BRICS

Basic Research in Computer Science

BRICS RS-95-1 Frandsen et al.: Dynamic Algorithms for the Dyck Languages

Dynamic Algorithms for the Dyck Languages

Gudmund Skovbjerg Frandsen
Thore Husfeldt
Peter Bro Miltersen
Theis Rauhe
Søren Skyum

BRICS Report Series RS-95-1

ISSN 0909-0878 January 1995

See back inner page for a list of recent publications in the BRICS Report Series. 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 WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

DYNAMIC ALGORITHMS FOR THE DYCK LANGUAGES

GUDMUND SKOVBJERG FRANDSEN, THORE HUSFELDT,
PETER BRO MILTERSEN, THEIS RAUHE, AND SØREN SKYUM

BRICS[*]
Department of Computer Science, University of Aarhus,
Ny Munkegade, DK–8000 Århus C, Denmark

11th January 1995

Abstract. We study dynamic membership problems for the Dyck languages, the class of strings of properly balanced parentheses. We also study the Dynamic Word problem for the free group. We present deterministic algorithms and data structures which maintain a string under replacements of symbols, insertions, and deletions of symbols, and language membership queries. Updates and queries are handled in polylogarithmic time. We also give both Las Vegas- and Monte Carlo-type randomised algorithms to achieve better running times, and present lower bounds on the complexity for variants of the problems.

[*] Basic Research in Computer Science, Centre of the Danish National Research Foundation.

This work was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 7141 (project ALCOM II). Gudmund Frandsen was partially supported by CCI-Europe. Peter Bro Miltersen was partially supported by a grant from the Danish Natural Science Research Council, part of his research was done done at the Department of Computer Science, University of Toronto.

1. Introduction

1.1. Dyck Languages. The language of properly balanced parentheses contains strings like ( ) and ( )(( )) but not)). The notion of balancedness also makes sense if we add more types of parentheses: ([ ])( ) balances but [ ) does not.

More formally, let $A = \{a_1, \ldots, a_k\}$ and $\bar{A} = \{\bar{a}_1, \ldots, \bar{a}_k\}$ be two disjoint sets of opening and closing symbols,respectively. For example, the pair $A = \{(, [, \textbf{do}, \textbf{if}\}$ and $\bar{A} = \{\,),], \textbf{od}, \textbf{fi}\}$ 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:

$$S \to SS \mid a_1 S \bar{a}_1 \mid \cdots \mid a_k S \bar{a}_k \mid \epsilon.$$

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

$$S \to SS \mid a_1 S \bar{a}_1 \mid \bar{a}_1 S a_1 \mid \cdots \mid a_k S \bar{a}_k \mid \bar{a}_k S a_k \mid \epsilon.$$

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 $x \in D'_k$ if and only if x 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 $a_1^{-1} a_2 a_2^{-1} a_1$ evaluates to unity.

The Dyck languages are covered in detail in Harrison's classical treatment [6].

1.2. The Dynamic Membership problem. In this paper 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 implement a data type that contains a string $x \in (A \cup \bar{A})^n$ of even length, initially a_1^n, with the following operations:

change(i, a)**:** change x_i to $a \in A \cup \bar{A}$,
member: return 'yes' if and only if $x \in D_k$.

The problem is well motivated: several modern editors 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. For such an application, the set of operations considered above is clearly not sufficient. At the very least, we need an operation **insert**(i, a) that inserts a symbol a between the $(i-1)$th and the ith character in the string, and an operation **delete**(i) that deletes the ith character in the string. Furthermore, the following operations would also be useful: an operation **prefix**(i) that tells whether the prefix $x_1 x_2 \cdots x_i$ is a member of the language, and, for the one-sided languages, an operation **match**(i) that given the position i of a parenthesis returns the index of its match, as well as an operation **mismatch** that returns the index of the first unmatched parenthesis. We choose the restricted set of operations above for reasons of expositional simplicity; most of our results are also valid under the extended set, we state explicitly when they are not.

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, this paper is a step in the right direction. We do not expect the algorithms in this paper 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 parenthesis matching.

1.3. Results. Our main model of computation will be a unit-cost random access machine with word-size $\mathrm{O}(\log n)$, where n is the size of the input; this model is also known as a *random access computer*.

