elements which are not segment representatives are  $1_M$ .
- a simple array  $C$ . For any segment  $(b_i, b_{i+1})$ ,  $C[\text{repr}(b_i + 1)] = A[b_{i+1}]$ . Other values of  $C$  are ignored.

We claim that we can support the following operation: given any element  $j \in (b_i, b_{i+1})$ , compute the composition of the entire segment in  $A^V$ . Clearly, we can recover the composition of the elements between  $b_i + 1$  and  $j - 1$ , by running a prefix query in  $A^V$ . In addition, we can define a monoid  $W$ , which is “the reverse” of  $V$ :  $(\forall) a, b : a \oplus_W b = b \oplus_V a$ . Since  $|W| = |V| \leq m - 1$ , we can construct a prefix structure on  $W$  by the induction hypothesis. We maintain  $A^W[n - i + 1] = A^V[i], (\forall) i$ . Now, by running a prefix query in  $A^W$  at position  $n - j + 1$ , we are effectively running a suffix query in  $A^V$ . By composing the prefix and the suffix, we obtain the composition of the entire segment.

We can now easily support a prefix query to position  $i$ . First, we run a prefix query in  $A^T$  up to  $\text{repr}(i) - 1$ . This will give the desired partial sum up to the predecessor of  $i$  in  $B$ . To get the last part of the prefix, which consists only of elements from  $V'$ , we simply ask for the prefix up to  $i$  in  $A^V$ . This works because the predecessor from  $B$  of  $i$  is the most recent breakpoint in  $A^V$ .

Updates are done in two steps: first we change the old value to  $1_M$ , and then we change this to the new value. We distinguish the following cases:

- both the old and new values are in  $V$  (possibly  $1_M$ ). We update  $A^V[i]$ , and compute the composition of the entire segment containing  $i$ . Then, we compose this with  $C[\text{repr}(i)]$ , and we update  $A^T[\text{repr}(i)]$  to this new value.
- an element from  $T \setminus V$  is replaced by  $1_M$ . We update  $A^V$  and remove  $i$  from  $B$ , which merges two segments. We remove the old segment representatives from  $A^T$  (set  $A^T[\text{repr}(i-1)] = A^T[\text{repr}(i+1)] = 1_M$ ). Then, we add the representative for the new segment. The value of  $B[\text{repr}(i)]$  is given by the old value  $B[\text{repr}(i+1)]$ , corresponding to the right segment;  $A^T[\text{repr}(i)]$  is obtained by recomputing the composition of  $i$ 's segment, as above.
- $1_M$  is replaced by an element in  $T \setminus V$ . We update  $A^V[i]$  to  $\textcircled{D}$ , and insert  $i$  into  $B$ , thus splitting a segment. We first remove the old segment's representative, setting  $A^T[\text{repr}(i)] = 1_M$ , and then we add the two new representatives. The values in  $B$  are obvious:  $B[\text{repr}(i-1)]$  gets the new value of  $A[i]$ , and  $B[\text{repr}(i+1)]$  gets the old  $B[\text{repr}(i)]$  from the unsplit segment. The values in  $A^T$  are obtained by recomputing the compositions of the two segments.
- $\textcircled{D}$  is replaced by  $1_M$ . We update  $A^T[i]$  to  $1_M$ . Changing the other structures is identical to the case when an element from  $T \setminus V$  is removed.
- $1_M$  is replaced by  $\textcircled{D}$ . We propagate the change to  $A^T[i]$ . Changing the other structures is identical to the case when an element from  $T \setminus V$  is added.

*Acknowledgements.* Some of the results in this paper originate in a project of the first author for a research course at Harvard. She is grateful to Nick Rogers for careful reviewing of manuscripts and useful comments made on that occasion.

