



Basic Research in Computer Science

BRICS DS-97-3 T. Husfeldt: Dynamic Computation

Dynamic Computation

Thore Husfeldt

BRICS Dissertation Series

DS-97-3

ISSN 1396-7002

December 1997

Copyright © 1997,

**BRICS, Department of Computer Science
University of Aarhus. All rights reserved.**

**Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.**

**See back inner page for a list of recent BRICS Dissertation Series publi-
cations. Copies may be obtained by contacting:**

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK-8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

**BRICS publications are in general accessible through the World Wide
Web and anonymous FTP through these URLs:**

`http://www.brics.dk`

`ftp://ftp.brics.dk`

This document in subdirectory DS/97/3/

Dynamic Computation

Thore Husfeldt

Ph.D. Dissertation



Department of Computer Science
University of Aarhus
Denmark

DYNAMIC COMPUTATION

A Dissertation Presented to the
Faculty of Science of the University of Århus in
Partial Fulfilment of the Requirements for the Ph.D. Degree

Thore Husfeldt
Winter 1997

Abstract. This thesis is in Theory of Computation. We study quantitative aspects of computational problems that arise in settings where the input instance is subject to changes, i.e., *dynamic* problems. The results include efficient dynamic algorithms and data structures and strong information-theoretic lower bounds for problems on graphs, strings, and finite functions.

Chapter 1 contains a brief introduction and motivation of dynamic computations, and illustrates the main computational models used throughout the thesis, the random access machine and the *cell probe model* introduced by Fredman.

Chapter 2 paves the road to proving lower bounds for several dynamic problems. In particular, the chapter identifies a number of key problems which are hard for dynamic computations, and to which many other dynamic problems can be reduced. The main contribution of this chapter can be summarised in two results. The first shows that the signed prefix sum problem, which has already been heavily exploited for proving lower bounds on dynamic algorithms and data structures, remains hard even when we provide some amount of non-determinism to the query algorithms. The second result studies the amount of extra information that can be provided to the query algorithm without affecting the lower bound. Some applications of these results are contained in this chapter; in addition, they are heavily developed for the lower bound proofs in the remainder of the thesis.

Chapter 3 investigates the dynamic complexity of the symmetric Boolean functions, and provides upper and lower bounds. These results establish links between parallel complexity (namely, Boolean circuit complexity) and dynamic complexity. In particular, it is shown that the circuit depth of any symmetric function and the dynamic prefix problem for the same function depend on the same combinatorial properties. The connection between these two different modes and models of computation is shown to be very strong in that the trade-off between circuit size and circuit depth is similar to the trade-off between update and query time.

Chapter 4 considers dynamic graph problems. In particular, it presents the fastest known algorithm for dynamic reachability on planar acyclic digraphs with one source and one sink (known as planar *st-graphs*). Previous partial solutions to this problem were known. In the second part of the chapter, the techniques for lower bound from chapter 2 are further exploited to yield new hardness results for a number of graph problems, including reachability problems in planar graphs and grid graphs, dynamic upward planarity testing and monotone point location.

Chapter 5 turns to strings, and focuses on the problem of maintaining a string of parentheses, known as the dynamic membership problem for the Dyck languages. Parentheses are inserted and removed dynamically, while the algorithm has to check whether the string is properly balanced. It is shown that this problem can be solved in polylogarithmic time per operation. The lower bound techniques from the thesis are again used to prove the hardness of this problem.

Contents

Preface	9
1 Overview	9
2 Acknowledgements	9
1 Dynamic Computation	11
1 Introduction	11
2 Paradigms	12
3 Models	14
2 Hard Dynamic Problems	19
1 Signed Prefix Sum	19
2 Nondeterministic Queries	20
3 Refinement	28
4 Applications	31
3 Symmetric Functions	33
1 Symmetric Functions	33
2 Communication Complexity	34
3 Boolean Circuit Complexity	35
4 Cell Probe Complexity	37
4 Graphs	45
1 Algorithms	45
2 Lower Bounds	61
5 Strings	67
1 Dyck Languages	67
2 Algorithms	72
3 Lower Bounds	78
Notation	83
Bibliography	85

Preface

1 OVERVIEW

This overview identifies the main contributions in this thesis, including bibliographic references to material that has been already published and due attributions to my co-authors.

Chapter 1 presents a brief introduction to the concept of dynamic computation, including a description of the computational paradigms and models used in the sequel. The chapter contains no new results.

In Chap. 2 we prove new hardness results for the cell probe model of computation. These lower bounds provide a *Leitmotiv* that will recur in all remaining chapters and serves as a connection between our results. The results were obtained jointly with Theis Rauhe [35] and extend previously published material with Theis Rauhe and Søren Skyum [36].

The three remaining chapters cover various aspects of dynamic computation and study different combinatorial objects: finite functions, graphs, and strings. They all appeal to the lower bound hardness results in Chap. 2 but are independent of each other and their ordering is arbitrary.

In Chap. 3 we will study the hardness of symmetric functions and compare it known results in other models. Again, these results are from [35]. The presentation of Boolean circuit complexity includes a new proof that was found in collaboration with Gerth Stølting Brodal [12].

Graphs are the object of study in Chap. 4. It includes an efficient dynamic reachability algorithm for a small class of digraphs from [34] and a couple of lower bounds for graph problem that stem from [36].

Our third case study is strings, namely strings of properly balanced brackets, to which we turn in the last chapter. This consists mainly of joint work with Gudmund Skovbjerg Frandsen, Peter Bro Miltersen, Theis Rauhe, and Søren Skyum [21]; but the lower bounds are updated [35, 36].

2 ACKNOWLEDGEMENTS

The computer science department in Århus has provided a stimulating environment for writing this thesis. I thank Gerth Stølting Brodal, Gudmund Frandsen, Peter Bro Miltersen, Theis Rauhe, and Søren Skyum for collaborating with me. The latter two wrote their master's thesis [60] on the work in Chap. 5 and have contributed many of the results as well as to the presentation. Many other Århus people have provided insightful comments, including Lars Allan Arge and Peter Binderup.

My advisor Sven Skyum has put up with countless scientific (and personal) oddities from my side. It is to his credit that even though I always enjoyed total freedom to pursue my own interests he somehow succeeded in gently nudging me onto the Right Way.

I spent part of my time at the Hebrew University in Jerusalem, and benefitted endlessly from discussion with people there, including Noam Nisan, Jiří Sgall, and Avi Wigderson. My stay had profound influence on my taste in computer science (and my view of the world, for that matter).

I enjoyed financial support from the Danish SU-system, the Århus Faculty of Science, the Esprit Alcom-IT project (number 20244), and the Danish Research Academy. I acknowledge the fact that I have completed my studies in a privileged time and country and sincerely wish that more people would have the same opportunities.

Some parts of the text add only little to the presentation of the material but did not seem to deserve the hard sentence of omission. They often point out technical details that seem important to me but few other people or contain comparisons to the literature that are of little interest to the general audience. To distinguish these esoteric passages from the main body of the text, and to further discourage the reader, they are set in smaller type.

This text was typeset using Donald Knuth's wonderful \TeX system [43], whose qualities never cease to amaze me. The body font is 10/13 Computer Modern [42], Knuth's own adaption of Monotype 8A. Kristoffer Rose was only a staircase away while I typed this material, and many figures were therefore done in his \Xy-pic system. Those graphical and typographical atrocities that remain in the text are entirely due to my own lack of knowledge and taste in these matters.

This thesis is written in my third-best language, since probably more people will enjoy it in poor English than in good Danish or German. Even though native English speakers are a tolerant bunch I apologise for every *faux pas* in this work—like referring to myself in the preface with the colloquial 'I' only to change to the more scientific 'we' on the next page.

CHAPTER 1

Dynamic Computation

Weep not that the world changes—did it keep
a stable, changeless state, 'twere cause indeed to weep.

—William Cullen Bryant, *Mutation*

A theory has only the alternative of being right or wrong. A model has
a third possibility: it may be right, but irrelevant.

—Jagdish Mehra (ed.) *The Physicist's Conception of Nature*, 1973.

1 INTRODUCTION

Computation can be seen as the process of transforming input to output, problem to solution, question to answer. But this does not give the whole picture, for many computational problems arise in contexts where the input is changing, a setting that we call *dynamic* as opposed to static.

Even though the *theory* of dynamic computation does not yet enjoy the same respectability as, e.g., parallel computation, the *concept* of dynamic computation it has been around for a long time.

One of the major discoveries of efficient algorithms is the *dynamic data structure*, where information about changing data is maintained in an efficient way—to solve a *static* problem! So even when electronic computers were solely used to solve static problems, dynamic computation turned out to be a useful concept; algorithmic problems in dynamic settings were studied almost from the beginning. For example, the behaviour of trees under random deletions was one of the first topics of algorithmic analysis.

Dynamic computation *per se* became an object of study somewhat later. Bentley [9] was among the first to pose the question of which efficient algorithms can be used in dynamic settings and changed the focus from individual problems to classes of problems. In the 80s, *dynamic algorithms* became an established subject area. Today, a complexity theory of dynamic computation is slowly evolving [51].

There are other ways to tell this story: The architecture of *electronic computers* in the Old Days physically reflected static view of computation: batch

driven processes transformed input to output. In contrast, today the user interacts with the computer terminal and many computational tasks are inherently dynamic.

Two issues motivate the theoretical study of dynamic computation: From a practical point of view, we want to *solve problems faster* by finding efficient algorithms that recompute parts of the solution as the instance is subject to changes, rather than finding the entire solution from scratch. From a theoretical point of view, we can hope for more *insight* into the nature of the problem at hand and into computation itself.

2 PARADIGMS

There are a number of paradigms for dynamic computation. We will employ all of them in this thesis without being very orthodox about mixing them up. The following list makes no claims to completeness. It is, however, informed of several notions that can be found in the literature. The reader is referred to [51, 48, 58, 22] for more background.

Dynamic membership problems. Let L denote a language over the alphabet A . The *dynamic membership* problem for L is to answer queries of the form ‘is x in L ?’ for some instance $x \in A^*$ that is subject to changes. The most basic update operation is

change(i, a): replace x_i by $a \in A$.

For example, let x be a string of brackets like $((()())())$ and let L be the language of properly nested brackets. Then the dynamic membership problem captures a feature of modern editors that parse the structure of the file while the user edits it. We will study this problem in Chap. 5.

Recomputation of functions. Let f denote a function in variables x_1, \dots, x_n . To compute f is to find the value of $f(x_1, \dots, x_n)$. To *recompute* f after x_i is changed to x'_i is to find the value of

$$f(x_1, \dots, x_{i-1}, x'_i, x_{i+1}, \dots, x_n)$$

given $f(x_1, \dots, x_n)$ and possibly an additional data structure. For a very easy example, the recomputation problem for the or-function in n variables (which we will denote or_n) would be to maintain an instance $x \in \{0, 1\}^n$ under the following updates:

change(i, a): replace x_i by $\neg x_i$,

query: return $x_1 \vee \dots \vee x_n$

For functions like this, that map to $\{0, 1\}$, the dynamic membership problem for $f^{-1}(x)$ is just the recomputation problem.

Dynamic algorithms. We may say that an algorithm is dynamic if it repeatedly solves some algorithmic problem on instances that are closely related. Often, it is a dynamised variant of some well-studied static problem. For example, a dynamic graph algorithm for reachability can answer reachability queries about a graph between updates that insert and delete edges.

This notion coincides with previous notions on decision problems. If the input graph is given by its incidence matrix then we can view this as a dynamic membership problem for the language of yes-instances. Upper and lower bounds for dynamic graph algorithms are studied in Chap. 4.

Abstract data types. The abstract data type view is the most general one and subsumes the other notions of dynamic computation. All data structures can be viewed as concrete implementations of a set of abstract operations. For example, the union–split–find problem on interval is to maintain a set $S \in \{1, \dots, n\}$ under the following operations:

insert(i): remove i from S ,

split(i): insert i into S ,

find(i): return the largest j such that $j \in S$ and $j \leq i$ (otherwise return 0).

It is difficult (and rather pointless) to draw a precise line between this view and the previous, mainly because the line between algorithms and data structures is not very sharp.

2.1 Other Notions

There are other computational paradigms for considering non-static problems that are closely related to ours. For example, the database view by Immerman and Patnaik [58], to which we briefly return later.

Among those views that receive a lot of attention is that of ‘on-line algorithms.’ Superficially, this seems closely related to dynamic computation: There is an update operation that gradually changes the setting, and knowing the updates beforehand would reduce the problem to the static version. Also, computation is potentially infinite in both settings, the resource bounds are put on the operations per update or query.

But the dynamic and on-line setting differ in many ways. One is operational: the on-line concept of effective computation, ‘competitiveness,’ is defined relative to an optimal algorithm. Also, the problem studied in both worlds are quite different.

Precise characterisations of when a problem is on-line and when it is dynamic are bound to fail, but here is one: In on-line problems, the input instance is *revealed* gradually to the algorithm, while in dynamic problems, it *changes* gradually. Alternatively, we can view on-line computation as a special case of dynamic computation.

This work has nothing to say about on-line computations.

3 MODELS

3.1 Random Access Computers

Our model of computation is the random access machine, on which all our algorithms are supposed to run. The machine's memory consists of registers with cell size c ; for most of our upper bound we will focus on logarithmic cell size, $c = \log n$. The reason for this particular choice of cell size is a question of consensus in the field more than anything else. In this thesis, we will sometimes consider polylogarithmic cell size as well.

The cost function is *unit cost*, so any register operation takes constant time. This model is sometimes called the *random access computer*.

We will not go into the details of which instruction set is provided with our RAM, since none of our results use such details. In § 2.1 of Chap. 2 we somewhat esoterically extend our instruction set with *nondeterministic* choices and assignments, but only to prove some strong lower bounds for deterministic computation.

When we say *efficient algorithm* for a dynamic problem we mean a RAM algorithm that runs in polylogarithmic time per update. The algorithm is allowed 'reasonable' preprocessing before the interaction starts, e.g., within polynomial time or logarithmic space bounds.

3.2 The Cell Probe Model

Our lower bounds will be proved in the *cell probe* or *decision-assignment-tree* model. A *decision-assignment tree* is a tree with three types of nodes:

1. assignment nodes are labeled with a *register* r and a *value* $1 \leq v \leq 2^c$ and have one child, we will sometimes draw their labels as $v/M[r]$,
2. decision nodes are labeled with a *register* r and have up to 2^c ordered children, we will sometimes draw their labels as $M[r]$,
3. answer nodes are leaves that are labeled 0 or 1.

A *query tree* is a decision-assignment tree with answer nodes at the leaves.

A *memory* is a string of *registers* that can hold values between 0 and 2^c . The *computation* of a decision-assignment tree on memory M is defined in the obvious fashion: Start at the root. From a decision node with label r , proceed to the child identified by the contents of M 's register r . From an assignment node with labels r and v , write v into M 's register r and proceed to the unique child. The *value* returned by a query tree is the label of the answer leaf reached.

To implement an abstract data type with a forest of decision assignment trees we associate a tree with each operation. We have focused on queries whose outcome is 0 or 1, but this could easily be extended.

Obviously, the cell probe model is at least as strong as the random access machine, so lower bounds for the former hold on the latter.

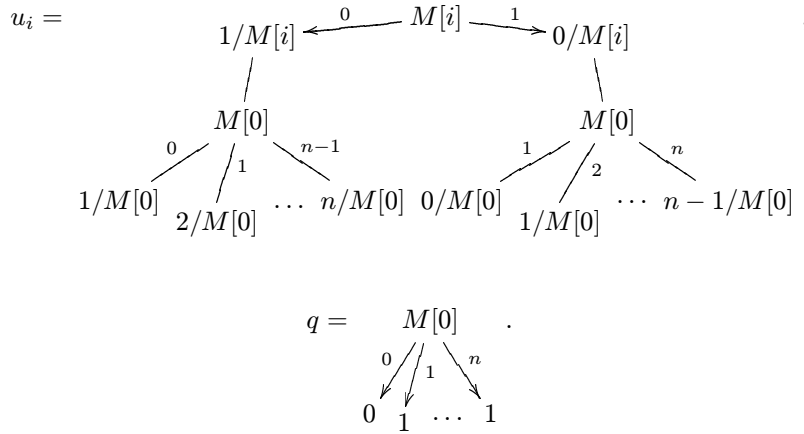


Fig. 1: A cell probe implementation for recomputing or_n . Register M_i stores x_i , register M_0 stores $x_1 + \dots + x_n$ using $\log n$ bits. There are n update trees u_1, \dots, u_n and a query tree q .

Time for some simple examples. Here is a forest of n trees that remembers the current instance in registers M_1, \dots, M_n :

$$u_1 = \begin{array}{c} M[1] \\ \swarrow 0 \quad \searrow 1 \\ 1/M[1] \quad 0/M[1] \end{array}, \quad \dots, \quad u_n = \begin{array}{c} M[n] \\ \swarrow 0 \quad \searrow 1 \\ 1/M[n] \quad 0/M[n] \end{array}, \quad (1)$$

where we let u_i denote the tree for the i th query $change(i)$, which is supposed to flip the i th bit of the instance.

Here is another, which implements the recomputation problem for the parity function in n variables:

$$u_1 = \dots = u_n = \begin{array}{c} M[1] \\ \swarrow 0 \quad \searrow 1 \\ 1/M[1] \quad 0/M[1] \end{array}, \quad q = \begin{array}{c} M[1] \\ \swarrow 0 \quad \searrow 1 \\ 0 \quad 1 \end{array}.$$

Note that this algorithm does not even remember the instance. It uses but a single register of memory and can do with cell size 1.

Figure 1 shows an implementation for recomputing or_n . It uses $M[0]$ to store a counter of the number of 1s in x . The figure also indicates that it makes sense to explain our algorithms in English rather than by exhibiting cell probe implementations.

Roots. The cell probe model was defined by Fredman [24] for constant cell size, but earlier Minsky and Papert [52, § 12.6] reason (about a static problem) within a model model that is essentially the same. Its most famous application

is a paper of Yao [72], to whom the model is often attributed, e.g., by Ajtai [2] in (yet) another remarkable paper.

3.3 Other Models of Computation

Structured models A lot of work has been done for lower bounds on dynamic problems in structured models like algebraic models, pointer machines, or comparison based algorithms. It is beyond the scope of this thesis to give an comprehensive overview of this rich field. The results are often edifying when it comes to understanding the hardness of problems relative to specific computational concepts and are valuable guides in the search for an efficient algorithm. However, as impossibility results *for modern electronic computers* their value dubious in light of schemes like hashing or recent algorithms that exploit the parallelism inherent in unit-cost register operations. In contrast, the information-theoretic lower bounds in the cell probe model speak with brutal finality.

Turing Machines Miltersen *et al.* use Turing machines to develop a framework for the study of dynamic computation (which they call incremental computation) with a structural flavour. The paper also contains results on cell probe computation and comments on the relationship between these two models. Miltersen [48] uses Turing machines to study space-bounded computation.

First Order Queries Immerman and Patnaik [58] define a concept of dynamic computation based on database theory. Roughly speaking, a problem is in the class Dyn-FO^+ if updates and queries are first-order computable. Little is known about the connection between cell probe computation and dynamic first order queries. One can prove, however, prove the following simulation result.

If we let $\text{CPROBE}(c, t)$ denote the class of languages that has dynamic membership cell probe implementations with cell size c and update and query time t Then for any $c, t \geq 1$ with $pc = O(\log n)$ we have

$$\text{CPROBE}(c, p) \subseteq \text{Dyn-FO}^+.$$

The inclusion is strict for $t \in o(\log n / \log \log n)$.

Here is why. Let L be a language in $\text{CPROBE}(c, t)$. Let T_1, \dots, T_{2n} denote the decision-assignment trees corresponding to the operations $\mathbf{change}(i, a)$ for $1 \leq i \leq n$ and $a \in \{0, 1\}$. From the constraints on c and t we infer

$$s = \max_{i \leq 2n} |T_i| = n^{O(1)}.$$

We will let M denote the trees' common memory (for simplicity, we can assume that $M(0)$ is zero iff the input instance is in L). It is not very hard to see that it is safe to assume that in any T_i , any memory location is queried at most once and on any path through the tree, and that there is at most one assignment to that location (which happens after any potential query).

We will model the family of decision-assignment trees in a structure over the signature σ . The universe U is of polynomial size, there are two relations in σ : the relation $\text{cont} : [|M|] \rightarrow [2^c]$ that is intended to reflect the contents of M , and the relation

$$\text{instr} : [2^{bt}] \times [n] \times \{0, 1\} \rightarrow \{\text{'dec'}, \text{'ass'}\} \times [|M|] \times [2^c] \times [2^{ct}],$$

that shall reflect the construction of the decision-assignment trees in that

$$\text{instr}(c, i, a) = (\text{type}, m, v, c')$$

means that the c th node of the tree for $\text{change}(i, a)$ is (depending on type) a decision node labeled by m whose v th son is node c' or an assignment of v to m whose sole descendant is c' . Note that both relations are of polynomial size.

With a polynomial amount of precomputation, we initialise the structure S_0 as such that the instr -relation is set up according to the T_i and the cont -relation is initialised according to the initial state of the memory (this initialisation is usually for free in the cell probe model, or may be polytime or logspace bounded).

We turn to the operations. It suffices to show that we can emulate the effect of T_i on M . To this end, we guess a computation path

$$c_1, \dots, c_t$$

using an existential quantification over $n^{O(1)}$ values. For each j and $j+1$ we now must check that the path adheres to the structure of T_i using instr .

Using a temporary relation cont' , we then note all memory updates and finally copy them into cont' . Note that because of our simplifying assumptions on T_i 's access to M , there is no need for a step-by-step simulation.

For the separation, [FMS93] shows that the word problem for a non-commutative group requires time $\Omega(\log n / \log \log n)$ even with logarithmic cell size. On the other hand, the problem is clearly first-order (see [IP94]).

This result has been independently obtained by Immerman and Miltersen (personal communication, unpublished).

CHAPTER 2

Hard Dynamic Problems

Yet we will ask;
That, if you fail in our request, the blame
May hang upon your hardness:

—William Shakespeare, *Coriolanus*

This chapter identifies some key problems that are hard for dynamic computation. These results will be used to prove lower bounds for the problems considered in the remaining chapters.

1 SIGNED PREFIX SUM

The *signed prefix sum* problem is to maintain a string $x \in \{-1, 0, +1\}^n$, initially $x = 0^n$, under updates that change the letters of x and queries that ask for its prefix sums:

update(i, a): change x_i to $a \in \{-1, 0, +1\}$,

sum(i): return $\sum_{j=1}^i x_j$.