Our main result is that the Dynamic Membership problem for all Dyck languages can be solved in polylogarithmic time per operation, the bound exact is $\mathrm{O}(\log^3 n \log^* n)$. We use a technique for maintaining dynamic sequences under equality tests by Mehlhorn, Sundar, and Uhrig [10], which also gives (Las Vegas-style) randomised algorithms that run in slightly better expected time: $\mathrm{O}(\log^3 n)$.

We achieve better bounds for Monte Carlo-style algorithms. Using the fingerprint method of Karp and Rabin [7], 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 $\mathrm{O}(\log^2 n)$ and D'_k in time $\mathrm{O}(\log n)$. For D_1 and D'_1 we can use simpler techniques to achieve better bounds. The table below states the order of the upper bounds. Except for the $\mathrm{O}(1)$ 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, prefix queries, and (for the one-sided case) match queries.

Table 1: Running times of algorithms with logarithmic word size.

Language	Operations	Type of algorithm		
		Deterministic	Las Vegas	Monte Carlo
D_1 or D'_1	all	$\mathrm{O}(\log n)$		
D'_1	**change member**	$\Theta(1)$		
D_k	all	$\mathrm{O}(\log^3 n \log^* n)$	$\mathrm{O}(\log^3 n)$	$\mathrm{O}(\log^2 n)$
D'_k	all	$\mathrm{O}(\log^3 n \log^* n)$	$\mathrm{O}(\log^3 n)$	$\mathrm{O}(\log n)$

We have no lower bounds for the restricted set of operations (**change**, **member**). However, if the **prefix**-operation is added, we can get a weak lower bound of $\Omega(\log\log n/\log\log\log n)$ using a technique from [11], obviously, the same bound holds with the **mismatch** query instead. If instead we allow insertion and deletion, a lower bound of $\Omega(\log n/\log\log n)$ can be derived from a result of Fredman and Saks [4]. The same lower bound holds if we replace **member** by **match** in the restricted of operations (for one-sided languages).

Table 2: Lower bounds for logarithmic word size.

Language	Operations	Lower bound
D_1 or D_1'	**change prefix**	$\Omega\big(\frac{\log\log n}{\log\log\log n}\big)$
D_1 or D_1'	**insert delete member**	$\Omega\big(\frac{\log n}{\log\log n}\big)$
D_1	**change match**	$\Omega\big(\frac{\log n}{\log\log n}\big)$

It is interesting that all upper bounds for the two-sided case are at least as good as the upper bounds for the one-sided case, implying that the former may be easier than the latter. In a more restricted model for dynamic algorithms, namely the cell probe model with cell size 1 (the *bit probe model*), we can indeed separate the complexity of the two problems: for D_1, we prove a lower bound of $\Omega(\log n/\log\log n)$ by a technique of Fredman [3], while we can bound the complexity of D_1' from above by $\mathrm{O}(\log\log n)$ using a construction of [2]. The latter bound is shown to be tight by a reduction from the Dynamic Word problem for the monoid $(\{0,1\},\vee)$, for which a lower bound is given in [2]. The upper bound for D_1 is $\mathrm{O}(\log n\log\log n)$, not quite matching the lower bound. These results are only valid for the restricted set of operations (and are only of theoretical interest anyway). The table below summarises these results.

Table 3: Bit probe complexity.

Language	Operations	Upper bound	Lower bound
D_1	**change member**	$\mathrm{O}(\log n\log\log n)$	$\Omega\big(\frac{\log n}{\log\log n}\big)$
D_1'	**change member**	$\Theta(\log\log n)$	

1.4. Related results. It is interesting that all Dyck languages seem to be equally hard in most non-dynamic computational models. Ritchie and Springsteel [13] showed that the one-sided Dyck languages are in deterministic logspace, Lipton and Zalcstein [8] extended this to the two-sided case (see also [6, 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 [2, 11]. 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 [2] give better upper bounds depending on the language's syntactic monoid $M(L)$.