## References

1. Thorup, M.: Near-optimal fully-dynamic graph connectivity. In: Proc. 32nd ACM Symposium on Theory of Computing (STOC). (2000) 343–350
2. Miltersen, P.B., Subramanian, S., Vitter, J.S., Tamassia, R.: Complexity models for incremental computation. Theoretical Computer Science **130** (1994) 203–236 See also STACS'93.
3. Miltersen, P.B.: 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.
4. Pătraşcu, M., Demaine, E.D.: Logarithmic lower bounds in the cell-probe model. *arXiv:cs.DS/0502041*. Submitted journal version of two publications appearing in SODA'04 and STOC'04 (2004)
5. Fredman, M.L.: The complexity of maintaining an array and computing its partial sums. Journal of the ACM **29** (1982) 250–260
6. Fredman, M.L., Saks, M.E.: The cell probe complexity of dynamic data structures. In: Proc. 21st ACM Symposium on Theory of Computing (STOC). (1989) 345–354
7. Frandsen, G.S., Miltersen, P.B., Skyum, S.: Dynamic word problems. Journal of the ACM **44** (1997) 257–271 See also FOCS'93.
8. Alstrup, S., Husfeldt, T., Rauhe, T.: Marked ancestor problems. In: Proc. 39th IEEE Symposium on Foundations of Computer Science (FOCS). (1998) 534–543
9. Krohn, K., Rhodes, J.: Algebraic theory of machines I. prime decomposition theorem for finite semigroups and machines. Transactions of the American Mathematical Society **116** (1965) 450–464

## A Upper Bounds for Predecessor Problems

**Theorem 7.** *The greater-than problem, the colored  $k$ -predecessor problem, and the segment representative problem have bit-probe complexity  $O(\frac{\lg n}{\lg \lg n})$ .*

The upper bound for the greater-than problem is remarkably easy. Consider a balanced tree with branching factor  $B = \Theta(\frac{\lg n}{\lg \lg n})$ , and with  $n$  leaves. Every possible value of the update parameter  $a$  is represented by a root-to-leaf path. In the update stage, we mark this root-to-leaf path, taking time  $O(\lg_B n) = O(\frac{\lg n}{\lg \lg n})$ . In the query stage, we first find the point where  $b$ 's path in the trie would diverge from  $a$ 's path. This can be done by scanning  $b$ 's path upwards, until we find the first marked node; this step takes time  $O(\lg_B n)$ . Next, we examine all the children of the lowest common ancestor. We find the marked child (corresponding to  $a$ ), and determine whether it is to the left or to the right of the child corresponding to  $b$ , which solves our problem. Thus, the total time for the query is  $O(\lg_B n + B) = O(\frac{\lg n}{\lg \lg n})$ . Next, we extend this basic idea to the colored  $k$ -predecessor problem. The segment representatives problem has been discussed in Section 4.

### A.1 The Colored $k$ -Predecessor Problem

Again, we construct a balanced tree with branching factor  $B = \Theta(\frac{\lg n}{\lg \lg n})$  with  $[1, n]$  in the leaves. A node is called *active* if any leaf under it is in the current set. Active nodes are specially marked, and hold the colors of the largest  $k$  elements under them. If fewer than  $k$  elements are under the node, the exact number is kept. Remember that  $k = O(1)$  so all this requires  $O(1)$  bits. Among the children of a node, the minimum and maximum active nodes are specially marked.

Using this information, a query proceeds as follows. Start scanning upwards from the query position towards the root, until you find an active node. Then, we examine all left siblings of the node from which we ascended, in right-to-left order. If we collect  $k$  colors, we stop. Otherwise, we continue scanning towards the root. If we ascend from a node marked as the minimum among its siblings, we simply go on. Otherwise, we scan left siblings again, collecting more predecessors. We continue climbing upwards until we find  $k$  predecessors, or we reach the root. Climbing requires  $O(\lg_B n)$  time in total. In addition to this, we make at most  $k + 1$  scans of the siblings, taking time  $O(kB)$ . This is so because we scan the siblings the first time we reach an active node, and then only when we have not ascended from a node marked as minimum among its siblings. Thus, for each additional scan, we collect at least one additional predecessor. Therefore, the total running time is  $O(\lg_B n + kB) = O(\frac{\lg n}{\lg \lg n})$ .

We now implement insertions. First, traverse the leaf-to-root path looking for an active node. Along the way, label nodes active, both minimum and maximum among siblings, and create a color list with just the new element. When we find the first active node, we scan all its children, and remark the minimum and maximum, as these might have changed. From now on, we only have to update the color lists of the active node and its ancestors. We have the following cases:

- if a node is the only active child (it is marked as both the minimum and maximum), we just copy its color list to the parent, in  $O(1)$  time per level.
- if a node has less than  $k$  colors in its list, we recalculate the list of the parent by traversing all its children. This takes  $O(B)$  time, but we do it at most  $k$  times, since the color list of the current node grows each time.
- otherwise, we are only interested in right siblings (because left siblings cannot add to an already full list). If the node is marked as the maximum child, we just copy its list to the parent, in  $O(1)$  time.
- otherwise, we recalculate the list of the parent by traversing all children. Each time we do this, the position of the change shifts by at least one to the left, because our node was not maximum among siblings. So we can only do this  $k$  times before the inserted value is shifted out and it does not matter.