An efficient data structure for this problem is immediate: Store x_1, \dots, x_n at the leaves of a balanced binary tree whose internal nodes store the sum of their children's values. Updates and queries can be performed in logarithmic time.

It is known that the lower bound for this problem is $\Omega(\log n / \log(c \log n))$ per operation. It applies even to the problem of finding the least significant bit of the prefix sum, namely the following query:

parity(i): return $\sum_{j=1}^i x_j \bmod 2$.

The bound is given in [25].

Optimal implementations. It is not hard to meet the lower bound in the cell probe model. Consider a balanced tree with n leaves and fan-out c ; the height of this tree is at most $\log n / \log c$. In each internal node we store parity of the bit sum of each of its c subtrees; this takes up c bits and hence fits into a single register. In the cell probe model, *any* register operation can be performed in unit time, so we are done.

This is a toy example of how to cram information into single registers, thereby exploiting the parallelism provided by unit-cost register operations. To make this work for realistic machines like RAMs, is much harder but not conceptually different. Dietz [16] has shown how to implement the above algorithm on a RAM to get the same, optimal, bounds with $c = \log n$. Interestingly, his work was spawned by the lower bound in [25], so here an upper bound on a realistic model was preceded by the matching lower bound in an unrealistic model.

Roadmap. The present chapter provides two new lower bounds for signed prefix sum-like problems. Theorem 1 studies the behaviour of query algorithms with nondeterminism, while the Refinement Lemma studies how much additional information can be disclosed to the query algorithms. Both results will come in a ‘balanced version’ that will prove very useful for constructing new lower bounds. Theorem 1 and the Refinement lemma are conceptually independent, but proving the latter requires some more work. So, to maintain a leisurely pace, we turn to nondeterminism first.

2 NONDETERMINISTIC QUERIES

In this section we give the query programs access to unbounded nondeterminism and show that the signed prefix sum problem is still hard. This in itself is not an interesting statement, since nondeterminism is not a realistic model of computation. Instead, the motivation for this result is its use for proving lower bounds for dynamic algorithms—in realistic models. We will see this several times in the remaining chapters. Also, the result allows us to make some points about the power of the time stamp method that would be less sharp otherwise, see § 4.2.

2.1 Nondeterminism

Nondeterministic Random Access Machines. We allow the query algorithm to nondeterministically load a new value into a memory cell. We can view this as extending the instruction set with *nondeterministic assignments* of the form

$$i \leftarrow S, \tag{1}$$

where the variable i (which is stored in a single memory cell) receives some new value from the finite set S . (There is no reason to be formal about how the set S may be specified, since we will be reasoning within the cell probe model in a minute, where formal definitions will be given.)

The *value computed* by a nondeterministic program is 1 if and only if there is a computation that returns 1. Here is an algorithm for prefix-or_n that makes use of nondeterminism to achieve constant time per operation:

$update(i):$ $M[i] \leftarrow \neg M[i]$	$query(i):$ $j \leftarrow \{1, 2, \dots, i\}$ return $M[j]$
---	--

We stress that like in the example above, nondeterminism is only allowed in the query operations.

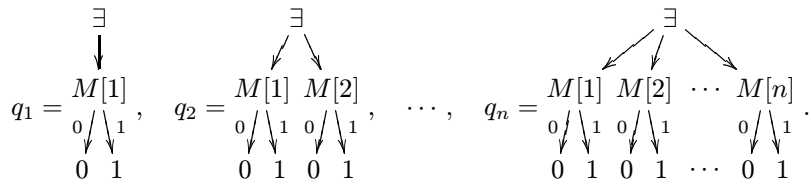
Nondeterministic Decision–Assignment Trees. To model this mode of computation in the cell probe model we introduce a new type of nodes to decision–assignment trees: a *nondeterministic node* is labelled ‘ \exists ’ and has up to 2^c successors (recall that c is the cell size of the memory). Computation is defined as follows. A *computation path* is a path in the tree from root to one of the leaves whose behaviour at decision and assignment nodes agrees with the previous definition. The *value computed* by a nondeterministic decision–assignment tree q on memory M is 1 if and only if there exists a computation path from the root to a leaf with label 1. In this definition we are guided by use the *dynamic membership view* of dynamic computation.

We have not defined which assignments to memory are performed by a nondeterministic computation when there is a choice.

Note that our computational resource is still the *depth* of a nondeterministic query tree, so the cost incurred by a nondeterministic choice is 1. This is a departure from our usual philosophy of charging only the number of probes into memory.

Obviously, nondeterministic decision–assignment trees are at least as strong as random access machines with unbounded nondeterminism. The cost of a nondeterministic assignment as in (1) is 2, since the \exists -node and the subsequent assignment node used to model this instruction each increase the tree depth by 1.

As an example we present n nondeterministic decision trees corresponding to the nondeterministic RAM-queries for prefix-or_n that we just saw. The update operations are the same as (1) on p. 15; recall that M_1, \dots, M_n contain the current instance.



So prefix-or_n has nondeterministic cell probe complexity 2 on cell size $\log n$.

Roots. Nondeterminism in decision trees has been studied by Manber and Tompa [46]. Nondeterminism in random access machines was introduced by Monien [53].

The literature does not agree on how to define this notion. *Bounded* nondeterminism uses nondeterministic branching in the form of a programming construct like **goto** l_1 **or** l_2 . *Unbounded* nondeterminism uses nondeterministic assignments to allow a register to receive a new value, for example by transferring the contents of an arbitrary other register as in ‘ $M[i] \leftarrow \bigcup_{n \geq 1} M[n]$ ’, or choosing an arbitrary integer ‘**guess** $M[i] \in N$.’ Additionally, models differ in their choice of cost function and register size.

For comparison, our lower bound applies to unbounded nondeterminism where the assignment operation $M[i] \leftarrow S$ loads $M[i]$ with an arbitrary value from $S \subseteq \{1, \dots, 2^c\}$; the set can be any function on n of the information gathered so far. This subsumes all the above notions with the same cell size.

2.2 Lower Bound for Prefix Sum

Theorem 1 *Consider any nondeterministic implementation for cell size c of the signed prefix sum problem with update time t_u . Then the query time t_q must satisfy*

$$t_q = \Omega\left(\frac{\log n}{\log(t_u c \log n)}\right). \quad (2)$$

The rest of this section is devoted to a proof of this result, which uses the *time stamp* technique of [25]. We will study a sequence of updates u_1, u_2, \dots , followed by a single query q_i , and argue that on the average (over choices of update sequences and query index), this query must read many memory cells.

The proof serves as the base for the more elaborate proofs of the balanced version of Thm. 1 and the Refinement lemma below.

Model and Notation. To each update we associate a *decision-assignment tree*. To each query $query(i)$ we associate a nondeterministic query tree q_i , with leaves labeled 0 and 1 to represent the possible answers. Let $qM \in \{0, 1\}$ denote the result of evaluating the decision tree q on memory M .

We will sometimes view (q_1M, \dots, q_nM) as a vector, the *query vector*, in $\{0, 1\}^n$ equipped with the Hamming distance

Updates and epochs. We will encode updates by binary strings $u \in \{0, 1\}^*$ in a way explained below. Write

$$d = \left\lceil \frac{\log n}{\log(t_u c \log n)} \right\rceil. \quad (3)$$

We split each string into into d substrings called *epochs*, whose length is given by

$$e(t) + \dots + e(1) = \left\lfloor \frac{n^{t/d}}{d} \right\rfloor. \quad (4)$$

Time flows backwards, so epoch 1 is the last substring of u , and in general the update string is an element in

$$U = U_d U_{d-1} \dots U_1, \quad \text{where } U_t = \{0, 1\}^{e(t)}.$$

The length of the entire update string is

$$|u| = e(d) + \dots + e(1) \leq \left\lfloor \frac{n}{d} \right\rfloor;$$

to alleviate notation we assume that n/d is an integer in the following. We will use abbreviations like $U_{>t}$ for $U_d \dots U_{t+1}$, the updates prior to epoch t , and $U_{\leq t}$ for $U_t \dots U_1$, the updates in epoch t and and later.

The value in (4) is chosen so that the epoch length decreases very fast, namely

$$|U_{<t}|^{O(t_{nc} \log n)} = |U_t|^{o(1)}, \quad (5)$$

where we have used (3).

We now define the updates. In general, the i th entry of update string u changes a letter in x to u_i . So, if u_i is 0 then nothing happens, if u_i is 1 then $\text{update}(\cdot, +1)$ is performed at some position specified below, while $\text{update}(\cdot, -1)$ is never performed in this update scheme.

The position of the affected letter is defined as follows: Consider the updates in epoch t and index them as $u_1 \cdots u_{e(t)} \in U_t$. Write x as a table of dimension $d \times n/d$ like this:

$$\begin{bmatrix} x_1 & x_{d+1} & & x_{n-d+1} \\ x_2 & x_{d+2} & & x_{n-d+2} \\ \vdots & \vdots & \dots & \vdots \\ x_d & x_{2d} & & x_n \end{bmatrix} ;$$

The i th update in epoch t affects the letter in row t and the column given by

$$(i-1) \cdot \left\lfloor \frac{n/d}{e(t)} \right\rfloor + 1, \quad (6)$$

so the distance between two letters affected in the same epoch is least

$$\left\lfloor \frac{n/d}{e(t)} \right\rfloor. \quad (7)$$

Also, no two epochs change the same letter. Fig. 2 shows a small example.

The next lemma is from [25].

Lemma 1 *Consider the query vectors that result from update strings that differ only in epoch t . For large n , at most $|U_t|^{\frac{9}{10}}$ are in the same Hamming ball of radius $\frac{1}{16}n$.*

Proof. Choose any such Hamming ball and pick an update string that results in a query vector in that ball. Let $u \in U_t$ denote epoch t of this string.

We will bound the number of $v \in U_t$ whose query vectors end up in the same ball. Let $w \in U_t$ record the difference between u and v , i.e., the i th letter of w is 1 if and only if u and v differ on the i th letter. Now let w' denote the string of prefix sum parities of w , i.e.

$$w'_i = w_1 + \cdots + w_i \pmod{2}, \quad 1 \leq i \leq e(t).$$

It is easy to see that w' records the difference between the query vectors resulting from u and v . Indeed, each 1 in w' yields an interval of indices where the vectors differ, and the length of this interval is d times the distance given by (7). In other words, each 1 in w' contributes as many points to the Hamming

Index i	1	2	5	10	15	20	25	30																							
Epoch affecting i	·	·	1	2	3	·	·	3	·	·	3	·	·	·	·	3	·	·	3												
Update affecting i																															
in epoch 3:	·	·	·	·	u_1	·	·	u_2	·	·	u_3	·	·	·	·	u_4	·	·	u_5	·	·	·	·	u_6	·	·	u_7				
in epoch 2:	·	·	·	·	u_8	·	·	·	·	·	·	·	·	·	·	·	u_9	·	·	·	·	·	·	·	·	·	·	·			
in epoch 1:	·	·	u_{10}	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·			
Examples of update strings:																															
$u = 0000000000$																															
			1	2	3		3		3					3	2	3			3		3										
resulting instance x	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
$x_1 + \dots + x_i$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
result of $query(i)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
$u = 1111111111$																															
			1	2	3		3		3					3	2	3			3		3										
resulting instance x	0	0	0	1	1	1	0	0	1	0	0	1	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	1	0	0	1
$x_1 + \dots + x_i$	0	0	0	1	2	3	3	3	4	4	4	5	5	5	5	5	5	6	6	7	8	8	8	8	8	8	9	9	9	10	
result of $query(i)$	0	0	0	1	0	1	1	1	0	0	0	1	1	1	1	1	1	0	0	1	0	0	0	0	0	0	1	1	1	0	
$u = 0101100111$																															
			1	2	3		3		3					3	2	3			3		3										
resulting instance x	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	
$x_1 + \dots + x_i$	0	0	0	1	2	2	2	2	3	3	3	3	3	3	3	3	3	4	4	5	6	6	6	6	6	6	6	6	6	6	
result of $query(i)$	0	0	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	

Fig. 2: Update scheme used in the proof of Thm 1.

distance between the resulting query vectors. So if we let $|w'|_1$ denote the number of 1s in w' , the Hamming distance between two query vectors is at least

$$|w'|_1 \cdot d \cdot \left\lfloor \frac{n/d}{e(t)} \right\rfloor \geq \frac{1}{2} |w'|_1 \cdot \frac{n}{e(t)}, \quad (8)$$

where we have used that $\lfloor a \rfloor \geq \frac{1}{2}a$ for $a \geq 1$.

By the triangle inequality, the maximum Hamming distance between two query vectors in the same ball is $\frac{1}{8}n$. This bounds the number of 1s in w' to $\frac{1}{4}e(t)$ for large n . Hence the number of choices for w' is given by

$$\sum_{i=0}^{\frac{1}{4}e(t)} \binom{e(t)}{i} < 2^{\frac{9}{10}e(t)}.$$

This also bounds the number choices of $v \in U_t$, since there is a 1-to-1 correspondence between v and w' . \square

Memories and time stamps. Let M^u denote the memory after updates u . We will imagine that whenever a register is written, it receives a *time stamp*, i.e., the name of the current epoch, that overwrites any previous time stamps. Thus the time stamp of a register is the name of the last epoch in which it was written. A computation *encounters* a time stamp if it reads a register with that time stamp.

For index i and update string u let $T(i, u)$ denote the set of ‘unavoidable’ time stamps, i.e., those encountered on every accepting computation path of q_i on M^u . If there are no accepting computations, the set is empty.

This deserves a more formal definition. Let w denote a witness for a computation path of q_i on M^u , and let for a moment $A(i, u)$ denote the set of witnesses that lead to accepting computations of q_i on M^u . Let for a moment $T(i, u, w)$ denote the set of time stamps encountered by the computation of q_i on M^u that is witnessed by w . Then

$$T(i, u) = \bigcap_{w \in A(i, u)} T(i, u, w), \quad \text{if } A(i, u) \neq \emptyset,$$

and $T(i, u) = \emptyset$ otherwise.

Claim 1 *If M^u and M^v differ only on registers with time stamp t , and t is in neither $T(i, u)$ nor $T(i, v)$, then $q_i M^u = q_i M^v$.*

Proof. If neither $q_i M^u$ or $q_i M^v$ accepts then there is nothing to prove. Assume without loss of generality that q_i has an accepting computation on M^u . Since t is not in $T(i, u)$, there must be an accepting computation that avoids registers with time stamp t . However, this computation might as well be executed on M^v , by the premise. Hence q_i has an accepting computation on M^v as well. \square

The time stamp method. The next lemma is the crux of the time stamp method. It shows for every epoch that for the majority of remaining updates, the registers written in this epoch are read by many queries. In other words, not all information about the present epoch can be propagated through updates in the remaining epochs. The fact that the epoch length decreases exponentially is of course crucial to this argument.

Lemma 2 *Fix any epoch $1 \leq t \leq d$ and past and future updates $x \in U_{<t}$, $y \in U_{>t}$. For large n , at least half of the update strings $u \in xU_t y$ satisfy*

$$|\{1 \leq i \leq n \mid t \in T(i, u)\}| \geq \frac{1}{16}n,$$

for $t_q = O(\log n)$.

Proof. Consider the set $V \subseteq xU_t y$ of updates after which fewer than $\frac{1}{4}n$ queries encounter time stamp t , i.e. xuy for $u \in U_t$ is in V

$$|\{1 \leq i \leq n \mid t \in T(i, xuy)\}| < \frac{1}{16}n.$$

We will bound the size of V below $\frac{1}{2}|U_t|$.

To this end partition V into equivalence classes such that u and v are in the same class only if M^u and M^v disagree only on registers with time stamp t .

Let us bound the number of classes. Note that at most $n2^{t_q c}$ registers appear in the entire forest of query trees. The number of updates in the last $t-1$ epochs is at most

$$r = t_u \cdot (e(t-1) + \dots + e(1)),$$

so the number of different memories that can result from this is

$$\sum_{i=0}^r \binom{n2^{t_q c}}{i} 2^{ic} \leq |U_{<t}|^{O(t_u c \log n)}, \quad (9)$$

where we have used the bound on t_q . We conclude using (5) that the number of classes is $|U_t|^{o(1)}$.

It remains to bound the size of each class. Consider two query vectors $q^u M^u$ and $q^v M^v$ for u and v in the same equivalence class. Then

$$|q^u M^u - q^v M^v| \leq \frac{1}{8}n, \quad (10)$$

because $\frac{15}{16}n$ entries of each vector depend only on registers with other time stamps than t . On these registers, the memories are indistinguishable and therefore yield the same result by Claim 1.

By (10), all vectors from the same class end up in a Hamming ball of radius $\frac{1}{16}n$, so Lemma 1 tells us that there can be only $|U_t|^{\frac{9}{10}}$ of them.

We conclude that the size of V is bounded by

$$|U_t|^{\frac{9}{10}} \cdot |U_t|^{o(1)}$$

which is less than $\frac{1}{2}|U_t|$ for large n . \square

We can now prove Thm. 1.

Proof of Theorem. Assume $t_q = O(\log n)$ so that the premise of Lemma 2 holds—else there is nothing to prove. The worst-case query time is larger than the average of $|T(i, u)|$ over choices of $i \in \{1, \dots, n\}$ and $u \in U$, so

$$|U|nt_u \geq \sum_{u \in U} \sum_{i=1}^n |T(i, u)| \quad (11)$$

$$= \sum_{t=1}^d \sum_{u \in U_{>t}} \sum_{w \in U_{<t}} \sum_{v \in U_t} \sum_{i=1}^n (t \in T(i, uvw)) \quad (12)$$

Lemma 2 tells us how many $v \in U_t$ fail to make the last sum exceed $\frac{1}{16}n$, so we can write

$$|U|nt_u \geq \sum_{t=1}^d |U_{>t}| \cdot |U_{<t}| \cdot \frac{1}{16}n \cdot \frac{1}{2}|U_t|,$$

which yields

$$|U|t_u \geq \frac{1}{32} \sum_{t=1}^d |U_{>t}| \cdot |U_{\leq t}| = \frac{1}{32}d|U|,$$

from which the bound follows. \square

2.3 Prefix Balancing

Consider an algorithm for signed prefix sum that makes the strong assumption that at all times during the operations, every prefix sum of the instance is bounded by $\log n / \log \log n$. Then this algorithm can do no better.

Theorem 1 (Balanced version) *Consider any nondeterministic implementation for cell size c of the signed prefix sum problem with update time t_u . Then for any integer function d with*

$$d = O\left(\frac{\log n}{\log t_u c \log n}\right),$$

the query time t_q must satisfy

$$t_q = \Omega(d),$$

even if the algorithm requires

$$\sum_{j=1}^i x_j \leq d \quad (13)$$

to hold for all $1 \leq i \leq n$ at all times.

$u = 111111111111:$	5	10	15	20	25	30
$x:$	0 0 0 + + + 0 0 - 0 0 + 0 0 0 0 0 - 0 - + 0 0 0 0 0 - 0 0 +					
$\Sigma_i x:$	0 0 0 1 2 3 3 3 2 2 2 3 3 3 3 3 3 2 2 1 2 2 2 2 2 2 1 1 1 2					
$q_i:$	0 0 0 1 0 1 1 1 0 0 0 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0					
$u = 01011001111:$	5	10	15	20	25	30
x	0 0 0 + + 0 0 0 + 0 0 0 0 0 0 0 0 - 0 - + 0 0 0 0 0 0 0 0 0 0					
$\Sigma_i x$	0 0 0 1 2 2 2 2 3 3 3 3 3 3 3 3 3 2 2 1 2 2 2 2 2 2 2 2 2 2					
q_i	0 0 0 1 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0					

Fig. 3: Balanced update scheme used in the proof of Thm. 1; compare this with page 24. We have written + for +1 and - for -1.

Especially, for maximal d and cell size $\log n$, no algorithm can do better than $O(\log n / \log \log n)$ per operation, which is the same bound as in the original formulation.

Proof. The parameter d plays the role of the value defined in (3). Its definition was only used in (5), which also holds under the present definition.

We construct a slightly different update scheme than above. As before, the meaning of an update string is defined epoch-wise. Consider the updates in epoch $u = u_1 \cdots u_{e(t)} \in U_t$. The i th update performs $update(j, a)$, where the update position j is given as in (6) on p. 23. The new value a is given by

$$(-1)^r, \quad \text{where } r = 1 + u_1 + \cdots + u_i \pmod{2}, \quad (14)$$

i.e., such that the nonzero updates in u alternate between -1 and $+1$, starting with $+1$.

It can be checked that the entire proof works *ad verbatim* with this new update scheme, simply because $+1 = -1 \pmod{2}$.

However, with the new scheme, the bound (13) holds at all times. For let x^t denote the string resulting from only the updates in epoch t ; this is well-defined because no two epochs write in the same positions by (6). Then we can write the instance x as $x^1 + \cdots + x^d$, where the sum is taken coordinate-wise. But by construction, every prefix of every x^t sums to at most 1, which implies the bound. \square

Note that the other main result of our paper [36] can be easily obtained by letting the alternating values in (14) start with either $+1$ or -1 , depending on the outcome of a fair coin flip at the start of every epoch. By standard probability theory, this implies that at all times during the updates, the *expected* absolute value of $\sum_{j=1}^i x_j$ is $\Theta(\sqrt{d})$. But we no longer have an application for this result.

3 REFINEMENT

We leave nondeterminism and turn to another variant of the signed prefix sum.

3.1 Signed Prefix Sum Refinement

We will study the performance of a query algorithm that receives a value s that is guaranteed to be *close to* (but not known to be equal to) the right sum. For example, the result § 2.3 implies that if s is guaranteed to lie within $\log n / \log \log n$ of $\sum_{j=1}^i x_j$, the problem retains its strong lower bound, even with nondeterminism. This section goes to show that even if s is known to be at most 1 off the right value, the query algorithm might as well ignore it.

More precisely, we will investigate the complexity of maintaining a string $x \in \{-1, 0, +1\}^n$, initially 0^n , under the following operations:

update(i, a): change x_i to $a \in \{-1, 0, +1\}$,

refine(i, s): return $\sum_{j=1}^i x_j \bmod 2$, provided that $|s - \sum_{j=1}^i x_j| \leq 1$ (otherwise the behaviour of the query algorithm is undefined).

We immediately state the result of this section in its balanced form; this will not make our proof any more complicated.

Refinement Lemma *Consider any implementation for cell size c of the signed prefix sum refinement problem with update time t_u . Then for any function d with*

$$d = O\left(\frac{\log n}{\log(t_u c \log n)}\right),$$

the query time t_q must satisfy

$$t_q = \Omega(d).$$

Moreover, this is true even if the algorithm requires

$$\sum_{j=1}^i x_j \leq d$$

to hold for all $1 \leq i \leq n$ at all times.