1.5. Preliminaries and notation. Strings will be denoted by lower-case letters $u, v, x, \ldots$. We let u_i denote the ith letter of string u and we write $u_{i:j}$ for $u_i\cdots u_j$. For letter a and string u, we put

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

the number of occurrences of a in u. All logarithms are base two.

We call a string *reduced* if it contains no neighbouring pair of matching parentheses. 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 [6]), we introduce two mappings

$$\mu_1, \mu_2 \colon (A \cup \bar{A})^* \to (A \cup \bar{A})^*.$$

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 $j = 1, 2$:

$$\mu_1(\epsilon) = \mu_2(\epsilon) = \epsilon, \qquad \mu_1(ua_i) = \mu_1(u)a_i,$$

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

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

One can show properties like $\mu_1(ua_i\bar{a}_iv) = \mu_1(uv)$ and $\mu_2(ua_i\bar{a}_iv) = \mu_2(u\bar{a}_ia_iv) = \mu_2(uv)$.

We formally define u^{-1} as $\bar{u}_n \cdots \bar{u}_1$ with the convention $\bar{\bar{a}} = a$ and $\epsilon^{-1} = \epsilon$.

2. Algorithms for One Pair of Parentheses

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

Proposition 2.1. *The Dynamic Membership problem for D_1' can be done in constant time per operation.*

Proof. Note first that for all $x \in \{a, \bar{a}\}^*$ we have

$$x \in D_1' \quad \leftrightarrow \quad |x|_a = |x|_{\bar{a}}.$$

The only if direction is obvious. The other follows from the fact that a reduced string over $\{a, \bar{a}\}^*$ cannot contain both a and $\bar{a}$.

Hence we only need to count the number of occurrences of a and $\bar{a}$ in x. With unit cost operations, this is easily done in constant time per update. □

The solution does *not* extend to the extended set of operations, in fact a larger lower bound is proved below. We leave it to the reader to prove a logarithmic upper bound. Using am algorithm by Dietz [1] a solution with an $\mathrm{O}(\log n/\log\log n)$ upper bound on the *amortised* complexity for the extended set of operations can also be found.

Proposition 2.2. *The Dynamic Membership problem for D_1 can be done in $\mathrm{O}(\log n)$ time per operation.*

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

We maintain a balanced binary tree whose ith 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

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

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. Most complicated are the **insert** and **delete** operations. To accommodate these, we have to maintain balance in the tree using any scheme for balancing dynamic search trees, e.g. *red–black trees* [5]. 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 4.2 calls for the most complicated extensions, in that we also need to be able to split and merge trees.

3. Algorithms for Many Pairs of Parentheses

We move now to the main result, extending the above to larger k. The basic idea resembles very much the data structure from Proposition 2.2: we represent x as a balanced binary tree, where 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.

3.1. A data structure for strings equality. Mehlhorn, Sundar, and Uhrig [10] 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,
destroy(s)**:** remove s from 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 techniques from [10] can easily be modified to cope with the above updates. The time bounds are summarised in the following lemma:

Lemma 3.1 ([10])**.** *Let S_i denote the family of strings after the ith operation and define*

$$N_m = \max_{0 \le i \le m} \sum_{s \in S_i} |s|.$$

There is a data structure for the above problem such that the mth operation takes time $\mathrm{O}(\log^2 N_m \log^* N_m)$.

3.2. The two-sided case.

Proposition 3.1. *The Dynamic Membership problem for D'_k can be done in* $\mathrm{O}(\log^3 n \log^* n)$ *time per operation.*

Proof. We maintain a balanced binary tree whose ith 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 Section 3.1 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

$$\mu_2(u) = u'aw \quad \text{and} \quad \mu_2(v) = w^{-1}bv', \qquad \text{with } \bar{a} \neq b, \tag{3.1}$$

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

$$\mu_2(uv) = u'abv'. \tag{3.2}$$

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 $\mathrm{O}(\log^2 N \log^* N)$, where N denotes the total length of all sequences in the tree. Now split $\mu_2(u)$ and $\mu_2(v)$ as in (3.1) above and construct $\mu_2(uv)$ by (3.2), using a constant number of operations, each of which takes time $\mathrm{O}(\log^2 N \log^* N)$. We remember to remove unused strings. The total number of operations for an update is then $\mathrm{O}(\log n \log^2 N \log^* N)$. To bound N we note that at each level of the tree, the total length of all sequences maintained at that level is $2n$ and hence $N = \mathrm{O}(n \log n)$, which gives the desired bound and completes the proof. □

3.3. 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.