We stop whenever we have updated the color list of a node, and nothing changed. Because higher nodes only care about the color lists of their children, they cannot see any difference (this is so even if the actual points with the associated colors have changed). By the analysis done in each case, the running time is again  $O(\lg_B n + kB) = O(\frac{\lg n}{\lg \lg n})$ .

Finally, we discuss deletes. We go up the tree from the leaf. As long as the current node is both a minimum and a maximum, we mark the parent inactive. When this is no longer the case, the parent remains active. We recalculate the minimum and maximum children at this level. Then, we only have to recalculate the color lists of the active node and its ancestors. This is identical to the insertion case, so we obtain the same running time.

## B Data Structure for Group Free Monoids

Here we analyze cases 1 and 2 given by the Krohn-Rhodes decomposition. Case 3 is analyzed in the main body of our proof (Theorem 5).

*Case 1* –  $M = \{1_M, a, a^2, \dots, a^k = a^{k+1}\}$ . The solution is simple based on the  $k$ -predecessor structure. We maintain such a structure on the set  $\{i \mid A[i] \neq 1_M\}$  (this includes breakpoints). A query can always be answered based on the values of the  $k$  preceding non-identity elements. Specifically, we compose these elements from right to left until either we hit a  $\textcircled{D}$ , or we reach  $a^k$ .

*Case 2* –  $\forall a, b \in M \setminus \{1_M\}, ab = a$ . In this case, a partial sum is determined by the value of the first non-identity element following a breakpoint. Our data structure has the following components:

- a structure for finding segment representatives with the dynamic set  $B = \{i \mid A[i] = \textcircled{D}\}$ . Denote the elements  $B = \{b_1, b_2, \dots\}$ .
- a colored predecessor structure, on the set  $C = \{i \mid A[i] \neq 1_M\}$  (note that this includes breakpoints). The “color” of  $i$  is  $A[i]$ .

- an array  $D[1..n]$ ; for any segment  $(b_i, b_{i+1})$ , the value of  $D[\text{repr}(b_i + 1)]$  is equal to the value of the first non-identity element from the segment, or  $1_M$  if none exists. The values of  $D$  at positions which are not representatives are ignored.

We first implement a query to position  $i$ . First, find the color of the predecessor of  $i$  in  $C$ . If the predecessor is a breakpoint, return  $1_M$ . Otherwise, the answer is  $B[\text{repr}(i)]$ .

We implement an update to position  $i$  in two steps: first, we change  $A[i]$  to  $1_M$ , and then change it to the new value. Updating  $B$  and  $C$  is trivial. For updating  $D$ , we have the following cases:

- replacing  $\textcircled{b}$  by  $1_M$ . This merges the two segments around  $i$ . The new segment's representative value,  $B[\text{repr}(i)]$ , will be obtained either from the first segment, i.e.  $B[\text{repr}(i - 1)]$  before  $i$  was removed, or from the second segment ( $B[\text{repr}(i + 1)]$ ) if  $B[\text{repr}(i - 1)] = 1_M$  (so the first segment contained only  $1_M$ 's).
- replacing  $1_M$  by  $\textcircled{b}$ . This splits a segment in two. We obtain the representative of the right segment,  $B[\text{repr}(i + 1)]$ , by querying the successor of  $i$  in  $C$ . For the left segment, we query the predecessor of  $i$ . If it is a breakpoint,  $B[\text{repr}(i - 1)] = 1_M$ ; otherwise, it is the old representative of the unsplit segment.
- replacing an element of  $M \setminus \{1_M\}$  by  $1_M$ . First, we query the predecessor of  $i$ ; if it is not a breakpoint, we have nothing to do. Otherwise, we query the successor of  $i$ , and update  $B[\text{repr}(i)]$  accordingly.
- replacing  $1_M$  by an element of  $M \setminus \{1_M\}$ . First, we query the predecessor of  $i$  in  $C$ ; if it is not a breakpoint, we have nothing to do. Otherwise, we set  $B[\text{repr}(i)]$  to the new value  $A[i]$ .