The proof can be seen as an extension of that for Thm. 1 in that many constructions are the same.

Query trees. To each query *refine*(i, s) we associate a query tree q_i^s , with leaves labeled 0 and 1 to represent the possible answers. Correctness will be given by the condition that

$$q_i^s M = \sum_{j=1}^i x_j \bmod 2, \quad \text{if } |s - \sum_{j=1}^i x_j| \leq 1.$$

For update string u we will write q_i^u for the query tree q_i^s corresponding to the ‘right guess’ $s = x_1 + \dots + x_i$, where x the instance resulting from updates u . The *query vector* is $(q_1^u M, \dots, q_n^u M)$, i.e., the responses yielded by guessing right every time.

Updates and epochs. Surprisingly, the update scheme for the proof of the Refinement lemma is exactly the same as that for Thm. 1, so the update position is given by (6) and the new value by (14). This also means that all prefixes balance, as claimed in the second part of the Refinement lemma.

However, the update scheme implies some properties of updates that differ only in a single epoch that we have not used yet. These are abstracted in the next claim.

Claim 2 *Let u and $v \in U$ be update strings that differ only in epoch t and let x and y denote the resulting instances. Then*

$$\left| \sum_{j=1}^i x_j - \sum_{j=1}^i y_j \right| \leq 1,$$

for all $1 \leq i \leq n$.

Proof. As before, let x^t denote the string resulting from only the updates in epoch t and write x as $x^1 + \dots + x^d$. If two update strings differ only in epoch t , the corresponding sums $x = x^1 + \dots + x^d$ and $y = y^1 + \dots + y^d$ differ only on the t th term, the prefix sum of which can be at most 1 by construction. \square

Memories and time stamps. For index i and update string u let $T(i, u)$ denote the set of time stamps encountered by q_i^u on M^u .

Claim 3 *Assume that for update strings u and v that differ only in epoch t , the resulting memories M^u and M^v differ only on registers with time stamp t . Then*

$$q_i^u M^u = q_i^v M^v$$

for all $1 \leq i \leq n$.

Proof. Let x and y denote the instances resulting from u and v , respectively. Choose $1 \leq i \leq n$. Let s denote $\sum_{j=1}^i x_j$. By Claim 2 and without loss of generality, $\sum_{j=1}^i y_j = s + 1$. By correctness,

$$q_i^s M^u = q_i^{s+1} M^u.$$

By assumption, we can replace M^u by M^v without changing the result. \square

We can now prove the Refinement lemma.

Proof of Refinement lemma. The proof of Thm. 1 can be reused *ad verbatim* with the new definitions. The only twist is in the proof of Lem. 2: In the present setting, there are $n(2n+1)$ query trees, so the left hand side of (9) should read

$$\sum_{i=0}^r \binom{n(2n+1)2^{t_{ac}}}{i} 2^{ib}$$

However, this does not affect the calculation. \square

4 APPLICATIONS

4.1 More Lower Bounds

As a warm-up, we now use Thm. 1 and the Refinement lemma to prove a lower bound for another problem—yet another toy problem, but an edifying one. And, as we will see later, a very useful one. The *balanced prefix sum vanishing* problem is to maintain a string $x \in \{-1, 0, +1\}^n$, initially 0^n , under updates that change the letters of x and queries that ask if a prefix sum of x vanishes:

update(i, a): change x_i to $a \in \{-1, 0, +1\}$ provided that $|x_1 + \cdots + x_j| \leq \lceil \log n / \log \log n \rceil$ holds for all j in the new instance,

vanish(i): return ‘yes’ if and only if $x_1 + \cdots + x_i$ equals 0.

Compared to signed prefix sum, this problem only has to keep track of instances that stay within a narrow interval and only has to check whether the prefixes are nonzero.

We show that this problem is just as difficult as signed prefix sum. We will give two proofs of this result, using both results of this chapter. The result is entailed by a much more general theorem in the next chapter.

Proposition 1 *Every implementation with cell size $\log n$ of the balanced prefix sum vanishing problem that uses polylogarithmic time for the updates must spend time $\Omega(\log n / \log \log n)$ per query.*

We focus on logarithmic cell size for no other reasons than ease of presentation. One can prove a more elaborate bound in terms of cell size and as a trade-off between update and query time (as in Thm 1) with little extra work.

Proof using Thm. 1. Let d be $\lceil \log n / \log \log n \rceil$ and consider an instance $x \in \{-1, 0, +1\}^n$ to the balanced signed prefix sum problem. Define $d + 1$ strings $y^{(t)}$ where

$$y^{(t)} = (-1)^t 0^{d-t} x, \quad 0 \leq t \leq d.$$

Let $t_u = t_u(n)$ denote the update time of our algorithm. Whenever x is changed, we can update the strings $y^{(t)}$ in time $(d+1) \cdot t_u(n+d)$, which is polylogarithmic if t_u is.

Index the strings $y^{(t)}$ from $-d$ to n so that the indices of their last part agree with x . We then have

$$\sum_{j=-d}^i y_j^{(t)} = -t + \sum_{j=1}^i x_j, \quad 0 \leq t \leq d, \quad 1 \leq i \leq n. \quad (1)$$

Here is a nondeterministic query that finds the i th prefix sum of x : Guess s from $\{0, \dots, d\}$, we know by the balancing condition that the sum is in that set. If indeed $\sum_{j=1}^i x_j$ equals s then by the above equation, $\sum_{j=-d}^i y_j^{(s)}$ vanishes. This we can verify with a single *vanish* query. The bound follows from Thm. 1.

□

Proof using the Refinement lemma. The construction of the strings $y^{(t)}$ is as above. To answer a query $refine(i, s)$ we check

$$\sum_{j=-d}^i y_j^{(s-1)}, \quad \sum_{j=-d}^i y_j^{(s)}, \quad \text{and} \quad \sum_{j=-d}^i y_j^{(s+1)}.$$

By (1), exactly one of these sums vanish because s is at most 1 off the correct sum. The bound follows from the Refinement lemma. \square

The first of these two proofs seems slightly easier, probably because nondeterminism is a familiar (if unrealistic) mode of computation. Often, both results can be applied and the choice is a matter of taste; an exception is Thm. 2 from the next chapter, where we cannot do without precision of the Refinement lemma.

4.2 On the Power of the Time Stamp Method

One of the messages of this chapter is that the time stamp method does not seem to be able to distinguish between nondeterministic and deterministic computation. This is certainly true for lower bound proofs that use reductions to prefix parity, since we have seen that this is a hard problem for nondeterministic computation. But also in a broader sense, the entire technique hinges on the hardness of propagating sufficient information to later epochs, and this information has to be present even for verifying a nondeterministic guess. Hence we can say that time stamp lower bounds can never be better than the best nondeterministic algorithm. We feel that this is valuable intuition about an important technique.

For example, consider the union–split–find problem on intervals from p. 2. Here is a nondeterministic algorithm: Maintain the interval boundaries in a doubly linked list; use a balanced search tree to facilitate this in logarithmic time. To answer a query, nondeterministically guess the left interval boundary and use the pointer to verify that the right interval boundary is indeed to the right of the queried point. This takes constant time. No time stamp lower bound will be able to beat this bound. In other words, we cannot hope for nonconstant time stamp lower bounds for the Union–Split–Find problem—or for related ones like existential range queries.

4.3 Amortised bounds.

The original lower bound for signed prefix sum [25] applies also to amortised bounds. Theorem 1 and the Refinement lemma are expressed in terms of worst case complexity. Indeed, the present proof of these result does not translate to an amortised lower bound in any obvious fashion, even though their proof is very close to that of [25]. This is mainly because our update sequences very much depend on their epochs. We conjecture that our results do hold in an amortised version as well.

CHAPTER 3

Symmetric Functions

And [we] have now arrived at the point of asking why are the majority bad, which question of necessity brought us back to the examination and definition of the true philosopher.

—Plato, *The Republic*

This chapter studies the dynamic complexity of symmetric functions. We have seen that the complexity of prefix-or_n is exponentially easier than prefix parity, so some symmetric functions are harder than others. We will see which and why in § 4.

The quest for connections to parallel computation encourages us to compare our findings with known results in other models; we include a review of these results with some new proofs in §§ 2 and 3.

1 SYMMETRIC FUNCTIONS

In this thesis, a *Boolean* function maps $\{0, 1\}^n$ to $\{0, 1\}$. Such a function is *symmetric* if it depends only on the number of 1s in the input $x = (x_1, \dots, x_n)$. Clearly, the value of a symmetric function is fixed under permutations of the variables. This holds also in the other direction: If f is the same function as $f \circ \pi$ for all permutation π then f is symmetric.

The symmetric functions include some of the most well-studied functions in complexity theory, examples follow. (Iverson's notation (P) means 1 if property P holds and 0 otherwise.)

$$\begin{aligned} \text{eq}_n^b(x) &= (x_1 + \dots + x_n = b), & \text{half}_n &= \text{eq}_n^{n/2}, \\ \text{par}_n(x) &= (x_1 + \dots + x_n = 0 \pmod{2}), \\ \text{th}_n^b(x) &= (x_1 + \dots + x_n \geq b), & \text{maj}_n &= \text{th}_n^{n/2}, \\ \text{and}_n &= \text{th}_n^n, & \text{or}_n &= \text{th}_n^1. \end{aligned}$$

In general, we can describe every symmetric function f in n variables by its *spectrum*, a string in $\{0, 1\}^{n+1}$ whose i th letter is the value of f on inputs where exactly i variables are 1.

The *boundary* b of a spectrum s denotes the smallest value such that $s_{\lfloor b \rfloor} = \dots = s_{\lfloor n-b \rfloor}$. The boundary of the parity function is $\frac{1}{2}n$, and the boundary of the threshold function th_n^b is $\min\{b, n-b\}$.

2 COMMUNICATION COMPLEXITY

Alice and Bob, two co-operating but distant players, each hold a set $A, B \subseteq \{1, \dots, n\}$ such that $|A| \neq |B|$. Using as little communication as possible, they want to find an element that is in one set but not in the other, i.e. in the *symmetric difference* $A \triangle B = (A - B) \cup (B - A)$.

This seems to have nothing but a (coincidental) nominal connection to symmetric functions, but the *cognoscenti* will recognise the problem as the ‘communication complexity version’ of computation of symmetric functions. This will be clarified in § 3.3.

How To Find Elements in the Symmetric Difference. We start with two protocols for finding an element in $A \triangle B$ that are obvious but nonoptimal.

Let us first see how Alice and Bob can find an element in $A \triangle B$ using $O(\log^2 n)$ bits of communication. The protocol for this is a binary search in $\log n$ rounds. Alice and Bob will maintain two integers l and r (for ‘left’ and ‘right’) such that

$$|A \cap \{l, \dots, r\}| \neq |B \cap \{l, \dots, r\}|.$$

This means that a valid answer is known to exist in the current interval $\{l, \dots, r\}$. Initially, $l = 1$ and $s = n$. The interval is halved each round, write

$$m = l + \lceil \frac{1}{2}(r - l) \rceil$$

for the ‘middle’ of l and r . Bob sends $|B \cap \{l, \dots, m\}|$ to Alice, who decides in which half to continue the search and tells Bob.

Under the stronger assumption that the parities of $|A|$ and $|B|$ differ, Alice and Bob need to send only $2\lceil \log n \rceil$ bits. They will ensure that

$$|A \cap \{l, \dots, r\}| \neq |B \cap \{l, \dots, r\}| \pmod{2}$$

during the protocol. Each round, Bob sends the parity of $|B \cap \{l, \dots, m\}|$, from which Alice can infer (and tell Bob) in which half to continue.

The next result shows how to achieve the asymptotic bound of the latter protocol under the conditions of the former.

Proposition 2 *If $|A|$ and $|B|$ differ then Alice and Bob can find an element in $A \triangle B$ using $7 \cdot \lceil \log n \rceil$ bits of communication.*

Proof. In addition to l and r as above, Alice and Bob maintain a *marker* $j \in \{1, \dots, \log b\}$, defined as follows. The marker j indicates a position where the binary representations of the player’s current sets differ, in other words,

$$|A \cap \{l, \dots, r\}| \neq |B \cap \{l, \dots, r\}| \pmod{2^j}. \quad (1)$$

Initially, such a marker can be found using at most $2\lceil \log n \rceil$ bits of communication: Bob sends the binary representation of $|B|$, starting with the most significant bit, until Alice tells him to stop.

The protocol proceeds in $\log n$ rounds as follows. First Alice and Bob have to ensure that one of the following conditions hold:

$$|A \cap \{l, \dots, m\}|_j \neq |B \cap \{l, \dots, m\}|_j \quad \text{or} \quad (2)$$

$$|A \cap \{m+1, \dots, r\}|_j \neq |B \cap \{m+1, \dots, r\}|_j, \quad (3)$$

where we write a_j for the j th bit of the binary representation of integer a . To this end, the players repeat the following until they succeed: Bob sends his 2 bits of information from (2) and (3) to Alice, who checks (and tells Bob using 1 bit) if one of the conditions holds. If not, the players decrement the marker j and retry. Because of the invariant (1), they eventually arrive at some nonzero marker j for which either (2) or (3) holds.

Alice can now tell Bob in which interval to continue the search, ending this round.

The players spend 4 bits in each of the $\lceil \log n \rceil$ rounds and an additional 3 bits every time the conditions (2) and (3) fail to hold. But the latter can only happen $\lceil \log n \rceil$ times in the entire protocol, since the marker j is decremented every time. \square

We mention for completeness that if $|A|$ and $|B|$ are not known to differ then the problem's complexity grows to $\Theta(n)$.

3 BOOLEAN CIRCUIT COMPLEXITY

We now recall results about the parallel complexity of symmetric functions, namely their *Boolean circuit* complexity. They show that the complexity of a symmetric function depends on the boundary of its spectrum, a connection that we will rediscover when we return to the dynamic setting in § 4.

3.1 Boolean Circuits

Our definition of Boolean circuits is completely standard: A *circuit* for a Boolean function $f: \{0, 1\}^n \rightarrow \{0, 1\}$ is an acyclic digraph with exactly 1 node of fan-out 0 (the *output node*) and n nodes of fan-in 0 (the *input nodes*). The input nodes are labeled with the integers $1, \dots, n$ and represent the inputs to f , all internal nodes are labeled \vee , \wedge , or \neg . The fan-in of \neg -gates is 1. If the fan-in of \vee - and \wedge -gates is 2, the circuit has *bounded fan-in*. Computation is defined in the obvious way.

The *depth* of a circuit is the length of its longest path. The depth of a function is the depth of the shallowest circuit that computes it. The *size* of a circuit is the number of its nodes.

3.2 Bounded Fan-In Circuits

This section considers the complexity of evaluating symmetric functions with Boolean circuits of bounded fan-in. The results of this section are well-known. However, the proofs are new and quite elegant and clearly present the inherent dependence of the parallel complexity on b .

We consider circuits of fan-in two over the basis \vee , \wedge , and \neg . For a function f we let $d(f)$ denote the depth of the shallowest circuit that computes it.

The proof uses the communication complexity result of the previous paragraph.

Proposition 3 *If $f: \{0, 1\}^n \rightarrow \{0, 1\}$ is symmetric then it has bounded fan-in Boolean circuit depth $\Theta(\log n)$.*

Our proof uses the well-known equivalence result of Karchmer and Wigderson [39]. Let $f: \{0, 1\}^n \rightarrow \{0, 1\}$ be a Boolean function. In the *Karchmer–Wigderson game for f* Alice receives $A \in f^{-1}(0)$, Bob receives $B \in f^{-1}(1)$, and they want to find an index where their input strings differ. Alternatively, if we view A and B as (the incidence vectors of) subsets of $\{1, 2, \dots, n\}$, Alice and Bob look for an element in $A \triangle B$. The communication complexity of the game is the minimal number of bits they have to exchange.

Proof of Prop. 3. Consider the game where Alice receives a set A in $f^{-1}(0)$ and Bob receives a set B in $f^{-1}(1)$. Then $|A|$ and $|B|$ differ because f is symmetric. By Prop. 2 the players can in logarithmic time find an index where their inputs differ. The result of Karchmer and Wigderson [39] tells us that exactly the same bound holds for the Boolean circuit depth of f . \square

We note that the construction is far from optimal with respect to the constants involved. See Boppana and Sipser [10] for a survey. Bounds on complexity of symmetric functions in terms of the boundary of the function’s spectrum exist for this class of circuit. However, the influence of this parameter is much more pronounced in the world of Boolean circuits with *unbounded fan-in*.

3.3 Unbounded Fan-In Circuits

This section surveys the complexity of symmetric functions in Boolean circuits with unbounded fan-in.

The famous Håstad lower bound says that the depth d of circuits for the parity function must satisfy

$$d = \Omega\left(\frac{\log n}{\log \log s}\right), \quad (1)$$

where s denotes the circuit’s size. For general symmetric functions the bound is

$$d = \Omega\left(\frac{\log b}{\log \log s}\right), \quad (2)$$

where b denotes the function’s boundary.

This characterises the symmetric functions with constant depth and polynomial size circuits (so-called AC^0 functions), since it is known that a symmetric function f has such circuits if only

$$b = \log^{O(1)} n$$

for all symmetric f . The lower bound (2) shows that the depth d becomes nonconstant for polynomial size s as soon as the boundary b is larger than polylogarithmic.

3.4 Roots.

The standard proof of Prop. 3 involves a clever addition circuit that computes the number of 1s in the input with logarithmic depth. This construction was discovered by several authors independently in the early 60s, among them Wallace [69] and, *à la russe*, Ofman [55]. Today, complicated constructions have reduced the constants in the complexity bound far below those of Prop. 3, see the references in [10].

Håstad's bound (1) for the depth of the parity function is from [33] and improves work of Ajtai [1] and Furst, Saxe, and Sipser [26]. Constant depth circuits for threshold functions with polylogarithmic boundary have been constructed by Ajtai and Ben-Or [3], Denenberg, Gurevich, and Shelah [15] and Fagin *et al.* [20]. These results were generalised to symmetric functions and expressed in terms of the boundary by Brustmann and Wegener [13] and Moran [54].

4 CELL PROBE COMPLEXITY

We return to dynamic computation in the cell probe model and will discover results that imitate those for circuits of unbounded fan-in. The gist is that the *dynamic prefix problem* (defined below) of a symmetric function is the same as the function's Boolean circuit depth with polynomial size unbounded fan-in circuits. To our knowledge this is the best formalisation of the widely held intuition that the complexity of dynamic and parallel computation are related. Not only do the circuit depth and the dynamic prefix problem depend on the same combinatorial properties of the symmetric functions (namely the size of their boundary b), even the quantitative aspects of trade-off results mimic each other: circuit depth depends on circuit size exactly as query time depends on word size and update time.

4.1 Threshold functions

We start with the threshold functions. Section 4.2 studies a more general problem and contains a stronger result, but threshold functions allow us to tell the best part of the the story without covering it in too many details. We will compare the hardness of functions like $\text{th}_n^{\lceil \log n \rceil}$, $\text{th}_n^{\lceil \sqrt{n} \rceil}$, and $\text{maj}_n = \text{th}_n^{\lceil n/2 \rceil}$.

Let the *boundary function* b be an integer function such that $b(i) \in \{0, \dots, \lceil \frac{1}{2}i \rceil\}$. We will study the threshold function th_i^b given by $\text{th}_i^b(x_1, \dots, x_i) = (x_1 + \dots + x_i \geq b(i))$. The *dynamic prefix problem for* th_n^b is to maintain a bit string $x \in \{0, 1\}^n$ under the following operations:

change(i): change x_i to $\neg x_i$,

query(i): return $(x_1 + \dots + x_i \geq b(i))$.

Note that the boundary $b(i)$ for the i th query depends on i and not on n . Consider for example the majority function $b(i) = \lceil \frac{1}{2}i \rceil$: its prefix query returns $(x_1 + \dots + x_i \geq \lceil \frac{1}{2}i \rceil)$, i.e., ‘is there a majority of 1s in the i th prefix?’.

RAM algorithms. Let us briefly see how fast we can solve this problem on the random access machine.

Proposition 4 *The dynamic threshold prefix problem for th_n^b can be solved on the RAM with logarithmic cell-size in time $O(\log b / \log \log n + \log \log n)$ per update, if $b(1), \dots, b(n)$ can be computed in the preprocessing stage of the algorithm.*

Proof (sketch). Let B denote $\max\{b(1), \dots, b(n)\}$ and note $B \in O(b)$. The data structure consists of a search tree for the (indices of the) B leftmost 1s in x and a priority queue for the others. For a prefix query, the index of the $b(i)$ th 1 of x is found in the search tree and compared with i . Standard data structures yield update times of order

$$O\left(\frac{\log b}{\log \log n} + \log \log n\right),$$

where the right term stems from the priority queue over $\{1, \dots, n\}$.

A number of recent results discuss the RAM instruction sets under which this works, which is outside the scope of this thesis. So are details about the computability of b : If b is a difficult function, the algorithm has to make a table of $b(1), \dots, b(n)$ in the preprocessing stage of the algorithm. \square

Lower bounds. Assume that there is a subset of the naturals where b is monotone and onto:

$$\text{there are } p(1) < p(2) < \dots < p(i) \dots \text{ such that } b(p(i)) = i.$$

We will call such functions *nice* for lack of a better word. Functions like $\lceil \log n \rceil$, $\lceil \sqrt{n} \rceil$ and $\lceil \frac{1}{2}n \rceil$ are nice, so is $\lceil g \rceil$ for any smooth unbounded convex function with $0 \leq g(i) \leq i$.

The lower bound for this problem depends on b in exactly the same way as for Boolean circuit complexity, compare (2).

Theorem 2 (for threshold functions) *Let $t_u = t_u(n)$ and $t_q = t_q(n)$ denote the update and query time of any cell size c implementation of the dynamic prefix problem for th_n^b for a nice boundary function b . Then*

$$t_q = \Omega\left(\frac{\log b}{\log(t_u c \log b)}\right).$$

Proof. The proof is a reduction from signed prefix sum and starts similar to that of Prop. 1.

First look at the function half_n given by $\text{half}_n(x) = (x_1 + \dots + x_n = \lceil \frac{1}{2}n \rceil)$. Let $x \in \{+1, 0, -1\}^n$ denote an instance to signed prefix sum. Let d be given as

$$d = \left\lceil \frac{\log n}{\log(t_u c \log n)} \right\rceil$$

and construct $d + 1$ strings $y^{(t)}$ as

$$y^{(t)} = (-1)^t 0^{d-t}, \quad 0 \leq t \leq d.$$

For convenience we index the $y^{(t)}$ using the set $\{-d, \dots, -1, 1, \dots, n\}$, so that $y_i^{(t)}$ is the same as x_i for all $1 \leq i \leq n$. By construction, we have

$$y_{-d}^{(t)} + \dots + y_i^{(t)} = t + x_1 + \dots + x_i. \quad (1)$$

as long as $x_1 + \dots + x_i$ is less than d . To find the i th prefix sum of x we guess $s \in \{0, \dots, d\}$. Our guess is the correct prefix sum if and only if $y_{-d}^{(-s)} + \dots + y_i^{(-s)}$ vanishes.