Proposition 3.2. *The Dynamic Membership problem for D_k can be done in* $\mathrm{O}(\log^3 n \log^* n)$ *time per operation.*

Proof. As before, we maintain a balanced binary tree whose ith 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 paranthesis cancels every right paranthesis, regardless of its type, by

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

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

$$\mu_1(y) \in \bar{A}^* A^* \quad \text{if and only if} \quad \exists u, v \colon uyv \in D_k, \tag{3.3}$$

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

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

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,

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

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

The upper bounds from the last two propositions can be improved to expected time $\mathrm{O}(\log^3 n)$ by using the Las Vegas variant of the algorithm described in Section 3.1, see [10].

4. Monte Carlo Algorithms

4.1. 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 [7].

Proposition 4.1. *The Dynamic Membership problem for* D'_k *can be done in* $\mathrm{O}(\log n)$ *time per operation such that the probability of an erroneous answer in any sequence of* n *updates is* $\mathrm{O}(\frac{1}{n})$.

Proof. We start by considering D'_2 over the alphabet $A = \{a_1, a_2\}$ and $\bar{A} = \{\bar{a}_1, \bar{a}_2\}$. 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* $\{a_1, a_2\}$) with concatenation as the operator and ϵ as the identity.

Following Lipton and Zalcstein [8] (see also [9, Problem 2.3.13]), we represent $(A \cup \bar{A})^*/\sim$ as a group of 2×2 integer matrices using the group homomorphism

$$h\colon (A \cup \bar{A})^*/\sim \;\to M_2(Z),$$

$$h(a_1) = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad h(a_2) = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}.$$

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

This suggests a randomised algorithms in the spirit of [7]: 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 ith leaf contains $h(x_i)$ and an internal node contains the product (in $M_2(Z_p)$) 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 $h(x) \equiv 1 \bmod p$ 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 [12] we guarantee that the probability of an erroneous answer in a sequence of n consecutive queries is bounded by $\mathrm{O}(\frac{1}{n})$.

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 $c_1, \ldots, c_k$ generate a free group, see [9, Problem 1.4.12]. □

4.2. 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 3.2) 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 Section 3.1 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 4.2. *The Dynamic Membership problem for D_k can be done in* $\mathrm{O}(\log^2 n)$ *time per operation such that the probability of an erroneous answer in any sequence of n updates is* $\mathrm{O}(\frac{1}{n})$.

Proof. As before, we maintain a balanced binary tree whose ith 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 3.2. 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 $w = y_{i_1} \ldots y_{i_l}$ for some $i_1 < \cdots < i_l$. Then we define

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

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 3.2. 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 $\mathrm{O}(\log n)$ 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 $\mathrm{O}(\log n)$, from which the stated time bound follows.

To bound the error probability, note that we use $\mathrm{O}(n)$ distinct versions of the data structure from Proposition 4.1. Using a prime from a larger set (say, $p \leq n^5$), we obtain the stated bound. □

5. Bit Probe Complexity

We turn now to consider solutions in the cell probe model with constant cell size, the *bit probe* model. Only the basic set of operations are considered. We will see that the strong restrictions of the model facilitate proving quite tight bounds and an exponential gap between the complexities of D_1 and D'_1. The lower bound techniques below are specific to the bit probe model, while both upper bounds are refinements of the algorithms in Section 2.

For the bounds in this and the following section, we must define the model of computation more precisely.

5.1. The Cell Probe Model. In the *cell probe* model (with *cell size* b), we focus on the number of memory cells accessed during the computation, all other operations are for free. More formally, to every instance of an update or query operation (say, $\mathbf{change}(3, a_2)$) we associate a *decision assignment tree*: a rooted tree of 2^b-ary read nodes and unary write nodes; the read nodes are labelled with a memory location l, and the write nodes are labelled with both a memory location l and a value $v \in \{0, \ldots, 2^b - 1\}$. Trees that correspond to query operations additionally have 'yes' or 'no'-leaves (or whatever possible answers there are to the query). To execute an operation, we start at the root of the corresponding tree and proceed towards the leaves. If at a write node (say, with label (l, v)), we write v into memory location l and proceed to the node's unique child. If at a read node (say, with label l), we proceed to the child pointed to by the contents of location l. The complexity of an operation is the maximum depth of its instances' trees. For example, the complexity of the change operation is the maximum depth of the trees for $\mathbf{change}(i, a)$ for $1 \leq i \leq n$ and $a \in A \cup \bar{A}$.

Note that the cell probe model is non-uniform. Obviously, lower bounds for this model are valid also on the random access machine with unit cost operations and word size b. In this section, we study the cell probe model with $b = 1$ (called the *bit probe* model), in the next section, we give lower bounds for $b = \mathrm{O}(\log n)$.

5.2. A fast one-bit counter. Since both Propositions 2.2 and 2.1 were based on counting, we will briefly review a construction from [2] that will be useful to us: a fast counter for cell size one.

The counter maintains a value c from some interval $\{l, \ldots, h\}$, for integers l, h. The counter supports the following operations:

increment: increment c by 1, provided $c < h$,
decrement: decrement c by 1, provided $c > l$,
test: return 'yes' iff $c = t$ for some fixed $t \in \{l, \ldots, h\}$.

It can be implemented such that incrementing and decrementing can be performed in time $\log\log(h - l) + \mathrm{O}(1)$ and the test operations takes constant time, see [2].

5.3. The two-sided case. The next two results give tight bounds on the bit probe complexity complexity of the language D'_1.

Proposition 5.1. *The Dynamic Membership problem for* D'_1 *can be done in* $\mathrm{O}(\log\log n)$ *time per operation in the bit probe model.*

Proof. Recall from the proof of Proposition 2.1 that we only need to count the number of as and $\bar{a}$s in the input. To this end we use the counter from the last paragraph, with $l = -n$, $h = n$, and $T = \{0\}$. □

Proposition 5.2. *The Dynamic Membership problem for* D'_1 *requires* $\Omega(\log\log n)$ *time per operation in the bit probe model.*

Proof. We use a result from [2], that the *Dynamic Word* problem for any commutative non-group requires time $\Omega(\log\log n)$ in the bit probe model. The Dynamic Word problem for a group G is to maintain a string $g_1 \cdots g_n$ of group elements under the **change** operation and a query that returns 'yes' if and only if $g_1 * \cdots * g_n$ (where $*$ denotes the product in G) is the identity.

We reduce the word problem for the monoid $(\{0,1\}, \vee)$ to an instance of D'_1. From $x \in \{0,1\}^n$ we construct a string $y \in \{a, \bar{a}\}^{2n}$ such that $y_{2i-1:2i} = a\bar{a}$ if $x_i = 0$, and $y_{2i-1:2i} = aa$ if $x_i = 1$. Clearly, $y \in D'_1$ if and only if x contains only zeroes. □

Note that the same proof works also for the one-sided case D_1. However, we will see that we can give a much stronger bound for that language in a moment.

5.4. The one-sided case. We turn to the one-sided Dyck language D_1. Again, the proof is a modification of the random access machine algorithm.

Proposition 5.3. *The Dynamic Membership problem for D_1 can be done in* $\mathrm{O}(\log n \log\log n)$ *time per operation in the bit probe model.*

Proof. Consider the proof of Proposition 2.2. With constant cell size, each of these calculations would take $\mathrm{O}(\log n)$ per level, for a total running time of $\mathrm{O}(\log^2 n)$. We can do better by observing that the we need at most four increments and decrements of r and l values at each level. Using the counter from Section 5.2, we achieve the stated bound. □

Now for the lower bound, which falls two $\log\log n$ factors short of matching the bound above. We thus establish that, perhaps surprisingly, there is an exponential gap between the complexities of D_1 and D'_1 for constant cell size. The technique is due to Fredman.

Proposition 5.4. *The Dynamic Membership problem for D_1 requires*

$$\Omega\Big(\frac{\log n}{\log\log n}\Big)$$

time par operation in the bit probe model.