This means that we can construct a data structure for signed prefix sum as follows. We maintain the $d + 1$ strings $y^{(t)}$ as bit strings of length $2(n + d)$ using the encoding

$$+1 \mapsto 11, \quad 0 \mapsto 01, \quad -1 \mapsto 00.$$

We can maintain these strings in time $2(d + 1) \cdot t_u(2n + 2d) + O(1)$ whenever x is changed. A prefix sum of $y^{(t)}$ vanishes if and only if there is the same number of 0s and 1s in the corresponding bit string, so the query time is $t_q(2n + 2d) + O(1)$. We conclude from Thm. 1 that $t_q = \Omega(\log n / \log(t_u c \log n))$.

The same bound must hold for the majority function $\text{maj}_i = (x_1 + \dots + x_i \geq \lceil \frac{1}{2}i \rceil)$, for we can express $\text{half}_i(x)$ as $\text{maj}_i(x) \wedge \text{maj}_i(\bar{x})$, where $\bar{x} = (\bar{x}_1, \dots, \bar{x}_i)$ consists of the negated values of x , which are easily maintained.

Now let b be any nice function and let $p(1), \dots, p(n)$ be such that $b(p(i)) = i$ holds. Assume we have an algorithm for the prefix problem for $\text{th}_n^{b(n)}$ with the parameters given in the statement of the theorem. We will construct an algorithm for the prefix problem for maj_n with instance $x \in \{0, 1\}^n$. Construct a bit string y as

$$y = 0 \dots 0x_1x_10 \dots 0x_2x_20 \dots 0x_nx_n,$$

where the letters of x are put in position $p(1) - 1, p(1), p(2) - 1, p(2), \dots, p(n) - 1, p(n)$; denote the length of y by $m = p(n)$.

The string y can be maintained in time $2t_u(m)$ for each update of x . For the query note that $2x_1 + \dots + 2x_i$ equals $y_1 + \dots + y_{p(i)}$, so

$$x_1 + \dots + x_i \geq \lceil \frac{1}{2}i \rceil \quad \text{if and only if} \quad y_1 + \dots + y_{p(i)} \geq i = b(p(i)),$$

so the majority function (left hand side) can be expressed in terms of $\text{th}_i^b(i)$ (right hand side). Hence the query time is $t_q(m)$. But from the bound on the complexity of the majority function we know

$$t_q(m) = \Omega\left(\frac{\log n}{\log(t_u(m)c(m)\log n)}\right).$$

The stated bound follows by substituting $b(m)$ for n . \square

The RAM algorithm from Prop. 4 shows that the lower bound is tight for logarithmic cell size and $b = \Omega(\log^{\log \log n} n)$.

4.2 General Symmetric Functions

We are interested in more general statements about other symmetric functions like equality, parity, or combinations like

$$\text{th}_n^{\lceil \sqrt{n} \log \log n \rceil} \wedge \text{par}_n.$$

It is not hard to change the last proof to work for equality functions, but this approach breaks down for spectra that are not very well-behaved. The non-deterministic algorithm from the last proof essentially guesses the position of a substring ‘10’ in the spectrum. For threshold and equality functions, it can make no false guesses since the substring appears only once in the spectrum, namely at b . But for spectra like ‘1110010000’ the present, this approach is doomed, the substring appears at positions 3 and 6. Of course, many individual spectra can be decomposed into easier cases, but we are looking for a general way to handle these functions.

Let $\langle f_n \rangle = (f_1, \dots, f_n)$ be a sequence of symmetric Boolean function where the i th function f_i takes i variables. The *dynamic prefix problem* for $\langle f_n \rangle$ is to maintain a bit string $x \in \{0, 1\}^n$ under the following operations:

change(i): change x_i to $\neg x_i$,

query(i): return $f_i(x_1, \dots, x_i)$.

Taking f_i to be $\text{th}_i^{b(i)}$ we have the same problem as in the last section, taking f_i to be par_i we have the prefix parity problem of [25].

Theorem 2 *Let b be a nice boundary function. Let $\langle f_n \rangle$ be a sequence of symmetric functions where $f_i: \{0, 1\}^i \rightarrow \{0, 1\}$ has boundary $b(i)$. Let t_u and t_q denote the update and query time of any cell size c implementation of the dynamic prefix problem for $\langle f_n \rangle$. Then*

$$t_q = \Omega\left(\frac{\log b}{\log(t_u c \log b)}\right).$$

Proof. The proof is a reduction from parity prefix refinement but otherwise similar to the proof for threshold functions.

First assume that f_i ’s boundary is in the middle, i.e. $b(i) = \frac{1}{2}i$ (for example, take h_i to be $\text{half}_i \wedge \text{par}_i$). Let $x \in \{+1, 0, -1\}^n$ denote an instance to prefix parity refinement and define $2d + 1$ strings in the previous proof.

Hence we can use the data structure for f_n to perform *refine*(i, s) as follows. Recall that one of $\{s - 1, s, s + 1\}$ is the right guess. If s is the right guess we have $x_1 + \dots + x_i = s$, so by (1),

$$y_d^{(-s)} + \dots + y_i^{(-s)} = 0 \tag{2}$$

and

$$y_d^{(-s+1)} + \dots + y_i^{(-s+1)} \neq 0. \quad (3)$$

If $s + 1$ is the right guess then (2) fails. If $s - 1$ is the right guess then (3) fails. So we can distinguish the three cases using only two queries, using the same encoding $\{-1, 0, +1\} \rightarrow \{0, 1\}^2$ as before. Conclusion by the Refinement lemma.

The rest of the proof, ‘stretching’ this result to smaller b , is the same as for threshold functions. \square

Important special cases are mentioned in the next result:

Corollary 1 *The complexity of the dynamic prefix problem for $\text{par}_n(x) = (x_1 + \dots + x_n = 0 \pmod 2)$, $\text{maj}(x) = (x_1 + \dots + x_n \geq \lceil \frac{1}{2}n \rceil)$, and $\text{half}(x) = (x_1 + \dots + x_n = \lceil \frac{1}{2}n \rceil)$ is $\Theta(\log n / \log \log n)$, for any cell size c with $\log n \leq c \leq \log^{O(1)} n$.*

The upper bound is from [16] and the result for parity is of course just [25], but for the other functions no lower bound larger than $\Omega(\log \log n / \log \log \log n)$ was known.

It is interesting to note that in Boolean circuit complexity the hardness result for parity immediately implies a hardness result for majority and threshold functions, because there is an obvious way to build circuits for the former from circuits for the latter. In contrast, there is no obvious cell probe reduction for the dynamic version; we have seen that we had to go via a strengthening of the model (namely, nondeterminism or refinement) to make the reduction work.

Cell probe implementations. To gauge the strength of our lower bound, we look at cell probe implementations.

Proposition 5 *Let $\langle f \rangle$ be a sequence of symmetric functions where $f_i: \{0, 1\}^i \rightarrow \{0, 1\}$ has boundary $b(i)$. The dynamic prefix problem for $\langle f \rangle$ can be solved in time $O(\log b / \log \log n)$ for cell size $c \geq \log^2 n$ and in time $O(\log b / \log \log n + \log \log n)$ for cell size $\log n \leq c \leq \log^2 n$.*

Proof (Sketch). The construction is similar to that from Prop. 4. For the i th query, look up index i in the search tree. The rank of that index (or its left neighbour if the index is not in the tree) is $\min\{b(i), x_1 + \dots + x_i\}$, from which $f(i)$ can be computed.

For larger cell size $c \geq \log^2 n$, there are priority queues over $\{1, \dots, n\}$ that use only constant time per update [11, 67]. (These work also on reasonable RAMs, so we could parameterise the statement of Prop. 4 with the cell size to get the same bounds.) \square

Summary. For nice boundary functions, the cell probe complexity of the prefix problem for any symmetric function is $\Theta(\log b / \log \log n)$ per operation

1. for any b and cell size $c \geq \log^2 n$, or,
2. for $b \in \Omega(\log^{\log \log n} n)$ and cell size $c \geq \log n$.

If simultaneously b and c are very small, there is a factor $\log \log n$ between our bounds. Especially, the complexity of prefix-or for logarithmic cell size remains unknown. It can be shown that with *nondeterministic* queries, our bounds is tight for all c and b . We conjecture that the deterministic complexity is the same as the nondeterministic complexity, i.e., that the lower bound is optimal. Especially we conjecture that the complexity of prefix-or is constant.

Range queries. These are useful lower bounds for certain *range query* problems.¹ The problem is to maintain a d -dimensional set $S \subseteq \mathbb{R}^d$ (for our lower bound, $d = 1$ is hard enough) under the following operations:

insert(x): Insert a point at coordinate $x \in \mathbb{R}^d$ into S ,

delete(x): Remove the point at $x \in \mathbb{R}^d$ from S ,

report(K): How many points are in $K \cap S$, where K is a rectangle in \mathbb{R}^d .

The problem has been studied for many other query operations and our understanding of its complexity varies with the type of query. For *counting* (as above), the Fredman–Saks bound applies even in one dimension. On the other hand, the problem of *existential range queries* (return ‘yes’ iff $K \cap S$ is nonempty) is among the most interesting problems at the time of writing, see [50] for some results.

Our lower bounds apply to versions of the problem where the query operation is a symmetric function, e.g., the majority function, in some disguise. Here is one:

insert(x, c): insert $x \in \mathbb{R}^d$ of colour $c \in \{\text{blue, red}\}$ into S ,

delete(x): remove the point at x if it exists,

blue(K): are there more blue than red points in $K \cap S$?

This corresponds to asking questions like ‘among the students aged 20 to 25, are there more males than females?’. Alternatively, in the monochromatic setting, we can ask: ‘Are there more students aged 20 to 25 than 23 to 30?’, reflected in the following query:

more(K_1, K_2): is $|K_1 \cap S| > |K_2 \cap S|$?

¹This entire material could have been presented as a study of one-dimensional range queries instead of prefix problems for Boolean function.

Discussion. What about boundary functions that are not nice? In the general definition, nonuniformity rears its ugly head. We cannot hope to prove clean statements about the complexity of this problem without any constraints are put on the sequence, much less hope for matching bounds. This is because of sequences that alternate between very easy (e.g., constant) functions and very hard ones (e.g., parity or ‘does the i th Turing machine halt?’) in some complicated way.

On the other hand, the niceness condition can easily be relaxed. For example, the function $b(i) = 2 \cdot \lfloor \frac{1}{2}i \rfloor$, whose first values are 0, 0, 2, 2, 4, 4, 6, 6, \dots , is not nice (it is not onto) but th_n^b is easily seen to be hard. However, our aim was for a simple definition that covers interesting functions, not an encyclopedic treatment.

Roots. The complexity of the prefix parity problem was characterised by Fredman and Saks [25] and Dietz [16]. Progress on prefix majority and equality was reported by Husfeldt, Rauhe, and Skyum [36], where lower bounds of order $\Omega(\log n / \log \log^2 n)$ and $\Omega(\sqrt{\log n / \log \log n})$ were found. Results on prefix problems for finite monoids with relevance to prefix-or are reported by Frandsen, Miltersen, and Skyum, [22], and Miltersen [49]. Fast priority queues for $\{1, \dots, n\}$ are given by Thorup [67] and Brodal [11]. For a recent survey of search trees on the RAM, see Anderson [5].

CHAPTER 4

Graphs

Graph problems, including problems on planar structures in computational geometry and graph drawing, have been the focal point of dynamic algorithms. Seminal work includes the dynamic convex hull algorithm by Overmars and van Leeuwen [57], Frekerickson’s algorithm for maintaining a minimum spanning tree in a plane graph [23], and the dynamic tree data structure of Sleator and Tarjan [62]. Recent successes in the field include the technique of sparsification [18] and the efficient algorithm for dynamic reachability in undirected graphs of Henzinger and King [32]. Albers, Cattaneo, and Italiano [4] report empirical results for dynamic graph algorithms. A forthcoming handbook chapter [17] provides an introduction with focus on undirected graphs, because “designing efficient fully dynamic data structures for directed graphs has turned out to be an extremely difficult task.”

Nonetheless, in §1 we present an efficient algorithm for the dynamic reachability problem for the class of planar digraphs with one source and one sink. In §2 we use our hardness results from Chap. 2 to provide new lower bounds for a handful of graph problems with efficient algorithms. Table 1 gives an overview.

In this chapter we consider only logarithmic cell size for no other reasons than compatibility with the literature.

1 ALGORITHMS

1.1 Planar Source–Sink Graphs

In this section we give an algorithm for the dynamic reachability for *planar source–sink graphs*, i.e., directed acyclic graphs that are drawn in the plane without intersecting edges and have exactly one source and one sink, see Figure 4.

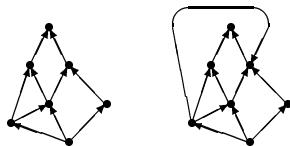


Fig. 4: Two planar source–sink graphs

Problem	Lower bound	Upper bound
Reachability in		
planar source–sink graphs	$\Omega\left(\frac{\log n}{\log \log n}\right)$ [25]	$O(\log n)$ †
upward planar source–sink graphs	$\Omega\left(\frac{\log n}{\log \log n}\right)$ †	$O(\log n)$ [65]
grid graphs	$\Omega\left(\frac{\log n}{\log \log n}\right)$ †	$O(\log n)$ [27, 19]
Upward planarity testing	$\Omega\left(\frac{\log n}{\log \log n}\right)$ †	$O(\log n)$ [64]
Monotone point location	$\Omega\left(\frac{\log n}{\log \log n}\right)$ †	$O(\log n)$ [7]

Tab. 1: Overview of graph results with references. Results from this thesis are marked with †.

Preliminaries A graph is *embeddable* on a surface if it can be drawn on the surface such that the edges do not intersect except at their endpoints. A graph is *planar* if it is embeddable in the plane and it is *upward planar* if in that embedding all edges point upward.

For node v of a digraph we let $\deg^+(v)$ and $\deg^-(v)$ denote its out- and indegree, respectively. A vertex v is a *source* if $\deg^-(v) = 0$, and a *sink* if $\deg^+(v) = 0$. We are now ready to define the class of graphs studied in this paper.

A digraph is a *source–sink graph* if it is acyclic and has exactly one source and one sink. If it can be embedded so that the source and the sink are on the same face, the graph is an *upward planar source–sink graph*.

Figure 4 shows two planar source–sink graphs, the left of which is also upward planar. The following properties of this class of graphs can be shown:

1. Every vertex is on a simple directed path from the source to the sink.
2. In every embedding, the incoming edges to any vertex appear consecutively around the vertex, and so do the outgoing edges; this determines the *left face* $\text{left}(v)$ and the *right face* $\text{right}(v)$ of a vertex, see Figure 5. This implicitly defines an order of the edges appearing around v , say, from the leftmost outgoing edge to the leftmost incoming edge in the clockwise direction. We will sometimes refer to this order as the *ordering of the edges around v* .
3. The boundary of every face consists of two directed paths with common origin and terminus vertices, see Figure 5.
4. Every planar source–sink graph can be embedded on the *sphere* such that all edges are directed upward (i.e., their projection on some fixed direction

is positive). For example, we could embed the graph from Figure 4 by placing the curved arc on the opposite side of the sphere.

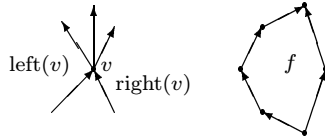


Fig. 5: A vertex and a face in a planar source-sink graph

In the rest of this section, $G = (V, E)$ will denote a planar source-sink graph with source s and sink t , vertices V and edges E . We let n denote the number of edges in the graph. For brevity, we will sometimes use the notation $u \prec v$ if there is a path from u to v . We will write $u \parallel v$ if neither $u \prec v$ nor $v \prec u$.

Source-sink graphs are often called *st-graphs* in the literature.

Dynamic Reachability We consider the dynamic reachability problem for planar source-sink graphs. Namely, we present a data structure that handles the following operations (for clarity, we have spelt out the embedding-preserving restrictions on the update operations):

insert(u, v): Insert an edge from vertex u to vertex v if they are on the same face and the new edge does not induce a directed cycle,

delete(u, v): Delete the edge from vertex u to vertex v provided $\deg^+(u) \geq 2$ and $\deg^-(v) \geq 2$,

query(u, v): return ‘yes’ if and only if there is a path from vertex u to vertex v .

Another good name for this problem is *dynamic transitive closure*, since the aim is to maintain the transitive closure relation \prec of G . *To maintain* here means to be able to answer queries about the transitive closure, for it is easy to see that the relation itself may change a lot with a single update, so there is no hope of maintaining an *explicit* representation (e.g., as an incidence matrix) of the relation in logarithmic time.

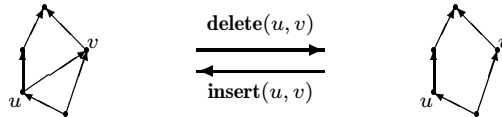


Fig. 6: Updates

We will present an algorithm for this problem that handles updates and queries in time logarithmic in the number of edges of the graph. The data

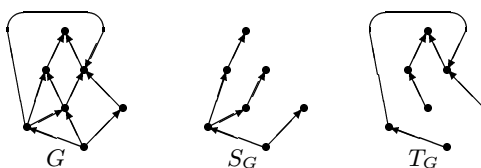


Fig. 7: A graph G with corresponding trees S_G and T_G .

structure can be initialised in linear time and uses linear space. This is Theorem 3. Together with a lower bound from § 2 this characterises the complexity of dynamic reachability on this class of graphs within a $\log \log n$ factor. The algorithm is pleasantly simple and should be easy to implement efficiently (the most complicated part is the dynamic tree data structure from [62], which also contains a discussion of implementation issues). The analysis is less simple and takes up most of this section, which is the longest in this thesis. For ease of presentation we will state and prove increasingly general versions of Theorem 3 step by step.

1.2 Reachability in Source–Sink Graphs

Tree decomposition. We employ an idea used in many efficient dynamic graph algorithms: Decompose the graph into a number of trees such that all the necessary information can also be derived from the trees.

The tree S_G is the subgraph of G constructed by removing all edges that are not the leftmost *incoming* edge of any vertex. Similarly, the tree T_G is constructed by removing all edges that are not the leftmost *outgoing* edge to any vertex. When the graph is fixed, we will drop the subscripts on S and T .

See Figure 7, which shows S and T for the graph from Figure 4. Observe the following facts:

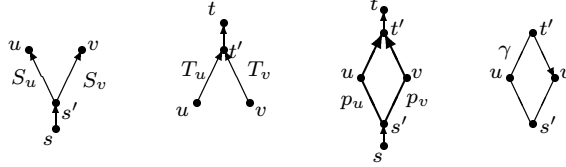
1. S and T are indeed trees,
2. S is divergent and rooted at s , while T is convergent and rooted at t (hence the names),
3. no subpath of T can ever leave another path to the right, and no subpath of S can ever enter another path from the right.

Let us emphasise the last innocent-looking and obvious item, since we will use it quite often:

Fact 1 *If a subpath of T crosses a subpath of S , it does so from right to left.*

We need some notation. For vertex $v \in V$ we let S_v denote the unique path from s to v in S and let T_v denote the unique path from v to t in T . For $u, v \in V$ we let s' denote the last vertex that is on both S_v and S_u . Let t' denote the first vertex that is on both T_v and T_u . The path p_u is the subpath of the concatenation of S_u and T_u from s' to t' . Symmetrically, p_v is the sub-path

of the concatenation of S_v and T_v from s' to t' . The figure below depicts this construction.



Whenever it seems convenient, we will also refer to the two paths as p_L and p_R , such that p_L is the path leaving s' to the left and p_R is the other path.

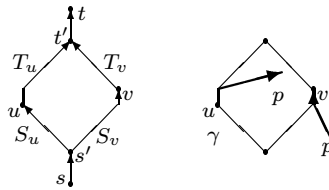
We will boldly confuse the edges of G with their embedding to alleviate notation. Namely, we introduce the curve γ which is the concatenation of (the embeddings of) p_L and p_R . The orientation of γ will be such that it agrees with the direction of p_L and the reversed direction of p_R . Recall that a curve is *closed* if its endpoints coincide, it is *simple* if it does not intersect itself except at its endpoints. Note that γ is closed and not necessarily simple.

The next lemma is the *crux* of our algorithm. It captures the following fact about reachability in spherical st -graphs: To get from vertex u to vertex v one can always choose a path whose first half stays in T and whose last half stays in S .

Lemma 3 *Let \leq_S and \leq_T denote the predecessor relation in S and T , respectively. Then $u \prec v$ if and only if*

$$\exists w \in V : u \leq_T w \wedge w \leq_S v.$$

Proof. Assume for contradiction that there is a path p from u to v even though S_v and T_u are vertex-disjoint.



Note that S_u crosses neither S_v (else S would not be a tree) nor T_u (else G would have a cycle). Similarly, T_v crosses neither T_u nor S_v nor S_u (the latter would form a cycle with p). So we have the situation depicted to the left in the above figure modulo the symmetrical case where u appears to the right of v .

Without loss of generality, we can split p into three parts p_u , p' and p_v , such that p_u is a (possibly empty) sub-path of T_u , p_v is a (possibly empty) sub-path of S_v and p' (which contains at least one vertex) has no vertices in common with either T_u or S_v .

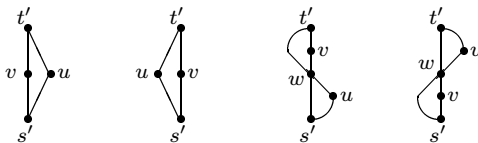
Note that p' leaves T_u before t' (else there would be a cycle in G) and does so to the right by Fact 1. Similarly, p' enters S_v after s' and does so from the right. The right part of the figure above conveys the absurdity of this: Part of p' is in the interior of γ , while another part is in the exterior. Hence p' must cross γ somewhere, but cannot by construction. \square

Upward Planar Graphs To see some of the present machinery in motion and to get our hands dirty before we study the full problem, let us derive an algorithm for *upward* planar source–sink graphs.

We must handle the existential quantifier of the last lemma without searching all of V . We will show that the existence of w ‘between u and v ’ can be read off the edges around s' and t' .

Lemma 4 *In a plane st -graph, the reachability information between u and v is uniquely determined by the appearance of p_u and p_v around s' and t' .*

Proof. The proof is a case analysis on the behaviour of p_u and p_v between s' and t' . We shall see that there are only four cases, depicted below.



First note that if $s' = u$ then there is a path from u to v and we are done. Similarly, the cases $s' = v$, $t' = u$, and $t' = v$ are trivial.

Assume first that p_u leaves s' to the right of p_v . There are two cases: Either p_u stays to the right of p_v (until the two paths finally meet at t') or it does not. In the former case (the leftmost example in the figure), there cannot be a path from v to u by Lemma 3.

In the latter case, p_u must cross p_v at some point to get to the other side. It cannot enter it anywhere except between s' and t' , by acyclicity of G and construction of t' , hence it enters at some vertex $w \neq t'$. Since w is on both p_u and p_v , one of the following must hold: (i) $u \prec w$ and $v \prec w$, (ii) $u \prec w$ and $w \prec v$, (iii) $w \prec u$ and $v \prec w$, or (iv) $w \prec u$ and $w \prec v$. The reader should check that all possibilities but the second contradict Fact 1 or induce an undirected cycle in S or T . Hence, by transitivity of \prec , we have $u \prec v$. Similar arguments show that once p_u has reached the left side of p , it cannot come back; hence it enters t' left of p_v . This is the third example in the figure above.

We can repeat the analysis for the case where p_u leaves s' left of p_v (depicted by the second and fourth examples), to complete Tab. 2.

Put succinctly, u and v are connected if and only if p_u and p_v ‘switch sides.’ \square

We can re-prove the following theorem of Tamassia and Preparata [65], using a different characterisation.

p_u leaves s' right of p_v	y	y	n	n
p_u enters t' right of p_v	y	n	y	n
Reachability	$u \parallel v$	$u \prec v$	$v \prec u$	$u \parallel v$

Tab. 2: Reachability in the plane case

Theorem 3 (Upward planar case [65]) *Dynamic reachability for plane st -graphs can be solved in time $O(\log n)$, where n denotes the number of edges. The data structure uses linear space and can be initialised in linear time.*

Proof. We first describe the data structure. With every vertex v we store two sequences of the incoming and outgoing edges of v , respectively, ordered according to the cyclic ordering around v . We can use balanced search trees for this.

In addition, we maintain the trees S and T using the *dynamic tree* data structure of Sleator and Tarjan [62].

After each insertion or deletion we must reorganise our data structures. An edge can be inserted into or deleted from the edge list around a vertex in time $O(\log n)$; maintaining the two dynamic trees is a standard technique.

To answer a query event u and v in S to find their nearest common ancestor s' , see [62]. Evert u and v in T to find their nearest common ancestor t' . From the edge lists around s' and t' we see which of p_u and p_v appears rightmost. By Tab. 2, this yields the reachability information. \square

1.3 Reachability: General Case

The gist of the last section was that

1. if u and v are connected, then p_u and p_v intersect,
2. if p_u and p_v intersect, then they ‘switch sides,’ i.e., they appear around s' in another order than they do around t' .

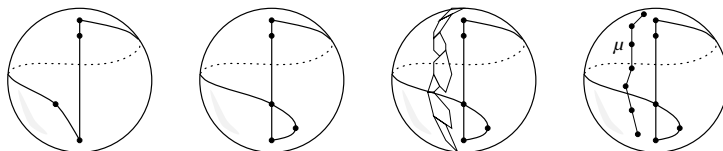


Fig. 8: The sphere: problems (left) and remedy (right).

The first item still holds in the general case. The second does not. The first two figures above show why the sphere is much more difficult than the plane: Paths can wrap around; the reader can easily check that both examples contradict Tab. 2. The remedy is to keep track of the globe-trotting of γ by

p_R right of p_L at t'	y	n	n	y	y	n	n	n	y	y	n	n
p_u right of p_v at s'	-	-	y	n	y	n	y	n	y	n	y	n
Index of t	0	1	0	1+	1+	2+	1+	0	1+	1+	2+	1+
Orientation of t	-	○	-	○	○	○	○	-	○	○	○	○
Reachability	$u \parallel v$		$u \prec v$					$v \prec u$				

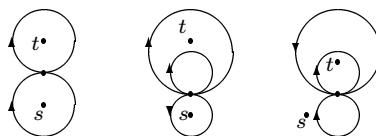
Tab. 3: Reachability in the general case. Dashes denote arbitrary or undefined entries.

maintaining a chain of faces between the poles, as indicated in the third figure; it is helpful to view this chain of faces as a path μ in the dual of the graph. The chain is called *the meridian* and formally introduced in § 1.4. First, we introduce some additional concepts to be able to formalise what we just sketched.

Index. A *region* is a maximal topologically connected subset in the complement of γ . A curve is *proper* if it intersects γ only at points where γ does not intersect itself. We define the function Ind that maps points to integers as follows: For x in a region the *index* $\text{Ind}(x)$ is the minimum number of intersections between γ and μ over all proper curves μ from s to x . Note that Ind is constant on every region, vanishes on the region of s , and in the plane case, also on the region of t .

For $\text{Ind}(t) > 0$, we define the *orientation of t* as follows: Let x be a point in a region incident to the region of t such that $\text{Ind}(x) = \text{Ind}(t) - 1$. Let μ be a proper curve from x to t that crosses γ only once. Then the orientation of t is *positive* if μ crosses γ from left to right, and *negative* otherwise.

Perhaps more intuitively, the orientation of t is the direction of the closed curve that separates the region of t from its neighbouring region with lower index. If this curve is oriented clockwise, the orientation of t is positive. The figure below shows some examples where $\text{Ind}(t) = 2$ and the orientation of t is positive.



The next lemma, which is the spherical analogue to Lemma 4, states that the concepts we introduced suffice to characterise the reachability information.

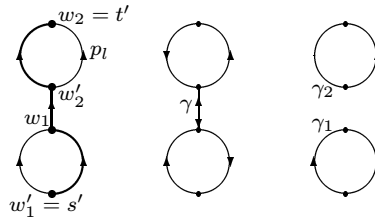
Lemma 5 *The reachability information between u and v is uniquely determined by (i) the index of t , (ii) the orientation of t , and (iii) the appearance of p_u and p_v around s' and t' .*

As Tab. 2 did in the upward planar case, Tab. 3 shows the precise connection. Note that indeed the reachability information is uniquely determined by the information above the rule. As one would expect, the case analysis is considerably more complicated than for the plane case. Figure 9 shows the possible

behaviour of p_u and p_v and can be used as a graphical proof of the lemma. The reader should check that all cases are consistent with Tab. 3.

Obviously, the sceptical reader should have no reason to believe that the examples in Figure 9 exhaust all possible cases. Unfortunately, the formal proof is somewhat tedious and un-intuitive. We confine it to the next section. At first reading the reader may simply choose to accept the result and continue to § 1.4.

Proof of Lemma 5 We have chosen to split the proof into a series of (easy) lemmas. We begin with some concepts that give a more fine-grained view of γ . Assume that p_R enters p_L at vertices w_1, \dots, w_k , with $w_k = t'$, and leaves it at vertices w'_1, \dots, w'_k , with $w'_1 = s'$ (the ordering agrees with the topological ordering of the vertices). Then for $i = 1, \dots, k$, the curve γ_i consists of the subpath of p_L from w'_i to w_i and the (reversed) subpath of p_R from w_i to w'_i .



The figure above gives an example. Note that all γ_i are subcurves of γ . On the other hand, not all of γ is necessarily part of some γ_i . The following lemma follows easily from the construction.

Lemma 6 *Let $\gamma_1, \dots, \gamma_k$ be a collection of curves as above. Then*

1. every γ_i is a simple closed curve,
2. for $i \neq j$, the curves γ_i and γ_j are disjoint except for the case $j = i + 1$, where they may intersect at $w_i = w'_{i+1}$.

Proof. Clearly, every γ_i is closed. Moreover, it consists of a part from p_R that cannot intersect itself (else there would be a cycle in G) and does not intersect p_L before w_i by construction; likewise, p_L does not intersect itself, so γ_i is simple. The same argument shows that two curves cannot intersect except as stated. \square

Let us introduce a shorthand notation that captures the way p_L and p_R cross. The *entrance sequence* E of p_R and p_L is a string of k letters from $\{R, L\}$ defined according to how the two paths cross. There is a letter in the sequence for every w_i , and that letter is an R if p_R enters p_L from the *right* at w_i , and an L if it enters from the *left*. Note that p_R enters p_L at least once, namely at t' , so the entrance sequence is nonempty. The entrance sequence for the example above is RL. Let us show that all letters but possibly the last are the same.

Lemma 7 $E \in R^+ \cup L^+R \cup R^+L \cup L^+$.

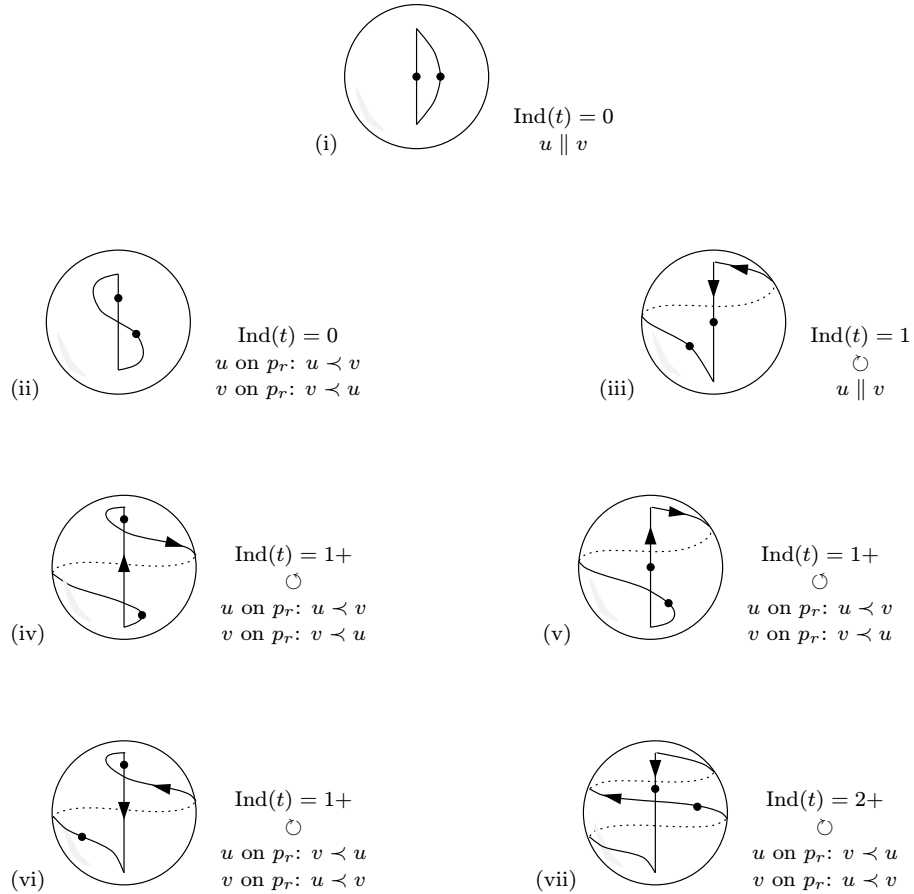


Fig. 9: Canonical examples of the behaviour of p_u and p_R on the sphere. The two topmost cases appear also in the plane, while the five other cases exploit the possibility to travel around the sphere. In all cases we give the index of t , and, if the latter is nonzero, the orientation of the region of t . In these cases, the orientation of γ is depicted by arrows. Fat dots indicate the possible positions of u and v . Examples (iii) to (vii) each represent an infinite number of cases in which the paths cross any number of times; in all those cases, the orientation and the reachability information is the same.

Proof. Assume without loss of generality that u is on p_L . Assume first that LR is a substring but not a suffix of the sequence, so p_R crosses p_L first from left to right (say, at vertex w_i) and then from right to left (at vertex w_{i+1}). From Fact 1 we learn that $u \prec w_i$ and $w_{i+1} \prec u$ which contradicts the ordering of the w_i . The case RL is analogous. \square

The next lemma is obvious, now that we have split γ into simple curves. We leave the proof to the reader.

Lemma 8 *Let E denote an entrance sequence of length k . Then the k curves $\gamma_1, \dots, \gamma_k$ satisfy:*

1. γ_1 separates s from t iff E begins with an L,
2. γ_i separates s from t for $i = 2, \dots, k - 1$,
3. γ_k separates s from t iff LL or RR is a suffix of E or $E = L$.

Moreover, γ_i is oriented clockwise iff $E_i = L$.

Lemma 9 *There is only one curve if and only if $u \parallel v$. Otherwise, $u \prec v$ if and only if $E_1 = R$ and $p_u = p_R$ or $E_1 = L$ and $p_v = p_R$.*

Proof. If u and v are connected then γ is non-simple from Lemma 3, so the first part of the statement holds. Assume $E_1 = R$ and $p_u = p_R$, so p_u crosses p_v from right to left. From Fact 1 we see that $u \prec w_1$ and $w_1 \prec v$ and are done by transitivity. The other cases are symmetrical. \square

Proof of Lemma 5. The proof is an easy but slightly tedious case analysis on the four different types of entrance sequences. The last two lemmas yield the number of cycles that separate s from t , their orientation and the reachability information. By inspection, all cases are seen to be consistent with Tab. 3. \square

1.4 Algorithm for Reachability

The Meridian We use the results of the last section to construct an algorithm that performs well in the amortised sense, i.e., a sequence of m updates and queries takes time $O(m \log n)$.

As mentioned in the last section, one of the main ideas behind our algorithm is to maintain a chain of faces between the poles, which we will now define.

A *meridian* (F^0, E^0) consists of a sequence of *meridian faces* $F^0 = \langle f_1, \dots, f_m \rangle$ and *meridian edges* $E^0 = \langle e_1, \dots, e_{m-1} \rangle$ such that

1. for $i = 1, \dots, m - 1$, edge e_i is on the boundaries of f_i and f_{i+1} ,
2. $f_i \neq f_j$ for $i \neq j$ (this implies $e_i \neq e_j$).

Moreover, $f_1 = \text{left}(s)$ and $f_m = \text{left}(t)$.

It is easy to see that the meridian corresponds to a *proper* curve μ by viewing the meridian as a path in the dual G^* of G and overlaying the embeddings of G^* and G in a straightforward way. We only have to observe that a path in G^* can never contain a point that embeds a vertex from G . Recall the right half of Figure 8 on page 51 for an example.

How to count wrap-arounds For curves α and β we let $\phi_R(\alpha, \beta)$ denote the number of times α crosses β from right to left. Symmetrically, $\phi_L(\alpha, \beta)$ denotes the number of times α crosses β from left to right.

Note that ϕ_L and ϕ_R have the nice property that if we decompose α into proper curves $\alpha_1, \dots, \alpha_k$ then we have, e.g.,

$$\phi_L(\alpha, \beta) = \sum_{i=1}^k \phi_L(\alpha_i, \beta). \quad (1)$$

If α is a closed curve and β is a proper curve (with respect to α) whose endpoints are on the same region (with respect to α), then β must leave the region bounded by α as often as it enters it, so

$$\phi_L(\alpha, \beta) - \phi_R(\alpha, \beta) = \phi_L(\beta, \alpha) - \phi_R(\beta, \alpha) = 0.$$

These properties are exploited in the proof of the following lemma.

Lemma 10 *The index and the orientation of t are given by the absolute value and the sign of*

$$\phi_R(\mu, p_L) + \phi_L(\mu, p_R) - \phi_L(\mu, p_L) - \phi_R(\mu, p_R),$$

respectively.

Proof. Observe that the meridian connects a point in the region of s , namely $\text{left}(s)$, to a point in the region of t , namely $\text{left}(t)$. Let $\gamma_1, \dots, \gamma_k$, with $k = \text{Ind}(t)$, denote the simple closed subcurves of γ that separate s from t . It is an easy corollary to lemmas 7 and 8 that the curves have the same orientation. Note that the meridian must cross all k curves at least once, but may take a detour: It can go back across a previously crossed curve and return later. Thus the index of t is given by

$$\text{Ind}(t) = \left| \sum_{i=1}^k \phi_R(\mu, \gamma_i) - \phi_L(\mu, \gamma_i) \right|.$$

We can split each γ_i into appropriately indexed subpaths p_L^i and p_R^i of p_L and p_R (and remember to reverse the direction of the latter) to derive

$$\text{Ind}(t) = \left| \sum_{i=1}^k \phi_R(\mu, p_L^i) + \phi_L(\mu, p_R^i) - \phi_L(\mu, p_L^i) - \phi_R(\mu, p_R^i) \right|.$$

All other subpaths of p_L and p_R form a number of closed curves that do not influence $\phi(\mu, \cdot)$, so we can extend the above sum to include all of p_L and p_R without changing the result. This proves the first statement.

For the second statement, observe that the orientation of t is positive if and only if all γ_i are oriented clockwise. In that case, the value of

$$\sum_{i=1}^k \phi_R(\mu, \gamma_i) - \phi_L(\mu, \gamma_i)$$

is negative, else it is positive. Indeed, the expression evaluates to either $\text{Ind}(t)$ or $-\text{Ind}(t)$, depending on the orientation of t . \square

We are ready to state our result. The algorithm performs well in the amortised sense, i.e. for sequences of updates. The next section considers the worst case.

Theorem 3 (Amortised version) *The dynamic reachability problem for planar source-sink graphs can be solved in amortised time $O(\log n)$, where n denotes the number of edges. The data structure uses linear space and can be initialised in linear time.*

Proof. We extend our previous data structure.

We still store the sequences of outgoing and incoming edges around every vertex and the dynamic trees for S and T . Additionally, we maintain the sequences of meridian faces F^0 and edges E^0 under insertion and deletion of subsequences, e.g., using balanced trees.

Also, with every edge e that is in either S or T , we store

$$\phi_R(\mu, e) = \begin{cases} 1, & \text{if } e = e_i \text{ for some } e_i \in E^0 \text{ and } \text{right}(e) = f_i, \\ 0, & \text{otherwise,} \end{cases}$$

which tells us if e is crossed by the meridian from right to left. Symmetrically, we store $\phi_L(\mu, e)$, which can be derived analogously. Using (1) above, we can now in time $O(\log |E|)$ calculate the value of $\phi_R(\mu, p)$ and $\phi_L(\mu, p)$ for every *dynamic path* p of S or T ; see [62] for the details and terminology.

With every face, we keep a topologically ordered sequence of the edges on the two paths that bound the face.

We turn to the operations. For the query operation, we again evert u and v in S and T to find their order around s' and t' . Using Lemma 10 and the data structure above, we find the index and orientation of t . Finally, we refer to Tab. 3 for the answer.

Consider the case where a new edge e is inserted into face f , splitting it into f' and f'' . The edge lists around f' and f'' are easily derived from the edge lists around f . The meridian is unaffected if $f \notin F^0$. Otherwise, one or both of f' and f'' may become part of the updated meridian, depending on where the meridian edges appear around f (we use the edge list around f to decide which case we are in). For example, if there is a meridian edge on both f'

and f'' , they both become part of the meridian and e becomes a new meridian edge. In any case, there are only a constant number of updates to the meridian lists. A straightforward analysis shows that all operations can be performed in logarithmic time, including the updates to the values of ϕ_L and ϕ_R stored in S and T .

Deletions are more involved. Consider the case where deletion of the edge e between faces f' and f'' creates a new face f . Creating the edge list around f is handled as above.

In contrast, the meridian may change drastically. The change occurs when both f' and f'' are meridian faces: We cannot just merge them into one, as that would violate the second condition in the definition of the meridian— put more graphically, the meridian curve μ would no longer be simple.

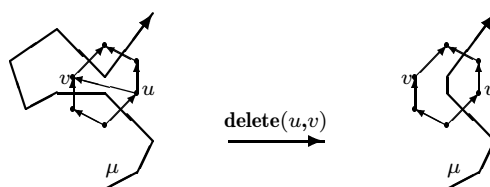


Fig. 10: Deletion of an edge that separates two meridian faces

To remedy this, we must remove everything between f' and f'' from the meridian, as shown in Figure 10. Even though the data structure for the meridian face and edge lists can be updated in logarithmic time, the values $\phi_L(\mu, e)$ and $\phi_R(\mu, e)$ at every removed meridian edge e also have to be changed, which takes time $O(n \log n)$ in the worst case. However, an easy amortisation argument (store a credit with each meridian edge) shows that a sequence of m updates and queries can be executed in time $O(m \log n)$. \square

1.5 Worst case time bounds

Sketch of technique We will now remove the amortisation, a task that involves some rather tedious arguments. We start with a rough sketch: Obviously, the major problem is that we do not have time to remove the meridian cycles arising from a delete operation. However, it is not very hard to believe that such meridian cycles can be shown not to influence the proof of Lemma 10: In a nutshell, whenever a path crosses a such a meridian *cycle*, it must re-cross the same cycle later in the other direction (meridian cycles cannot separate the source from the sink). Hence we choose to let sleeping dogs lie. We do *not* remove the meridian cycles but instead just make sure that they stay cycles as the graph undergoes further changes.

The minor problem left is that this results in more and more meridian cycles as we go, so we use ‘global rebuilding’ [56] to construct an unpolluted data structure in the background.

Now for the details.

False meridians We introduce some more meridians (E^k, F^k) for $r > 0$. To distinguish them from the original meridian (E^0, F^0) , we from now on refer to the latter as the *prime meridian*.

A *false meridian* (F^k, E^k) for $r > 0$ of size l consists of a sequence of faces $F^k = \langle f_1^k, \dots, f_l^k \rangle$ and $E_m^k = \langle e_1^k, \dots, e_l^k \rangle$ such that

1. for $i = 1, \dots, l - 1$, edge e_i^k is on the boundaries of f_i^k and f_{i+1}^k ,
2. edge e_l is on the boundaries of f_l^k and f_1^k ,
3. $f_i^k \neq f_j^k$ for $i \neq j$ (this implies $e_i^k \neq e_j^k$).

Thus the difference between a false meridian and the prime meridian is that the former is cyclic in the sense that the last face is incident to the first. Also, a false meridian need not contain $\text{left}(s)$ nor $\text{left}(t)$. The embedding of a false meridian is a closed proper curve.

Our algorithm will not be able to distinguish false meridians from the prime one. More precisely, when γ crosses a meridian at some point, the algorithm cannot *locally* deduce whether this meridian is the prime meridian or some other. Let us argue that this does not matter.