Proof. Fredman [3] essentially shows the stated lower bound on the complexity of the *Which Side?* problem, which is to maintain a single value $t \in \{1, \ldots, n\}$ under the following operations:

change(i)**:** set $t = i$,
which side(i)**?:** return 'left' if $t < i$ and 'right' if $t > i$.

We reduce this problem to the Dynamic Membership problem for D_1. For an instance of Which Side? of size n we will maintain a string $x \in \{a, \bar{a}\}^{2n}$, initially $aa(a\bar{a})^{n-1}$. Generally, x will be of the form $(a\bar{a})^{2n}$ with the exception $x_{2t-1:2t} = aa$. The **change** operation is easily simulated to maintain this invariant. To simulate the **which side**(i)**?** operation, we put $x_{2i-1:2i} = \bar{a}\bar{a}$. Now x will balance iff $i > t$. We remember to change $x_{2i-1:2i}$ back after the query. □

6. Lower Bounds for Logarithmic Cell Size

We do not know any lower bound on the complexity of any Dyck language with the basic set of operations. However, when we extend the operations in two different ways, we can show lower bounds for *all* Dyck languages. First, we will modify the operations by replacing **member** with **match** and show a lower bound that is valid for all one-sided Dyck languages.

The first two lower bounds use a result of Fredman and Saks [4] that gives a lower bound of $\Omega(\log n/\log\log n)$ on the (amortised) complexity of the *Dynamic Parity Prefix* problem: given a vector $x_1, \ldots, x_n$ of bits, maintain a data structure that is able to react to the following operations for all $i = 1, \ldots, n$:

change(i)**:** negate the value of x_i,
parity(i)**:** return $\bigoplus_{j=1}^{i} x_j$, the parity of the first i elements.

6.1. With match queries.

Using the above, we can show the same lower bound for all one-sided Dyck languages when the only operations are **change** and **match**.

Proposition 6.1. *The Dynamic Membership problem with match queries for any one-sided Dyck language requires*

$$\Omega\Big(\frac{\log n}{\log\log n}\Big)$$

time per operation with logarithmic cell size.

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

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

where $h(0) = \bar{a}$ and $h(1) = a$.

We represent x by the string

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

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

$$\mathbf{match}(6n - 3i + 1) = 6n + i + 2 \cdot |x_{1:i}|_1 \qquad \text{for } i = 1, \ldots, n,$$

so we can calculate $\bigoplus_{j=1}^{i} x_i$ in constant time given the answer to the match query. □

6.2. With insert and delete.

We can show an $\Omega(\log n/\log\log n)$ lower bound with membership queries, provided insertions and deletions are allowed. Our reduction goes via the *List Representation* problem: maintain a list $L \in \{0,1\}^*$ under the following operations:

insert(i)**:** insert '1' between the $(i-1)$th and the ith element,
delete(i)**:** delete the ith element,
value(i)**:** return L_i.

The lower bound is stated without proof for a slightly different problem in [4], we give the proof here for completeness.

Lemma 6.1. *List Representation requires*

$$\Omega\Big(\frac{\log n}{\log\log n}\Big)$$

time per operation with logarithmic cell size.

Proof. Let $x \in \{0,1\}^n$ be an instance of the Parity Prefix problem and assume without loss of generality that n is even. Initially, set $L = (01)^{n^2}$. We will maintain the following invariant:

$$\bigoplus_{j=1}^{i} x_j \oplus k = L_{2ni+k}, \quad \text{for } i = 1, \dots, n \text{ and } k = -|x_{1:i}|_0, \dots, |x_{1:i}|_1.$$

So $L_{2ni} = \bigoplus_{j=1}^{i} x_j$, which shows how to use **value** to perform the **parity** operation. Whenever x_i is changed from zero to one, we perform **insert**$(i(2n-1))$, and whenever x_i is changed from one to zero, we perform **delete**$(i(2n-1))$. Note that under replacement updates, it is easy to keep track of the value of each x_i using only constant overhead, so we can indeed distinguish the two cases above. The straightforward but tedious proof that these operations maintain the invariant is left to the reader. □

We have left to show that the operations on the Dyck membership problems suffice to simulate the crucial **value** operation in the problem above.