Denote by μ^k the curves that correspond to false meridians. Since these curves are closed we can use the discussion from § 1.4 to derive

$$\phi_R(\mu^k, \gamma) - \phi_L(\mu^k, \gamma) = 0,$$

for all μ^k . Hence we can add the vanishing term

$$\sum_{k>0} \phi(\mu^k, p_L) + \phi_L(\mu^k, p_R) - \phi_L(\mu^k, p_p) - \phi_R(\mu^k, p_R),$$

where the sum is over all false meridians, to expression (10) without changing the result.

Now that we have seen that the false meridians do not mess up our analysis, let us see that they even make life simpler. We finally arrive at the general statement of this section's result.

Theorem 3 *Dynamic reachability for planar source–sink graphs can be solved in time $O(\log n)$. The data structure uses linear space and can be initialised in linear time.*

Proof. We modify the data structure from the amortised case as follows. With every edge we store the value

$$\sum_{k \geq 0} \phi_R(\mu^k, e), \quad (2)$$

where the sum is over all meridians including the prime. Likewise, we store $\sum \phi_L(\mu_k, e)$.

The two balanced trees for each face that maintain the two sequences of edges around the face are modified so that each internal node computes the sum of the values stored at its children. This allows us to calculate the value

$$\sum_i \sum_k \phi_R(\mu^k, e_i)$$

for each sequence of faces $\langle e_i \rangle$ that appear consecutively around the face in time logarithmic in the length of the sequence. Likewise for ϕ_L .

Note that we do *not* maintain sequences of false meridians (but still maintain the prime meridian). The false meridians appear in the data structure only implicitly in the value from (2) stored at each edge. Let us very briefly sketch how to handle the updates.

We turn to the update operations. Whenever a new edge is inserted into a face that appears on some (possibly false) meridian, we have to update the value from (2). The modified balanced search trees with each face allows us to compute the number of meridians that enter and leave the two new faces. From these values, we can derive the value stored with the new edge consistently with some legal rearrangement of the false meridians.

Whenever an edge deletion induces a cycle in the prime meridian, we remove that cycle from the corresponding list in the data structure as before and make the removed cycle a new false meridian. Figure 11 gives an example.

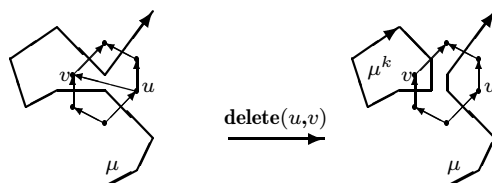


Fig. 11: Edge deletion, worst case

We are almost finished. The only problem is that the number of false meridians is unbounded and hence the values stored with each edge may at some point become exponential.

To avoid this, we use the standard trick of *global rebuilding* (see Chapter 5 of [56]): Construct a new data structure in the background, based on only the prime meridian. After a linear number of operations, the construction has finished and we switch to this new structure (which may already have some false meridians but nevertheless cannot be too large). Now all calculation takes place using the new data structure and we refresh the old data structure in the background. This process of switching data structures is repeated *ad infinitum*. We leave the details with the reader. \square

Note that it is easy to extend the data structure to cope with a *report* operation that outputs a path from u to v if it exists in time $O(\log n + r)$, where r denotes the length of the path. We leave the detail to the reader.

1.6 Relation to previous results

Two partial solutions to our problem are known: Tamassia and Preparata [65] consider *upward* planar source–sink graphs. Tamassia and Tollis [66] extend the result to planar source–sink graphs but restrict the repertoire of update operations to avoid fundamental problems with edge deletion. The present algorithm subsumes and extends the results from these papers in that it removes the restrictions of both. It is only fair to say that for most applications, the algorithm from [65] probably suffices.

We can give an order-theoretic description of the conceptual difference between these algorithms. Both [65] and [66] rely on the well-known fact that the transitive closure of an upward planar source–sink graph can be expressed as the intersection of two *total* orders \leq_L and \leq_R . Symbolically,

$$u \prec v \text{ if and only if } u \leq_L v \wedge u \leq_R v,$$

where we write \prec for the transitive closure. (In other words, upward planar source–sink graphs are the Hasse diagrams of planar lattices.) The first paper shows that in the restricted case, \leq_L and \leq_R are easily maintained as the graph changes. The second paper shows under which updates the orderings remain maintainable in the general case. Kelly [41] has shown that for general planar graphs, the number of total orders needed to express the transitive closure as their intersection is unbounded.

Our approach uses a different characterisation of the transitive closure. We maintain two orders (call them \leq_S and \leq_T for a moment) with the property that

$$u \prec v \text{ if and only if } \exists w \in V : u \leq_T w \wedge w \leq_S v.$$

It is by no means clear that one can handle the existential quantifier over the vertices V of the graph in logarithmic time. Indeed, note that our algorithm is unable to identify such a w , it merely verifies its existence.

Other Classes of Graphs. Italiano *et al.* [37] present a dynamic reachability algorithm for *series parallel* digraphs; apart from these and the class studied in the present paper, no other class of digraphs is known to the author that allows fully dynamic reachability algorithms within polylogarithmic time bounds. The only other nontrivial upper bound is the already cited $O(n^{2/3} \log n)$ for plane graphs from [63] and recent work of Henzinger and King [31].

Other dynamic problems on planar source–sink graphs are studied in [6] and [64]. Reference [65] contains pointers to a vast number of applications of these graphs within visibility representations, graph drawing and embedding, motion planning, computational geometry, lattice theory, and VLSI design.

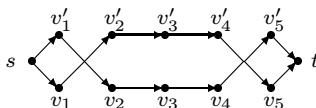
2 LOWER BOUNDS

2.1 Reduction to Prefix Parity

It is well-known how to prove lower bounds for dynamic graph algorithms using a reduction to the prefix parity problem. Let us see an example: an $\Omega(\log n / \log \log n)$ lower bound for dynamic reachability in directed graphs.

Let $x_1, \dots, x_n \in \{0, 1\}$ be an instance of the Dynamic Parity Prefix Problem. Construct the graph $G = (V, E)$ as follows: The vertex set V contains source s and sink t as well as $2n + 2$ vertices v_1, \dots, v_{n+1} and v'_1, \dots, v'_{n+1} . Intuitively,

v_i and v'_i correspond to variable x_i . The edges are constructed from the values of the variables: If x_i is false then E includes the edges (v_i, v_{i+1}) and (v'_i, v'_{i+1}) , else it includes (v_i, v'_{i+1}) and (v'_i, v_{i+1}) . The figure below gives an example for $(x_1, \dots, x_4) = (1, 0, 0, 1)$.



It is not hard to see that we can simulate every update operation to the vector x_1, \dots, x_n using a constant number of insert and delete operations on G without violating its topology. For the query operation, observe that

$$\sum_{i=1}^j x_i = 1 \pmod{2} \quad \text{if and only if} \quad v_1 \prec v'_{j+1}, \quad j = 1, \dots, n.$$

Thus the lower bound on prefix parity implies a lower bound on dynamic reachability.

Note that the construction yields a planar graph: Embed the crossing edges on the back of the sphere and use the stereographic projection for a planar embedding. Hence the bound holds for the easier problem of dynamic reachability on planar graphs. It even holds for the class of planar source–sink graphs that we studied in § 1, proving optimality of Thm. 3 within a factor of $\log \log n$.

Discussion Algorithm designers and lower bound hunters face the dichotomy that many problems have good upper bounds, many problems have good lower bounds, but fewer have both. Consider the problem from the last section: dynamic reachability for planar source–sink graphs, with or without the upward planarity constraint. Prior to the work published in this thesis, the world looked like this:

Class of graphs	Upper bound	Lower bound
<i>Upward</i> planar st-graphs	$O(\log n)$	$\Omega(\log \log n / \log \log \log n)$
Planar st-graphs	$O(n^{2/3} \log n)$	$\Omega(\log n / \log \log n)$

So even though a considerable amount of work lies behind these results ([65, 63, 8, 51]), there is an exponential gap between the bounds. (Theorems 3 and 5 reduce both gaps to logarithmic.)

Our aim will be to provide lower bounds for graph problems for which efficient algorithms actually do exist. The simplicity and similarity of the proofs demonstrates the strength and applicability of the hardness results from Chap. 2.

2.2 Grid Graphs

The vertices of a *grid graph* of width w and height h are integer points (i, j) in the plane for $1 \leq i \leq w$ and $1 \leq j \leq h$. All edges have length 1 and are parallel to the axes. The dynamic reachability problem for these graphs looks like this:

$flip(x, y)$: add an edge between $x \in [w] \times [h]$ and $y \in [w] \times [h]$ or remove it if it exists,

$reachable(x, y)$: return ‘yes’ if and only if there is a path from x to y .

Grid graphs are planar, so standard techniques yield an $O(\log n)$ time algorithm for this problem, where $n = h \cdot w$ denotes the problem size. The next result shows that this is close to optimal:

Theorem 4 *Every implementation of dynamic reachability for grid graphs requires time*

$$\Omega\left(\frac{\log n}{\log n \log n}\right)$$

per operation.

Proof. Let $x \in \{-1, 0, +1\}^n$ be an instance of balanced prefix sum vanishing. We will construct a grid graph of dimension $(2w + 1) \times 2n$ as follows, where the width is given by $w = \lceil \log n / \log \log n \rceil$.

Consider any grid point with odd coordinates $(2i - 1, 2j - 1)$, drawn as \bullet below. It will be connected to one of the three odd grid points above it depending on the value of x_j as follows:

$$x_j = +1 : \begin{array}{c} \circ \\ | \\ \bullet \end{array}, \quad x_j = 0 : \begin{array}{c} \circ \\ | \\ \bullet \end{array}, \quad x_j = -1 : \begin{array}{c} \circ \\ | \\ \bullet \end{array}$$

Of course, at the left and right borders of the graph this rule may be violated because of lack of grid points, the edges in question are simply omitted.

The idea is that the path from $(0, 1)$ mimics the prefix sums of x in that it passes through $(2j, s)$ if and only if $x_1 + \dots + x_j$ equals s . This exploits that the path stays inside the the graph all the time, which is precisely the balancing constraint on x .

It remains to note that the graph can be maintained efficiently. Any changed letter in x incurs $O(w)$ edges to be inserted or deleted. So if the update time of the graph algorithm is polylogarithmic then the graph can be maintained in polylogarithmic time. The bound follows from Prop: 1. \square

Narrow Grid Graphs. The efficient algorithms for grid graphs work for ‘square’ graphs of equal size and width, while the hard graph constructed in the last proof has only logarithmic width $w = O(\log h)$. So we can say that narrow grid graphs are pretty much as hard as square ones. However, this is not true for *very* narrow graphs: It is known that the reachability problem for grid graphs of *constant* width can be solved in time $O(\log \log n)$ per operation (Miltersen and Subramanian, unpublished, using [22]).

This leaves open the question of what happens for graphs of sublogarithmic width. A more subtle statement of the last result gives a lower bound for these graphs that smoothly connects the two extremes.

Theorem 4 (For narrow graphs) *Every dynamic reachability algorithm for grid graphs of width $w = O(\log n / \log \log n)$ takes time $\Omega(w)$ per operation.*

The proof is more or less the same, a direct reduction from signed prefix sum, using the balanced version of Thm. 1 to keep the path within the graph's width.

We conjecture that the complexity of the reachability problem for narrow graphs depends *linearly* on w , as hinted by the lower bound. However, the known upper bounds for constant width grid graphs depend exponentially on w .

2.3 Upward Planar Graphs

We turn to directed graphs, namely to upward planar graphs, for which the lower bound presented in § 2.1 does not seem to work.

The same technique that we used for grid graphs above yields strong lower bounds for this class of graphs. It works even for the source–sink graphs considered in § 1.

Theorem 5 *Every dynamic reachability algorithm for upward planar source–sink graphs takes time*

$$\Omega\left(\frac{\log n}{\log c \log n}\right)$$

per operation.

Proof. From an instance $y \in \{0, \pm 1\}^n$ of signed prefix sum we construct a digraph $G = (V, E)$. The vertex set consists of the source s , the sink t , and $2d + 3$ vertices for each letter y_i :

$$V = \{v_{ij} \mid 1 \leq i \leq n + 1, -d - 1 \leq j \leq d + 1\}.$$

The i th row is connected to its upper neighbour according to the value of y_i :

$$\begin{aligned} & \{(v_{ij}, v_{(i+1)j'}) \mid 1 \leq i \leq n, -d - 1 \leq j \leq d + 1\}, \\ & \text{where } j' = \begin{cases} j + y_i, & \text{if } |j + y_i| \leq d + 1, \\ d + 1, & \text{if } j + y_i = d + 2, \\ -d - 1, & \text{if } j + y_i = -d - 2. \end{cases} \end{aligned} \quad (1)$$

Figure 12(b) gives an example. Note how the path starting in $(1, 0)$ (the middle vertex in the bottom row) mimics $s_i = \sum_{j=1}^i x_j$. Indeed, there is a path from $(1, 0)$ to $(i + 1, u)$ for $1 \leq i \leq n$ and $-d \leq u \leq d$ if and only if $s_i = u$. We are going to use the reachability query to detect this.

First, we finish the construction by adding some more edges that have only technical significance and make sure that G is an st -graph. At the ends of the graph, $2m + 3$ edges connect s to the bottom row and $2m + 1$ edges connect the topmost row to t ,

$$\{(s, v_{1j}), (v_{j(n+1)}, t) \mid -m - 1 \leq j \leq m + 1\}.$$

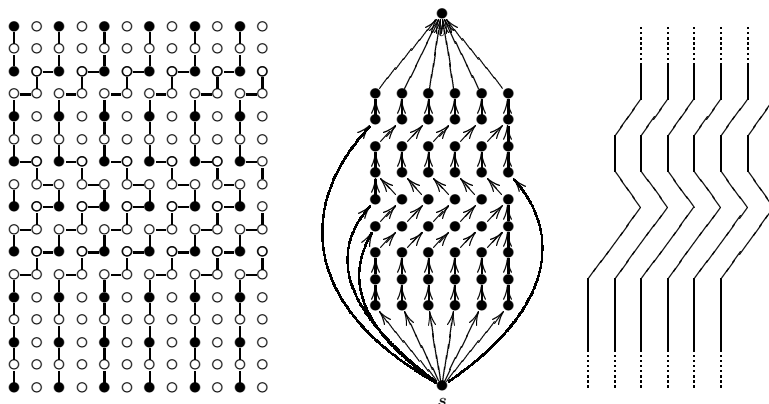


Fig. 12: Planar graphs corresponding to $x = (0, 0, +1, +1, -1, 0, +1, 0)$. Left: grid graph. Odd grid points are marked \bullet , even grid points are marked \circ . Middle: upward planar source-sink graph. Right: monotone planar subdivision.

At the top- and bottommost rows, edges connect s to all vertices that would otherwise be sources:

$$\{(s, v_{i(m+1)}) \mid y_{i-1} = -1\} \cup \{(s, v_{i(-m-1)}) \mid y_{i-1} = 1\}.$$

Again, this graph can be maintained in polylogarithmic time as x changes. And as before, there is a path from v_{10} to v_{1i} if and only if the i th prefix sum of x vanishes. \square

It is not hard to see that the same bound holds for dynamic *upward planarity testing* using essentially the same proof (check for planarity of with a new edge from v_{10} to v_{1i} for the query).

2.4 Planar Point Location

We (pretend to) leave dynamic graph algorithms and turn to computational geometry to see yet another application of our technique.

A classical problem in that field is *planar point location*: given a subdivision of the plane, i.e., a partition into polygonal regions induced by the straight-line embedding of a planar graph, determine the region of query point $q \in \mathbb{R}^2$.

In the dynamic version, updates consist of insertion and deletion of vertices or (chains of) edges. An important restriction of the problem, for which our bound will apply, considers only *monotone* subdivisions, where the subdivision consists of polygons that are monotone (so no horizontal line crosses any polygon more than twice). Preparata and Tamassia [59] give an algorithm that runs in time $O(\log^2 n)$ per operation, this was improved to query time $O(\log n)$ by Baumgarten, Jung, and Mehlhorn [7].

The literature does not quite agree on the exact choice of operations, since the representation of the polygons defines what updates are feasible. Our lower

bound does not depend on these choices, but we have to make one for concreteness:

$move(p, q)$: move polygon corner p to q , provided the subdivision remains monotone,

$query(x)$: return ‘yes’ if and only if x is in the same polygon as the origin.

Our operations are very weak, since we want to prove a useful lower bound. Efficient algorithms are known for more powerful operations that return the name of a queried polygon, and insert and delete (chains of) edges, see [7, 14, 59].

Theorem 6 *Every algorithm for dynamic planar point location in monotone subdivisions uses $\Omega(\log n / \log \log n)$ steps per operation.*

Proof. To prove a lower bound for this problem we construct a monotone subdivision from the instance $x \in \{0, \pm 1\}^n$ that is similar to the upward planar graph from before. This is easier drawn than explained formally; see Figure fig: subdivision. There are 2 unbounded polygons at the sides and $d + 1$ strip-like ones with common sides and common top and bottom corners at infinity. Each of the strip-like polygons mimics the path described by x with $+1$ meaning ‘go right’ and -1 meaning ‘go left.’

To answer a sum query for the i th prefix, we simply check if the point $(0, i)$ lies in the left unbounded polygon or not. \square

CHAPTER 5

Strings

The endeavour is . . . to provide a wide variety of brackets. In regular use we have (in printer’s language) the three sorts: parentheses $()$, braces $\{\}$, and brackets $[]$; and their normal order is $\{\{(\dots)\}\}$. When necessary they can be extended to ‘full face’ brackets and parentheses $\llbracket, ()$. . . We then have the extended set of $\llbracket(\{\{(\dots)\}\})\rrbracket$.

Two further sorts of bracket can be made available on the keyboard: ‘double’ brackets $\llbracket\rrbracket$ and ‘angular’ brackets $\langle\rangle$. Double brackets can be placed outside the bracket sequence as

$$\llbracket(\{\{(\dots)\}\})\rrbracket.$$

—Chaundy et al., *The Printing of Mathematics*

This author believes that they *should be left in whatever order and variety the author has indicated in the manuscript*. It is always desirable, however, to *count* the pairs because it is not uncommon for an author to leave out the closure of a parenthesis or a bracket in error.

—Ellen E. Swanson, *Mathematics into Type*

Our last chapter is about strings, an area that has received far less attention than dynamic graphs algorithms. This is strange because interactive manipulation of strings—in modern text processors or editors—poses several challenging and interesting computational problems. While strings are well studied objects in other areas of combinatorial algorithms, they have been largely ignored by dynamic algorithms designers.

We will study the language membership problem for the *Dyck languages*, the class of strings of properly balanced brackets, and obtain efficient algorithms for maintaining balance in a string of brackets. Again, we will apply the results of Chap. 2 to prove strong lower bounds.

1 DYCK LANGUAGES

The language of *properly balanced brackets* contains strings like $()$ and $()(())$ but not $)$. The notion of balancedness also makes sense if we add more types of brackets: $(\llbracket)\llbracket$ balances but \llbracket does not.

More formally, let $A = \{a_1, \dots, a_k\}$ and $\bar{A} = \{\bar{a}_1, \dots, \bar{a}_k\}$ be two disjoint sets of opening and closing symbols, respectively. For example, the pair $A = \{(\text{, } [, \text{ do, if}\}$ and $\bar{A} = \{),], \text{ od, fi}\}$ captures the nested structure of programming languages. The *one-sided Dyck language* D_k over $A \cup \bar{A}$ is the context-free language generated by the following grammar:

$$S \rightarrow SS \mid a_1 S \bar{a}_1 \mid \dots \mid a_k S \bar{a}_k \mid \epsilon.$$

Closely related is the *two-sided Dyck language* D'_k over $A \cup \bar{A}$ defined by

$$S \rightarrow SS \mid a_1 S \bar{a}_1 \mid \bar{a}_1 S a_1 \mid \dots \mid a_k S \bar{a}_k \mid \bar{a}_k S a_k \mid \epsilon.$$

This corresponds to two-sided cancellation, so now also $)$ (and $(|)$ balance, while $]$ still does not.

The two-sided Dyck language has an algebraic interpretation. If we identify \bar{a}_i with a_i^{-1} and view concatenation as the product operator then x is in D'_k if and only if it equals the identity in the free group generated by A . For example, $\bar{a}_1 a_2 \bar{a}_2 a_1 \in D'_2$ because $a_1^{-1} a_2 a_2^{-1} a_1$ reduces to unity.

The Dynamic Membership problem We consider the problem of maintaining membership in D_k or D'_k of a string from $(A \cup \bar{A})^n$ dynamically. More precisely, we want to maintain a string $x \in (A \cup \bar{A})^n$ of even length, initially a_1^n , with the following operations for any Dyck language D :

change(i, a): change x_i to $a \in A \cup \bar{A}$,

member: return ‘yes’ if and only if $x \in D_k$.

We can use this set of updates to analyse the *word* problem for the free group. Here, the *member* query returns ‘yes’ if and only if $\prod_i x_i = 1$. In this context, *product* or *identity* may be better names for the query. However, we will refrain from distinguishing between the word problem for the free group and the membership problem for two-sided Dyck languages to keep the exposition simple.

Much of the *practical* motivation for the present work stems from modern editors, many of which have editing modes for specific programming languages where a rudimentary on-line syntax check is performed whenever the source is changed. We would like to know whether such a check can be performed faster than in the straightforward way. To model this more closely, we replace the *change* operation with the following:

insert(i, a): insert symbol a between x_{i-1} and x_i ,

delete(i): remove the i th symbol of x .

We can simulate a *change* update by one insertion and deletion, so this is clearly harder.

Other *queries* are interesting, too. From the theorists point of view, the word problem must be studied under the prefix query:

prefix(i): return ‘yes’ if and only if $\prod_{j=1}^i x_j$ is the identity, (or, equivalently, if $x_1 \cdots x_i \in D$.)

while any useful editor ought to be able to answer queries like

interval(i, j): return ‘yes’ if and only if $x_i \cdots x_j$ is in D ,

match(i): return the index of the bracket matching x_i ,

mismatch: return the index of the first unmatched bracket.

(The last two operations do not make sense in the two-sided case.)

Readers discouraged by this plethora of operations may rest assured that most of our results are independent of which operations we choose. We state explicitly when this is not the case.

Of course, the Dyck languages do not capture all aspects of the far more complicated grammar of real programming languages. Ultimately, we could wish for a fast dynamic algorithm for recognition of (a large subclass of) the deterministic context-free languages, which would allow us to implement on-line syntax checking in an editor. Hopefully, our results are a step in the right direction. We do not expect them to be particularly useful in practice as is, however. Even though they run in polylogarithmic time (and the hidden constants are of moderate size), one should keep in mind that the original, sequential algorithm is extremely simple and probably outperforms them in normal applications. Although extremely long files do arise in practice, problems of quite a different nature—like paging, network access, etc.—would dwarf the execution time of both a dynamic and a sequential algorithm for bracket matching.

Results Our main upper bound result is that the dynamic membership problem for all Dyck languages can be solved in polylogarithmic time per operation, the exact bound is $O(\log^3 n \log^* n)$. We use a technique for maintaining dynamic sequences under equality tests by Mehlhorn, Sundar, and Uhrig [47], which also gives (Las Vegas-style) randomised algorithms that run in slightly better expected time: $O(\log^3 n)$. We achieve better bounds for Monte Carlo-style algorithms. Using the fingerprint method of Karp and Rabin [40], where strings are represented by (non-unique) fingerprints in the form of a matrix product modulo a small randomly chosen prime, D_k can be done in time $O(\log^2 n)$ and D'_k in time $O(\log n)$. For D_1 and D'_1 we can use simpler techniques to achieve better bounds.

See Tab. 4 for an overview. Note that except for the $O(1)$ algorithm for D'_1 , all algorithms are also valid (and have the same complexity) when we extend the operations to insertion and deletion of single characters, interval queries, and (for the one-sided case) match queries.

A lower bound of $\Omega(\log n / \log \log n)$ holds for Dyck languages with two or more letters and with the restricted set of operations (*change*, *member*). The same bound holds for one-letter Dyck languages with *interval* queries. The proof builds on the hardness results from Chap. 2. The bounds holds for the membership problem of any Dyck language if we allow *insert* and *delete*. Using a technique from [49, 8], we can show a lower bound for restricted updates for D_1 of $\Omega(\log \log n / \log \log \log n)$. This separates the complexities of the dynamic membership problem for D_1 and D'_1 .

Problem	Lower bound	Upper bound	
With <i>change</i> updates:			
D_1	$\Omega\left(\frac{\log \log n}{\log \log \log n}\right)$	$O(\log n)$	
D'_1		$\Theta(1)$	
$D_k(k > 1)$	$\left. \vphantom{\begin{matrix} D_k(k > 1) \\ D'_k(k > 1) \end{matrix}} \right\} \Omega\left(\frac{\log n}{\log \log n}\right)$	Deterministic	M ^{te} Carlo
$D'_k(k > 1)$		$O(\log^3 n \log^* n)$	$O(\log^2 n)$
With <i>insert/delete</i> updates:			
D'_1		$\Theta\left(\frac{\log n}{\log \log n}\right)$	
any other	$\Omega\left(\frac{\log n}{\log \log n}\right)$	As with <i>change</i> .	
Bit probe model with <i>change</i> updates:			
D_1	$\Omega\left(\frac{\log n}{\log \log n}\right)$	$O(\log n \log \log n)$	
D'_1		$\Theta(\log \log n)$	

Tab. 4: Results for the membership problem for the Dyck languages.

The table also lists results for cell size 1 (the *bit probe model*) from [21], but we have not included them in this thesis because both upper and lower bounds are of a different flavour than in the rest of the text.

Related results It is interesting that all Dyck languages seem to be equally hard in most non-dynamic computational models. Ritchie and Springsteel [61] showed that the one-sided Dyck languages are in deterministic Logspace, Lipton and Zalcstein [44] extended this to the two-sided case (see also [30, Exercises 22 and 23]). One can phrase this even stronger in terms of circuit complexity: all Dyck languages are complete for TC^0 under AC^0 -reductions (this appears to be folklore).

Dynamic Word and Prefix problems for *finite* monoids are studied in [22, 49]. The free group of k generators studied in the present paper is infinite.

Turning from context-free to regular languages, it is easy to find logarithmic time algorithms for the Dynamic Membership problem for the latter class. The results from [22] give better upper bounds depending on the language's syntactic monoid $M(L)$.

Immerman and Patnaik [58] consider dynamic algorithms for the Dyck languages and construct algorithms with update and query operations in dyn-FO, but no efficient sequential upper bounds follow from this.

Preliminaries. For letter a and string u , we put

$$|u|_a = |\{i \mid u_i = a\}|,$$

the number of occurrences of a in u .

We call a string *reduced* if it contains no neighbouring pair of matching brackets. So, for the one-sided case, $([])$ is not reduced but $[])$ is. In the two-sided case, the latter is not reduced. To formalise this (following Harrison [30]), we introduce two mappings

$$\mu_1, \mu_2: (A \cup \bar{A})^* \rightarrow (A \cup \bar{A})^*.$$

We want $\mu_1(u)$ and $\mu_2(u)$ to be the reduced form of u using one- and two-sided cancellation, respectively. To this end we define for each $1 \leq i \leq k$ and $j = 1, 2$:

$$\begin{aligned} \mu_1(\epsilon) &= \mu_2(\epsilon) = \epsilon, & \mu_1(ua_i) &= \mu_1(u)a_i, \\ \mu_2(ua_i) &= \begin{cases} \mu_2(u)a_i, & \text{if } \mu_2(u) \notin (A \cup \bar{A})^*\bar{a}_i, \\ u', & \text{if } \mu_2(u) = u'\bar{a}_i, \end{cases} \\ \mu_j(u\bar{a}_i) &= \begin{cases} \mu_j(u)\bar{a}_i, & \text{if } \mu_j(u) \notin (A \cup \bar{A})^*a_i, \\ u', & \text{if } \mu_j(u) = u'a_i. \end{cases} \end{aligned}$$

One can show properties like $\mu_1(ua_i\bar{a}_i v) = \mu_1(uv)$ and $\mu_2(ua_i\bar{a}_i v) = \mu_2(u\bar{a}_i a_i v) = \mu_2(uv)$. We formally define u^{-1} as $\bar{u}_n \cdots \bar{u}_1$ with the convention $\bar{a} = a$ and $\epsilon^{-1} = \epsilon$.

Mathematics about brackets can be confusing, so we sometimes frame the 'formal language brackets' as \square to distinguish them from brackets that are used as signs of aggregation.

Roots. The Dyck languages bear the name of the German mathematician Walther von Dyck (1856–1934), who studied finitely generated free groups [68]. They appeared in computer science as formal languages in the work of Chomsky and Schützenberger, who coined their name.

2 ALGORITHMS

All algorithms in this section work on the random access machine with logarithmic cell size and unit cost instructions.

2.1 One Type of Brackets

We begin with two easy upper bounds for D_1 and D'_1 , respectively.

Proposition 6 *The Dynamic Membership problem for D'_1 can be solved in constant time. With the extended set of updates, the complexity raises to $\Theta(\log n / \log \log n)$.*

Proof. We focus on *change* and *member*. Note first that for all $x \in \{\llbracket, \rrbracket\}^*$ we have

$$x \in D'_1 \iff |x|_{\llbracket} = |x|_{\rrbracket}.$$

The only if direction is obvious. The other follows from the fact that a reduced string over $\{\llbracket, \rrbracket\}^*$ cannot contain both \llbracket and \rrbracket . Hence we only need to count the number of occurrences of \llbracket and \rrbracket in x to answer membership queries.

With change operations, we merely need to store x and increment or decrement a counter accordingly to the update, which can be done in constant time per update.

The solution can *not* be used for the extended set of operations, since we cannot keep track of the indices that fast (a lower bound for *list indexing* is given in [25]), and [16] presents an $O(\log n / \log \log n)$ algorithm. Details are given in [21]. \square

Proposition 7 *The dynamic membership problem for D_1 can be solved in time $O(\log n)$ per operation.*

Proof. First note that for any $x \in \{\llbracket, \rrbracket\}^*$, the reduced string $\mu_1(x)$ is of the form $\rrbracket^r \llbracket^l$ for integers $l, r \geq 0$. We can view l and r as the number of excessive left and right brackets, respectively.

We maintain a balanced binary tree whose i th leaf represents x_i and where each internal node represents the concatenation of its children's strings. With each node we store the tuple (r, l) describing the reduced form of the string it represents.

For the operations first note that $x \in D_1$ if and only if the root contains the tuple $(0, 0)$, corresponding to $\mu_1(x) = \epsilon$. To handle the updates it suffices

to note that the value of a node can be easily derived from the values of its children, since

$$\mu_1(\lceil^{r_1} \lfloor^{l_1} \lceil^{r_2} \lfloor^{l_2}) = \begin{cases} \lceil^{r_1} \lfloor^{l_2+l_1-r_2}, & \text{if } l_1 \geq r_2, \\ \lceil^{r_1+r_2-l_1} \lfloor^{l_2}, & \text{otherwise.} \end{cases}$$

We can redo these calculations bluntly at each level and achieve a running time proportional to the height of the tree.

The data structure is easily generalised to the extended set of operations using any scheme for balancing dynamic search trees, e.g. *red-black trees* [29].

□

We will not comment on such extensions any further; the reader can check that they are also possible for all the algorithms in Sections 3 and 4. The algorithm in the proof of Proposition 9 calls for the most complicated extensions, in that we also need to be able to split and merge trees.

2.2 Many Types of Brackets

We move now to the main result, extending the above to larger k . The basic idea resembles very much the data structure from Proposition 7: we represent x as a balanced binary tree whose internal nodes correspond to substrings of x . At each node, we store entire sequences (rather than just a tuple as above) that are formed from the sequences stored at its children. To this end we first need a recent surprising construction for dynamically maintaining sequences.

A data structure for string equality Mehlhorn, Sundar, and Uhrig [47] present a data structure for dynamically maintaining a family of strings under equality tests. We use a slightly modified set of updates that is better suited to our problem. More precisely, we want to maintain an initially empty family S of strings from a finite alphabet Σ under the following operations:

create(σ): create a new (one-letter) string $s = \sigma \in \Sigma$ and add it to S ,

concatenate(s, s'): create a new string $s'' = ss'$ and add it to S ,

split(s, i): create new strings $s' = s_1 \cdots s_i$ and $s'' = s_{i+1} \cdots s_n$, and add them to S ,

equal(s, s'): return ‘yes’ if and only if $s = s'$,

lcp(s, s'): return the length of the longest common prefix of s and s' .

The time bounds for these operations are summarised in the following lemma.

Lemma 11 ([47]) *Any of the above operations takes time $O(\log n(\log m \log^* m + \log n))$ on a unit-cost RAM with cell size $O(\log(n+m))$, where m is the number of operations executed for the string family S so far, and n is the total length of involved strings in the operation.*

In our setting we do not allow the time complexity and cell size to depend on m . To get around this problem we use the global rebuilding technique of Overmars [56] to keep the size of m linear relative to the input size of the problems we consider.

The time bounds in the lemma are all explicitly stated in [47] except for the *lcp* operation. For completeness, we show how to implement this operation within the claimed time bound.

Let s and s' be two strings in the string family S for which we want to find the length of the longest common prefix. We adopt the notation of [47] letting $\bar{s} = (\tau_0, \tau_1, \dots, \tau_{2t})$ and $\bar{s}' = (\tau'_0, \tau'_1, \dots, \tau'_{2t'})$, with $\tau_0 = s$, $\tau'_0 = s'$ and $\tau_{2i-1} = \text{elpow}(\tau_{2i-2})$, $\tau_{2i} = \text{shrink}(\tau_{2i-2})$, $\tau'_{2i-1} = \text{elpow}(\tau'_{2i-2})$ and $\tau'_{2i} = \text{shrink}(\tau'_{2i-2})$ for all $i \geq 1$.

Let $N(\tau_i, p)$ denote the value of the p th node of (tree that represents the) list τ_i . The longest common prefix operation may now be implemented as follows:

```

proc lcp( $s, s'$ )
(1) find  $i$  such that  $N(\tau_i, 1) \neq N(\tau'_i, 1)$ 
     $p \leftarrow 1$ 
    while  $i > 0$  do  $\{I(i, p)\}$ 
        if  $i$  is even
(2)     $q \leftarrow$  the position of the rightmost node of  $T_{i-1}$ 
        encoded by  $N(\tau_i, p - 1)$ 
        elseif  $i$  is odd
        write the blocks at  $N(\tau_i, p)$  and  $N(\tau'_i, p)$ 
        as  $\sigma^k$  and  $\rho^l$ , respectively
        if  $\sigma \neq \rho$ 
(3)     $q \leftarrow$  the position of the rightmost node of  $T_{i-1}$ 
        encoded by  $N(\tau_i, p - 1)$ 
        else
(4)     $m \leftarrow \min\{k, l\}$ 
         $q \leftarrow$  the position of the node of  $T_{i-1}$  encoded by
        the  $m$ th  $\sigma$  of  $N(\tau_i, p) = \sigma^k$ 
        fi
    fi
(5)    increment  $q$  until  $N(\tau_{i-1}, q) \neq N(\tau'_{i-1}, q)$ 
         $p \leftarrow q$ 
         $i \leftarrow i - 1$ 
    od
    return  $p - 1$ 
end

```

It is easy to verify that the invariant $I(i, p)$ defined as

$$N(\tau_i, p) \neq N(\tau'_i, p), \quad \text{but } N(\tau_i, q) = N(\tau'_i, q) \text{ for all } q = \{1, \dots, p - 1\},$$

holds inside the while-loop. Hence after the loop, $p - 1$ contains the length of the longest common prefix of s and s' .

The number of iterations of the while loop is $O(\log n)$. The values of the assignments (1), (2), (3) and (4) can be found in time $O(\log n)$ (of course, if the search (1) fails for all i , the two strings are equal). By lemma 4.2 in [47] the number of increments in line (5) is at most $O(1)$. Thus in total the *lcp* operation takes time $O(\log^2 n)$ as claimed. A similar argument shows that the expected time bound for the randomised version is $O(\log^2 n)$ too.

The two-sided case

Theorem 7 (Two-sided) *The dynamic membership problem for D'_k can be done in solved $O(\log^3 n \log^* n)$ per operation.*

Proof. We maintain a balanced binary tree whose i th leaf represents x_i and where each internal node represents the concatenation of its children's strings. With the node representing (say) y we store $\mu_2(y)$ and $\mu_2(y^{-1})$.

Let us see how we handle the operations. The query operation is easy since the root contains $\mu_2(x)$. For the change operation, we will show how to use the data structure from Lem. 11 to maintain the two sequences at each node. First note that the leaves of the tree are easily changed because $\mu_2(x_i) = x_i$ and $\mu_2(x_i^{-1}) = \bar{x}_i$.

From the leaf, the change propagates towards the root of the tree. To handle the changes at an internal node we exploit a useful property of the reduction function μ_2 : given $u, v \in (A \cup \bar{A})^*$, write

$$\mu_2(u) = u'aw \quad \text{and} \quad \mu_2(v) = w^{-1}bv', \quad \text{with } \bar{a} \neq b, \quad (1)$$

for some $u', v', w \in (A \cup \bar{A})^*$ and $a, b \in (A \cup \bar{A})$. Then one can show

$$\mu_2(uv) = u'abv'. \quad (2)$$

Consider for concreteness an internal node whose children represent (say) u and v , respectively. Let w denote the longest common prefix of $\mu_2(u^{-1})$ and $\mu_2(v)$, which can be found from the information at the children of the node in time $O(\log^2 N \log^* N)$, where N denotes the total length of all sequences in the tree. Now split $\mu_2(u)$ and $\mu_2(v)$ as in (1) above and construct $\mu_2(uv)$ by (2), using a constant number of operations, each of which takes time $O(\log n(\log m \log^* m + \log n))$ by Lemma 11. Employing the global rebuilding technique as mentioned in § 2.2 we ensure that $m \in O(n)$. We only need to observe that the above data structure can be rebuilt from scratch using at most $O(n)$ of the operations from the MSU data structure. We conclude that the total amount of time for an update is $O(\log^3 n \log^* n)$. \square

The one-sided case The proof for D_k is similar to that for D'_k but marred by the less nice algebraic properties of μ_1 .

Theorem 7 (One-sided) *The dynamic membership problem for D_k can be solved in time $O(\log^3 n \log^* n)$ per operation.*

Proof. As before, we maintain a balanced binary tree whose i th leaf represents x_i and where each internal node represents the concatenation of its children's strings.

We define yet another cancellation function μ , where every left bracket cancels every right bracket, regardless of its type, by

$$\begin{aligned} \mu(\epsilon) &= \epsilon, & \mu(ua_i) &= \mu(u)a_i, \\ \mu(u\bar{a}_i) &= \begin{cases} \mu(u)\bar{a}_i, & \text{if } \mu(u) \notin (A \cup \bar{A})^*A, \\ u', & \text{if } \mu(u) \in u'A. \end{cases} \end{aligned}$$

For every y we can write $\mu(y)$ as $y_{\bar{A}}y_A$ for some $y_{\bar{A}} \in \bar{A}^*$ and $y_A \in A^*$. With the tree node for y we store a bit that is true if and only if $\mu_1(y) \in \bar{A}^*A^*$ (equivalently, $\mu(y) = \mu_1(y)$), as well as the strings y_A and $y_{\bar{A}}$, and their formal inverses $(y_A)^{-1}$ and $(y_{\bar{A}})^{-1}$. The intuition is that if this bit is false then x cannot balance, since

$$\mu_1(y) \in \bar{A}^*A^* \quad \text{if and only if} \quad \exists u, v: uyv \in D_k, \quad (3)$$

and then it suffices to store that information only. In the other case, $\mu_1(y)$ consists only of $y_{\bar{A}}$ and y_A , and these two strings (together with their formal inverses) are easily maintained, as we shall see below.

We turn to the operations. First note that membership of x in D_k can be read off the root node, since

$$x \in D_k \quad \text{if and only if} \quad \mu_1(x) \in \bar{A}^*A^* \quad \text{and} \quad x_A = x_{\bar{A}} = \epsilon.$$

For the updates it suffices to explain how we can derive the information at a node from its children using a constant number of string operations. Let u and v denote the strings represented by the node's children and assume without loss of generality $|u_A| \geq |v_{\bar{A}}|$ (the other case is symmetrical). Write u_A as $u_{A,1}u_{A,2}$, where $|u_{A,2}| = |v_{\bar{A}}|$. Then $y_A = u_{A,1}v_A$ and $y_{\bar{A}} = u_{\bar{A}}$. Moreover,

$$\mu_1(y) \in \bar{A}^*A^* \quad \text{if and only if} \quad \mu_1(u), \mu_1(v) \in \bar{A}^*A^* \quad \text{and} \quad u_{A,2} = (v_{\bar{A}})^{-1}.$$

The formal inverses $(y_A)^{-1}$ and $(y_{\bar{A}})^{-1}$ are easily maintained. This completes the proof. \square

The upper bounds from the last two propositions can be improved to expected time $O(\log^3 n)$ using the Las Vegas variant of the algorithm described in § 2.2, see [47].

2.3 Monte Carlo Algorithms

The two-sided case We begin with D'_k , which is quite simple. We use the well-known *fingerprint* string matching technique of Karp and Rabin [40].

Proposition 8 *The dynamic membership problem for D'_k can be solved in time $O(\log n)$ per operation such that the probability of an erroneous answer in any sequence of n updates is $O(\frac{1}{n})$.*

Proof. We start by considering D'_2 over the alphabet $A = \{\square, \blacksquare\}$ and $\bar{A} = \{\square, \blacksquare\}$. Define the congruence \sim by

$$u \sim v \quad \text{if and only if} \quad \mu_2(u) = \mu_2(v).$$

Then the quotient $(A \cup \bar{A})^* / \sim$ is a group (*the free group over $\{\square, \blacksquare\}$*) with concatenation as the operator and ϵ as the identity.

Following Lipton and Zalcstein [44] (see also [45, Problem 2.3.13]), we represent $(A \cup \bar{A})^*/\sim$ as a group of 2×2 integer matrices using the group homomorphism

$$h: (A \cup \bar{A})^*/\sim \rightarrow M_2(\mathbb{Z}),$$

$$h(\mathbb{I}) = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad h(\bar{\mathbb{I}}) = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}.$$

In this terminology, x is in D'_2 if and only if $h(x)$ is the identity matrix.

This suggests a randomised algorithm in the spirit of [40]: compute $h(x)$ modulo a randomly chosen prime p and check whether the result is the identity matrix.

For the dynamic version we need to maintain $h(x) \bmod p$ under updates to x , we write n for $|x|$. For a fixed prime $p \leq n^4$ we can recompute $h(x) \bmod p$ in logarithmic time using a balanced binary tree, where the i th leaf contains $h(x_i)$ and an internal node contains the product (in $M_2(\mathbb{Z}_p)$) of the value of its children. Thus the root contains $h(x) \bmod p$.

To bound the probability of error we note that all entries in the matrix $h(x)$ are bounded by 3^n , so there can be at most n distinct primes p such that $h(x) \equiv 1 \pmod p$ if in fact $h(x) \neq 1$. Choosing $p \leq n^4$ randomly and choosing a new p for every n operations by the *global rebuilding* technique of Overmars [56] we guarantee that the probability of an erroneous answer in a sequence of n consecutive queries is bounded by $O(\frac{1}{n})$.

The above construction can be extended to larger k using the fact that the free group on k generators is a subgroup of the free group on two generators g_1, g_2 . Indeed, if for $1 \leq i \leq k$ we put $c_i = g_1^i g_2^i$ then c_1, \dots, c_k generate a free group, see [45, Problem 1.4.12]. \square

The one-sided case The algorithm for D_k is somewhat more difficult. We will combine the tree-structure we used for the deterministic algorithm for D_k (Proposition 7) with the Monte Carlo algorithm for D'_k from the last proposition. Recall that in the deterministic algorithm, we use the expensive string operations from § 2.2 to test whether certain internal substrings (namely, $u_{A,2}$ and $(v_{\bar{A}})^{-1}$) constitute a match. But since $u_{A,2} \in A^*$ and $v_{\bar{A}} \in \bar{A}^*$, this is true if and only if $u_{A,2}v_{\bar{A}} \in D'_k$, so we can use the much faster Monte Carlo algorithm for D'_k instead.

Proposition 9 *The dynamic membership problem for D_k can be solved in time $O(\log^2 n)$ per operation such that the probability of an erroneous answer in any sequence of n updates is $O(\frac{1}{n})$.*

Proof. As before, we maintain a balanced binary tree whose i th leaf represents x_i and where each internal node represents the concatenation of its children's strings.