Proposition 6.2. *The Dynamic Membership problem with inser and delete for any Dyck language requires*

$$\Omega\Big(\frac{\log n}{\log\log n}\Big)$$

time per operation with logarithmic cell size.

Proof. Let x be an instance of any Dyck language D that contains the pair $\{0,1\}$ of matching parentheses. We will show that we can simulate a **value** operation that given i returns x_i. Note that with that operation we can solve the Dynamic List Representation problem above, which implies the stated lower bound by the last lemma.

From x of length n construct a string y of length $4n$ as follows: the left half of y represents x in the sense that for $i = 1, \dots, n$ we have $y_{2i} = x_i$ and $y_{2i-1} = 0$, and the right half of y is constructed such that $y \in D_1$ by putting $y_{4n-i+1} = 1 - y_i$ for $i = 1, \dots, 2n$. These invariants are easily maintained under insertions and deletions.

To see if x_i is (say) zero, we change y_{2i} into zero and see if y still balances. It is clear that if indeed x_i is zero, then y has not changed and the answer is yes. On the other hand, if x_i is one then the new y cannot balance as it contains too many zeroes, no matter what language D is. □

6.3. With prefix. Finally, we give a (weaker) lower bound for any Dyck language, if the operations are augmented with a **prefix** query instead of insertions and deletions.

Proposition 6.3. *The Dynamic Membership problem with prefix for any Dyck language requires*

$$\Omega\Big(\frac{\log\log n}{\log\log\log n}\Big)$$

time per operation with logarithmic cell size.

Proof. Consider the Dyck language D_1 over $\{a, \bar{a}\}$ for concreteness (the proof is the same for the two-sided case). Assume that we have an implementation for the Dynamic Prefix problem for D_1 that handles updates and queries in time $t = t(n)$. We will transform the prefix problem for D_1 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 'yes' iff $x_{1:i} \in D_1$.

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 $(a\bar{a})^{n/2}$. The data structure is based on the algorithm for the dynamic problem. Initialise the data structure for $(a\bar{a})^{n/2}$ using **init**. Let M_0 denote the resulting contents of the machine's memory. Use the **change** operation to transform $(a\bar{a})^{n/2}$ 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 $\mathrm{O}(rt)$ 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 $\mathrm{O}(rt)$ 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 $S \subseteq \{1, \ldots, n\}$, using space $\mathrm{O}(|S|^{\mathrm{O}(1)})$, such that the following type of query can be answered:

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

Note that by storing S in an ordered list, we achieve a size $|S|$ data structure that makes all queries answerable in time $\mathrm{O}(|S|)$. Beame and Fich (personal communication) improving Miltersen [11], have shown that for any scheme (with the stated size bound) there exists a set S for which there is a lower bound of

$$t = \Omega\Big(\frac{\log\log n}{\log\log\log n}\Big) \tag{6.1}$$

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 $S \subseteq \{1, \ldots, n\}$, construct the string $x \in \{a, \bar{a}\}^{2n}$ as follows: for each $i \notin S$, we let $x_{2i-1:2i} = a\bar{a}$, and for each $i \in S$, we let

$$x_{2i-1:2i} = \begin{cases} aa, & \text{if } |S \cap \{1, \ldots, i-1\}| = 0 \mod 2, \\ \bar{a}\bar{a}, & \text{otherwise.} \end{cases}$$

It is easy to see that

$$|S \cap \{1, \ldots, i\}| = 0 \mod 2 \quad \text{if and only if} \quad x_{1:2i} \in D_1.$$

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 $\mathrm{O}(|S|t)$, which is polynomial in $|S|$ (recall that we can assume $t \in \mathrm{O}(|S|)$), and therefore the lower bound (6.1) applies. □