For every y we define $y_A, y_{\bar{A}}, u, v, u_A = u_{A,1}u_{A,2}$, and $v_{\bar{A}}$ as in the proof of Proposition 7. In particular, we assume $|u_A| \geq |v_{\bar{A}}|$ (the other case is symmetrical). Write $w = u_{A,2}v_{\bar{A}}$. With the tree node for y (of length m , say) we maintain the following information:

1. a bit that is true if and only if $\mu_1(y) \in \bar{A}^*A^*$,
2. three balanced binary search trees whose leaves store the indices (in x) of $y_A, y_{\bar{A}}$, and w , respectively,
3. the lengths $|y_A|, |y_{\bar{A}}|$, and $|w|$,
4. a string $w_{\#} \in (A \cup \bar{A} \cup \{\#\})^m$ defined as follows: since w is a subsequence of y , we can write $w = y_{i_1} \dots y_{i_l}$ for some $i_1 < \dots < i_l$. Then we define

$$w_{\#} = \#^{i_1-1}y_{i_1}\#^{i_2-i_1-1}y_{i_2}\#^{i_3-i_2-1}y_{i_3} \dots \#^{i_l-i_{l-1}-1}y_{i_l}\#^{m-i_l}.$$

One can view this as a padded w of fixed length.

Note that we do not store y itself.

Turning to the operations, we first note that the query is handled as in the proof of Proposition 7. A tedious case analysis shows that when a single letter of y is changed then at most two changes are induced in each of y_A and $y_{\bar{A}}$ and at most four changes in w and $w_{\#}$. The corresponding updates at the node representing y can be done in time $O(\log n)$ given knowledge about the updates at lower levels.

To see whether $w \in D'_k$, we apply the technique from the last proposition, using $w_{\#}$ as instance; the extra letter $\#$ is handled by letting h map it to the identity matrix. Hence we can maintain the information at each level of the tree in time $O(\log n)$, from which the stated time bound follows.

To bound the error probability, note that we use $O(n)$ distinct versions of the data structure from Proposition 8. Using a prime from a larger set (say, $p \leq n^5$), we obtain the stated bound. \square

3 LOWER BOUNDS

We return to the results of Chap. 2 to prove lower bounds for our problem.

3.1 Dynamic Membership

For ease of notation we will slightly change the setting. We add a third letter to our alphabet, a blank \square that does not affect language membership. Formally we look at the language D' of strings over $\{\square, \square, \square\}$ that are in the Dyck language D if the blanks are removed. This language is not harder than the Dyck language because of the encoding

$$\square \mapsto \square \square, \quad \square \mapsto \square \square, \quad \square \mapsto \square \square,$$

but much easier to reason about.

We first consider the interval problem where the query $balance(i, j)$ returns ‘yes’ if and only if $x_i \dots x_j$ is properly balanced.

Lemma 12 *Every algorithm for the interval problem of a Dyck language requires time $\Omega(\log n / \log \log n)$.*

Proof. The proof for D'_1 follows directly from Prop. 1 with the encoding

$$+1 \mapsto \llbracket, \quad -1 \mapsto \rrbracket, \quad 0 \mapsto \square.$$

The proof for the one-sided Dyck language D_1 is almost as easy. Encode

$$+1 \mapsto \llbracket \llbracket, \quad -1 \mapsto \rrbracket \rrbracket, \quad 0 \mapsto \llbracket \square.$$

and pad the string to the left with $2n$ opening brackets¹. Call this string y and index it symmetrically around the middle as $y = y_{-2n}, \dots, y_{-1}, y_1, \dots, y_{2n}$. The i th prefix sum of x vanishes if and only if the number of brackets in y_1, \dots, y_{2i} is exactly i , which we can check with an interval query for $y_{-i} \dots y_{2i}$. \square

Theorem 8 *Every algorithm for dynamic membership for any Dyck language with two or more types brackets requires time $\Omega(\log n / \log \log n)$.*

Proof. We first prove the claim for D'_2 . We will use the *member* query for D'_2 to solve an instance of the prefix problem from the last proposition.

Let $x \in \{\llbracket, \rrbracket, \square\}^n$ be an instance of the dynamic interval problem for D'_1 . Construct

$$y = \square x_1 \square x_2 \square \dots \square x_n \square x^R$$

and note that y is in D'_2 . To answer an interval query about x we merely insert a matching pair of square brackets in y at the corresponding place:

$$y' = \square x_1 \square \dots \square x_{i-1} \llbracket x_i \rrbracket \dots \square x_j \llbracket x_{j+1} \rrbracket \dots \square x_n \square x^R$$

It is easy to see that $y' \in D'_2$ iff $x_1 \dots x_i \in D'_1$. After the query y' is changed back to y .

In the one-sided case, we have to extend both ends of the instance with parentheses to

$$y = \llbracket^{2n} \square x_1 \square x_2 \square \dots \square x_n \square x^R \rrbracket^{2n},$$

just to make sure that y is in D_2 . The rest of the proof is the same. \square

¹Apparently, in the mid-60s a handful of cards filled with closing brackets was always to be found next to Aarhus University's punched card reader for the IBM 7090 machine of the Northern Europe University Computing Centre in Lundtofte. These were routinely appended to users' Lisp programs to make sure they were properly balanced.

3.2 One-sided, one-letter membership

The only membership problem not covered by the last two results is D_1 . (Recall that the membership problem for D'_1 is in constant time by Proposition 6.) We need a totally different technique for this, and our bound still leaves an exponential gap, ending this thesis on an unhappy note.

We tentatively conjecture that the complexity of this problem is $\Omega(\log n / \log \log n)$.

Proposition 10 *The Dynamic Membership problem for D_1 requires*

$$\Omega\left(\frac{\log \log n}{\log \log \log n}\right)$$

time per operation with logarithmic cell size.

Proof. Consider the Dyck language D_1 over $\{\llbracket, \rrbracket\}$ for concreteness (the proof is the same for the two-sided case). Assume that we have an implementation for the Dynamic *Interval* problem for D_1 that handles updates and queries in time $t = t(n)$. (We transform the result to the membership problem at the end of the proof.) We will transform this problem to a static problem and show a lower bound for the latter by a reduction.

Consider the problem of finding a static data structure that is able to answer the following type of query in time t :

balance(i): return $(x_1 \cdots x_i \in D_1)$.

Note that no updates take place, and that the trivial solution (store all the answers in advance) uses linear space. We will now show that we can use far less space if x is not too different from $(\llbracket \rrbracket)^{n/2}$. The data structure is based on the algorithm for the dynamic problem. Initialise the data structure to $(\llbracket \rrbracket)^{n/2}$. Let M_0 denote the resulting contents of the machine's memory. Use the *change* operation to transform $(\llbracket \rrbracket)^{n/2}$ into x ; let M_x denote the resulting memory contents. Note that M_0 and M_x differ at no more than rt cells, where r denotes the number of changes. We store the difference in a perfect hash table, using $O(rt)$ space. We can hardwire M_0 into our algorithm and hence we can simulate the query operations as if the memory was M_x using only $O(rt)$ space.

We introduce now another static problem, for which a lower bound is known. The *range query* problem is to find a scheme to store an arbitrary set $S \subseteq \{1, \dots, n\}$, using space $O(|S|^{O(1)})$, such that the following type of query can be answered:

parity(i): return the value $|S \cap \{1, \dots, i\}| \bmod 2$.

Note that by storing S in an ordered list, we achieve a size $|S|$ data structure that makes all queries answerable in time $O(|S|)$. It is known that for any scheme with the stated size bound there exists a set S for which there is a lower bound of

$$t = \Omega\left(\frac{\log \log n}{\log \log \log n}\right) \tag{1}$$

on the time t needed for a query.

All that is left is to reduce the range query problem to the static Dyck problem introduced above. Given $S \subseteq \{1, \dots, n\}$, construct the string $x \in \{\square, \square\}^{2n}$ as follows: for each $i \notin S$, we let $x_{2i-1}x_{2i} = \square\square$, and for each $i \in S$, we let

$$x_{2i-1}x_{2i} = \begin{cases} \square\square, & \text{if } |S \cap \{1, \dots, i-1\}| = 0 \pmod{2}, \\ \square\square, & \text{otherwise.} \end{cases}$$

It is easy to see that

$$|S \cap \{1, \dots, i\}| = 0 \pmod{2} \quad \text{if and only if} \quad x_1 \cdots x_{2i} \in D_1.$$

Hence we can use the data structure for the static Dyck prefix problem to solve the range query Problem for arbitrary S . The size of this data structure is $O(|S|t)$, which is polynomial in $|S|$ (recall that we can assume $t \in O(|S|)$), and therefore the lower bound (1) applies.

We leave it to the reader to transform the proof to the membership problem, using a similar trick as in the proof of Thm. 8. \square

The compress-and-communicate technique used to prove (1) is due to Miltersen [49] and Xiao [71], combining ideas of Willard [70] and Ajtai [2]. The result has been improved to that stated in (1) by Beame and Fich [8]. Miltersen *et al.* [50] give an accessible presentation of a weaker result.

Other operations With any other set of operations, even D_1 is hard. The bound for insertions and deletions follows from [25], since list ranking is difficult. The next result shows the bound for *change* and *match*.

Proposition 11 *The dynamic membership problem with match queries for D_1 requires*

$$\Omega\left(\frac{\log n}{\log \log n}\right)$$

time per operation with logarithmic cell size.

Proof. Let $x \in \{0, 1\}^n$ be an instance of the Dynamic Parity Prefix problem. Define $z \in \{\square, \square\}^{3n}$ by

$$z = \square^2 h(x_n) \square^2 h(x_{n-1}) \square^2 \cdots \square^2 h(x_2) \square^2 h(x_1),$$

where $h(0) = \square$ and $h(1) = \square$.

We represent x by the string

$$y = \square^{3n} z \square^{3n} z^{-1}.$$

Note that y is always in D_1 and hence any match query will be well-defined. Indeed, we have

$$\text{match}(6n - 3i + 1) = 6n + i + 2 \cdot |x_1 \cdots x_i|_1 \quad \text{for } i = 1, \dots, n,$$

so we can calculate $x_1 + \cdots + x_i \pmod{2}$ in constant time given the answer to the match query. \square

Notation

$|A|$ cardinality of the set A

$A\Delta B$ symmetric difference of the sets A and B , i.e. $(A \cup B) \setminus (B \cap A)$

$|x|_a$ number of a s in the string x

x^R the string x reversed

$\lfloor a \rfloor$ floor of a

$\lceil a \rceil$ ceiling of a

$\log a$ logarithm to the base 2 of a

$(a = b)$ 1 if $a = b$ and 0 otherwise

$(a \in S)$ 1 if $a \in S$ and 0 otherwise

(P) 1 if P holds and 0 otherwise

Roots. The notation for floors and ceilings is from the early 60s; according to [28] it was introduced (together with the handy (P) -notation) by Iverson [38], who “found that typesetters could handle the symbols by shaving the tops and bottoms off of ‘ \lfloor ’ and ‘ \lceil ’.”

Bibliography

- [1] Miklós Ajtai. Σ_1^1 -formulae on finite structures. *Ann. Pure and Applied Logic*, 24:1–48, 1983.
- [2] Miklós Ajtai. A lower bound for finding predecessors in Yao’s cell probe model. *Combinatorica*, 8(3):235–247, 1988.
- [3] Miklós Ajtai and Michael Ben-Or. A theorem on probabilistic constant depth computations. In *Proc. 16th STOC*, pages 471–474, 1984.
- [4] David Albers, Giuseppe Cattaneo, and Giuseppe F. Italiano. An empirical study of dynamic graph algorithms. In *Proc. 7th SODA*, pages 190–201, 1996.
- [5] Arne Anderson. Sorting and searching revisited. In *Proc. 5th SWAT*, volume 1097 of *Lecture Notes in Computer Science*, pages 185–197. Springer Verlag, Berlin, 1996.
- [6] Giuseppe Di Battista and Roberto Tamassia. Algorithms for plane representations of acyclic digraphs. *Theoretical Computer Science*, 61:175–198, 1988.
- [7] Hanna Baumgarten, Hermann Jung, and Kurt Mehlhorn. Dynamic point location in general subdivisions. In *Proc. 3rd SODA*, pages 250–258, 1992.
- [8] Paul Beame and Faith Fich, 1994. Personal communication, reported by Peter Bro Miltersen.
- [9] Jon Louis Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.
- [10] Ravi B. Boppana and Michael Sipser. The complexity of finite functions. In Jan van Leeuwen, editor, *Algorithms and complexity*, volume A of *Handbook of theoretical computer science*, chapter 14, pages 757–804. Elsevier, Amsterdam, 1990.
- [11] Gerth Stølting Brodal. Predecessor queries in dynamic integer sets. In *Proc. 14th STACS*, 1997. to appear.
- [12] Gerth Stølting Brodal and Thore Husfeldt. A communication complexity proof that symmetric functions have logarithmic depth. Technical Report RS-96-1, BRICS, 1996.

- [13] Bettina Brustmann and Ingo Wegener. The complexity of symmetric functions in bounded-depth circuits. *Information Processing Letters*, 25(4):217–219, 1987.
- [14] Yi-Jen Chiang and Roberto Tamassia. Dynamic algorithms in Computational Geometry. Technical Report CS-91-24, Dept. of Comp. Sc., Brown University, 1991.
- [15] Larry Denenberg, Yuri Gurevich, and Saharon Shelah. Definability by constant-depth polynomial-size circuits. *Information and Control*, 70(2/3):216–240, aug/sep 1986.
- [16] Paul F. Dietz. Optimal algorithms for list indexing and subset rank. In *Proc. 1st WADS*, volume 382 of *Lecture Notes in Computer Science*, pages 39–46. Springer Verlag, Berlin, 1989.
- [17] David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Dynamic graph algorithms. In *Handbook of Algorithms and Theory of Computation*, chapter 22. CRC Press, 1997.
- [18] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. In *Proc. 33rd FOCS*, pages 60–69, 1992.
- [19] David Eppstein, Giuseppe Italiano, Roberto Tamassia, Robert E. Tarjan, Jeffery Westbrook, and Moti Yung. Maintenance of a minimum spanning forest in a dynamic planar graph. *Journal of Algorithms*, 13:33–54, 1992.
- [20] Ronald Fagin, Maria M. Klawe, Nicholas J. Pippenger, and Larry Stockmeyer. Bounded-depth, polynomial-size circuits for symmetric functions. *Theoretical Computer Science*, 36(2–3):239–250, April 1985.
- [21] Gudmund Skovbjerg Frandsen, Thore Husfeldt, Peter Bro Miltersen, Theis Rauhe, and Søren Skyum. Dynamic algorithms for the Dyck languages. In *Proc. 4th WADS*, volume 955 of *Lecture Notes in Computer Science*, pages 98–108. Springer, 1995.
- [22] Gudmund Skovbjerg Frandsen, Peter Bro Miltersen, and Sven Skyum. Dynamic word problems. In *Proc. 34th FOCS*, pages 470–479, 1993.
- [23] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal of Computing*, 14(4):781–798, 1985.
- [24] Michael L. Fredman. Observations on the complexity of generating quasi-Gray codes. *SIAM Journal of Computing*, 7(2):134–146, 1978.
- [25] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st STOC*, pages 345–354, 1989.

- [26] Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial time hierarchy. *Math. Systems Theory*, 17:13–27, 1984.
- [27] Harold N. Gabow and Matthias Stallman. Efficient algorithms for graphic matroid intersection and parity. In *Proc. 12th ICALP*, volume 194 of *Lecture Notes in Computer Science*, pages 210–220. Springer Verlag, Berlin, 1985.
- [28] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison–Wesley, 1989.
- [29] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th FOCS*, pages 8–21. IEEE Computer Society, 1978.
- [30] Michael A. Harrison. *Introduction to Formal Language Theory*. Addison–Wesley, 1978.
- [31] Monika Rauch Henzinger and Valerie King. Fully dynamic biconnectivity and transitive closure. In *Proc. 36th FOCS*, 1995.
- [32] Monika Rauch Henzinger and Valerie King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *27th STOC*, pages 519–527. ACM, 1995.
- [33] Johann T. Håstad. Almost optimal lower bounds for small depth circuits. In *Proc. 18th STOC*, pages 6–20, 1986.
- [34] Thore Husfeldt. Fully dynamic transitive closure in plane dags with one source and one sink. In *Proc. 3rd ESA*, volume 955 of *Lecture Notes in Computer Science*, pages 199–212. Springer Verlag, Berlin, 1995.
- [35] Thore Husfeldt and Theis Rauhe. Cell probe lower bounds for re-evaluating symmetric functions and for dynamic algorithms. Manuscript, 1996.
- [36] Thore Husfeldt, Theis Rauhe, and Søren Skyum. Lower bounds for dynamic transitive closure, planar point location, and parentheses matching. *Nordic Journal of Computing*, 3(4):323–336, 1996.
- [37] Giuseppe F. Italiano, Alberto Marchetti Spaccamela, and Umberto Nanni. Dynamic data structures for series parallel digraphs. In *Proc. First WADS*, volume 382 of *Lecture Notes in Computer Science*, pages 352–373. Springer Verlag, Berlin, 1989.
- [38] Kenneth E. Iverson. *A Programming Language*. Wiley, 1962.
- [39] Mauricio Karchmer and Avi Wigderson. Monotone circuits for connectivity require super-logarithmic depth. *SIAM J. Disc. Math*, 3(2):255–265, May 1990.
- [40] Richard M. Karp and Michael O. Rabin. Efficient randomised pattern-matching algorithms. *IBM J. Res. Develop.*, 31(2):249–260, March 1987.

- [41] David Kelly. On the dimension of partially ordered sets. *Discrete Mathematics*, 35:135–156, 1981.
- [42] Donald Ervin Knuth. *Computer Modern Typefaces*, volume E of *Computers & Typesetting*. Addison–Wesley, 1986.
- [43] Donald Ervin Knuth. *The T_EXbook*, volume A of *Computers & Typesetting*. Addison–Wesley, 1986.
- [44] Richard J. Lipton and Yechezkel Zalcstein. Word problems solvable in logspace. *Journal of the ACM*, 24(3):522–526, 1977.
- [45] Wilhelm Magnus, Abraham Karrass, and Donald Solitar. *Combinatorial Group Theory*, volume 13 of *Pure and Applied Mathematics*. Interscience Publishers, 1966.
- [46] Udi Manber and Martin Tompa. The complexity of problems on probabilistic, nondeterministic, and alternating decision trees. *Journal of the ACM*, 32(3):720–732, July 1985.
- [47] Kurt Mehlhorn, Rajamani Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality-tests in polylogarithmic time. In *Proc. 5th SODA*, pages 213–222. ACM-SIAM, 1994.
- [48] Peter Bro Miltersen. On-line reevaluation of functions. Technical Report DAIMI PB–380, Computer Science Department, Aarhus University, 1992.
- [49] Peter Bro Miltersen. Lower bounds for union–split–find related problems on random access machines. In *Proc. 26th STOC*, pages 625–634. ACM, 1994.
- [50] Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. On data structures and asymmetric communication complexity. In *Proc. 27th STOC*, pages 103–111. ACM, 1995.
- [51] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models for incremental computation. *Theoretical Computer Science*, 130:203–236, 1994.
- [52] Marvin L. Minsky and Seymour A. Papert. *Perceptrons*. MIT Press, 1969.
- [53] Burkhard Monien. About the derivation languages of grammars and machines. In *Proc. 4th ICALP*, volume 52 of *Lecture Notes in Computer Science*, pages 337–351. Springer Verlag, Berlin, 1977.
- [54] Shlomo Moran. Generalized lower bounds derived from Hastad’s main lemma. *Information Processing Letters*, 25:383–388, 1987.
- [55] Yu. Ofman. On the algorithmic complexity of discrete functions. *Dokl. Akad. Nauk SSSR*, 145:48–51, 1962. English translation appears in *Soviet Physics Doklady* 7(7):589–591, 1963.

- [56] Mark H. Overmars. *The design of dynamic data structures*, volume 156 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1983.
- [57] Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and Systems Sciences*, 23(2):166–204, 1981.
- [58] Sushant Patnaik and Neil Immerman. Dyn-FO: A parallel, dynamic complexity class. In *Proc. 13th ACM Symp. on Principles of Database Systems (PODS)*, pages 210–221, 1994.
- [59] Franco P. Preparata and Roberto Tamassia. Fully dynamic point location in a monotone subdivision. *SIAM Journal of Computing*, 18(4):811–830, 1989.
- [60] Theis Rauhe and Søren Skyum. Dyck-sprogene – dynamiske algoritmer og deres kompleksitet. Master’s thesis, Department of Computer Science, Århus University, 1995.
- [61] Robert W. Ritchie and Frederick N. Springsteel. Language recognition by marking automata. *Information and Control*, 20:313–330, 1972.
- [62] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and Systems Sciences*, 26:362–391, 1983.
- [63] Sairam Subramanian. A fully dynamic data structure for reachability in planar digraphs. In *Proc. 1st Ann. European Symp. on Algorithms (ESA)*, volume 726 of *Lecture Notes in Computer Science*, pages 372–383. Springer Verlag, Berlin, 1993.
- [64] Roberto Tamassia. On-line planar graph embedding. *Journal of Algorithms*, 21(2):201–239, 1996.
- [65] Roberto Tamassia and Franco P. Preparata. Dynamic maintenance of planar digraphs, with applications. *Algorithmica*, 5:509–527, 1990.
- [66] Roberto Tamassia and Ioannis G. Tollis. Dynamic reachability in planar digraphs with one source and one sink. *Theoretical Computer Science*, 119:331–343, 1993.
- [67] Mikkel Thorup. On RAM priority queues. In *Proc. 7th SODA*, pages 59–67, 1996.
- [68] Walther von Dyck. Gruppentheoretische studien. *Math. Ann.*, 20:1–45, 1882.
- [69] C. S. Wallace. A suggestion for a fast multiplier. *IEEE Trans. Comput.*, 3(1):14–17, 1964.
- [70] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.

- [71] B. Xiao. *New bounds in cell probe model*. Doctoral dissertation, University of California, San Diego, 1992.
- [72] Andrew C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, July 1981.

Recent BRICS Dissertation Series Publications

- DS-97-3 Thore Husfeldt. *Dynamic Computation*. December 1997. PhD thesis. 90 pp.
- DS-97-2 Peter Ørbæk. *Trust and Dependence Analysis*. July 1997. PhD thesis. x+175 pp.
- DS-97-1 Gerth Stølting Brodal. *Worst Case Efficient Data Structures*. January 1997. PhD thesis. x+121 pp.
- DS-96-4 Torben Braüner. *An Axiomatic Approach to Adequacy*. November 1996. Ph.D. thesis. 168 pp.
- DS-96-3 Lars Arge. *Efficient External-Memory Data Structures and Applications*. August 1996. Ph.D. thesis. xii+169 pp.
- DS-96-2 Allan Cheng. *Reasoning About Concurrent Computational Systems*. August 1996. Ph.D. thesis. xiv+229 pp.
- DS-96-1 Urban Engberg. *Reasoning in the Temporal Logic of Actions — The design and implementation of an interactive computer system*. August 1996. Ph.D. thesis. xvi+222 pp.