References

1. Paul F. Dietz, *Optimal algorithms for list indexing and subset rank*, Proc. First Workshop on Algorithms and Data Structures (WADS) (F. Dehne, J.-R. Sack, and N. Santoro, eds.), Lecture Notes in Computer Science, vol. 382, Springer Verlag, Berlin, 1989, pp. 39–46.
2. Gudmund Skovbjerg Frandsen, Peter Bro Miltersen, and Sven Skyum, *Dynamic word problems*, Proc 34th Ann. Symp. on Foundations of Computer Science (FOCS), 1993, pp. 470–479.
3. Michael L. Fredman, *The complexity of maintaining an array and computing its partial sums*, Journal of the ACM **29** (1982), 250–260.
4. Michael L. Fredman and Michael E. Saks, *The cell probe complexity of dynamic data structures*, Proc. 21st Ann. Symp. on Theory of Computing (STOC), 1989, pp. 345–354.
5. Leo J. Guibas and Robert Sedgewick, *A dichromatic framework for balanced trees*, Proc. 19th Ann. Symp. on Foundations of Computer Science (FOCS), IEEE Computer Society, 1978, pp. 8–21.
6. Michael A. Harrison, *Introduction to formal language theory*, Addison-Wesley, 1978.
7. Richard M. Karp and Michael O. Rabin, *Efficient randomised pattern-matching algorithms*, IBM J. Res. Develop. **31** (1987), no. 2, 249–260.
8. Richard J. Lipton and Yechezkel Zalcstein, *Word problems solvable in logspace*, Journal of the ACM **24** (1977), no. 3, 522–526.
9. Wilhelm Magnus, Abraham Karrass, and Donald Solitar, *Combinatorial group theory*, Pure and Applied Mathematics, vol. 13, Interscience Publishers, 1966.
10. K. Mehlhorn, R. Sundar, and C. Uhrig, *Maintaining dynamic sequences under equality-tests in polylogarithmic time*, Proc. 5th Ann. Symp. on Discrete Algorithms (SODA), ACM-SIAM, 1994, pp. 213–222.
11. Peter Bro Miltersen, *Lower bounds for union-split-find related problems on random access machines*, Proc. 26th Ann. Symp. on Theory of Computing (STOC), ACM, 1994, pp. 625–634.
12. Mark H. Overmars, *The design of dynamic data structures*, Lecture Notes in Computer Science, vol. 156, Springer Verlag, Berlin, 1983.
13. R. W. Ritchie and F. N. Springsteel, *Language recognition by marking automata*, Information and Control **20** (1972), 313–330.

Recent Publications in the BRICS Report Series

RS-95-1 Gudmund Skovbjerg Frandsen, Thore Husfeldt, Peter Bro Miltersen, Theis Rauhe, and Søren Skyum. *Dynamic Algorithms for the Dyck Languages*. January 1995. 21 pp.

RS-94-48 Jens Chr. Godskesen and Kim G. Larsen. *Synthesizing Distinguishing Formulae for Real Time Systems*. December 1994. 21 pp.

RS-94-47 Kim G. Larsen, Bernhard Steffen, and Carsten Weise. *A Constraint Oriented Proof Methodology based on Modal Transition Systems*. December 1994. 13 pp.

RS-94-46 Amos Beimel, Anna Gál, and Mike Paterson. *Lower Bounds for Monotone Span Programs*. December 1994. 14 pp.

RS-94-45 Jørgen H. Andersen, Kåre J. Kristoffersen, Kim G. Larsen, and Jesper Niedermann. *Automatic Synthesis of Real Time Systems*. December 1994. 17 pp.

RS-94-44 Sten Agerholm. *A HOL Basis for Reasoning about Functional Programs*. December 1994. PhD thesis. viii+224 pp.

RS-94-43 Luca Aceto and Alan Jeffrey. *A Complete Axiomatization of Timed Bisimulation for a Class of Timed Regular Behaviours (Revised Version)*. December 1994. 18 pp. To appear in *Theoretical Computer Science*.

RS-94-42 Dany Breslauer and Leszek Gąsieniec. *Efficient String Matching on Coded Texts*. December 1994. 20 pp.

RS-94-41 Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. *On Data Structures and Asymmetric Communication Complexity*. December 1994. 17 pp.

RS-94-40 Luca Aceto and Anna Ingólfsdóttir. *CPO Models for GSOS Languages — Part I: Compact GSOS Languages*. December 1994. 70 pp. An extended abstract of the paper will appear in: *Proceedings of CAAP '95*, LNCS, 1995.