A New Dynamic Algorithm for the Dyck Language

Sapozhnikov Denis

Higher School of Economics, Moscow, Russia,
dsapozhnikov@hse.ru

Abstract. We study Dynamic Membership problems for the two-sides Dyck language, the class of strings of properly balanced parentheses. We present a non-deterministic algorithm and a data structure such that together they solve the following problems: replacements of symbols, insertions, deletions of symbols, and language membership queries.

Key words: Dyck Language, balanced braces sequence, dynamic algorithm, persistent structure.

1 Introduction

1.1 The Dyck Language

The Dyck Language is a language describing balanced bracket sequences. Let $A = \{a_1, a_2, \dots, a_k\}$ and $\overline{A} = \{\overline{a}_1, \overline{a}_2, \dots, \overline{a}_k\}$ be two disjoint sets of opening and closing symbols, respectively. For example, the pair $A = \{(, [, do, if\}$ and $\overline{A} = \{),], od, fi\}$ captures the nested structure of programming languages. The two-sided Dyck language D'_k over $A \cup \overline{A}$ is the context-free language generated by the following grammar:

- $\epsilon \in D'_k$
- If $S \in D'_k$ then $\forall i : a_i S \overline{a}_i \in S$
- If $S_1, S_2 \in D'_k$ then $S_1 S_2 \in D'_k$

1.2 The Dynamic Membership Problem

In this paper we consider the problem of maintaining membership in D'_k of a string from $(A \cup \overline{A})^n$ dynamically. More precisely, we want to implement a data structure $\mathcal{D}$ containing a string $x \in (A \cup \overline{A})^n$ of even length, with the following behaviour:

1. At creation, $\mathcal{D}$ is initialized by the empty string.
2. The following operations are supported by $\mathcal{D}$:
   member: return 'yes' if and only if $x \in D'_k$
   create(σ): create a new data structure $\mathcal{D}$ with $x = \sigma$
   split(i): split string x into two strings $x' = x_1 x_2 \dots x_i$ and $x'' = x_{i+1} x_{i+2} \dots x_n$
   merge(x', x''): concatenate two strings x' and x'' into one $x = x'x''$

1.3 Results

Our main model of computation will be a unit-cost random access machine with word-size O(log n), where n is the size of the input; this model is also known as a random access computer. We present a new data structure solving the Dynamic Membership Problem. Each query is completed in $O(\log n \log h)$ time, where h is the maximum nesting depth of brackets in the string and $O(n \log n)$ memory. In the worst case our result is equal to the one by Frandsen et al. [1].

2 Algorithm

2.1 A Data Structure for String Equality Verification

We create a non-deterministic structure $\mathcal{S}$ that handles the following operations for strings over a finite alphabet Σ:

create(σ): create a new string $s = \sigma$ in time $O(1)$
split(s, i): create two new strings $s' = s_1 s_2 \dots s_i, s'' = s_{i+1} s_{i+2} \dots s_n$ in time $O(\log n)$
merge(s', s''): create new string $s = s's''$ in time $O(\log(|s's''|))$
equal(s', s''): return 'yes' if and only if $s' = s''$ in time $O(1)$
copy(s): create a new copy of string s in time $O(1)$

We can use a implicit persistent treap [2, 3] (or other self-balancing binary search trees [4] with implicit keys) with polynomial hashing [5] to comply with the all requirements. Let each v-th vertex contains a polynomial hash of $s_{l \dots r}$ if it represents substring $s_{l \dots r}$. This simple idea allows you to implement $\mathcal{S}$.

2.2 Theoretical Background

Let us define inverse string s^{-1} to be $\overline{s}_n \overline{s}_{n-1} \dots \overline{s}_1$ with the convention $\overline{\overline{a}}_i = a_i$ and $\overline{\epsilon} = \epsilon$. We say that a string is *reduced* if it contains no neighbouring pair of matching parentheses. Let us introduce a one-sided associative calculus by alphabet $\Sigma = A \cup \overline{A}$ with relations of matching parentheses: $\forall i : a_i \overline{a}_i \to \epsilon$.

Let μ be a map from Σ^* to Σ^* such that $\mu(s)$ is derived from s in one-sided associative calculus and $\mu(s)$ is not reducible.

Lemma 1. *The map $\mu(s)$ is a function. In other words, for any string s there exists a unique $\mu(s)$ regardless of the order of reducing actions.*

Lemma 2. *Let s be a substring of string t. If any $a_i \overline{a}_j$ is a substring of $\mu(s)$ then $t \notin D'_k$.*

Lemma 3. *Suppose that any substring s of a string t does not satisfies the hypotheses of Lemma 2; then $\mu(t) = \overline{a}_{i_1} \overline{a}_{i_2} \dots \overline{a}_{i_k} a_{j_1} a_{j_2} \dots a_{j_m}$. This means that $\mu(t)$ consists of a prefix of closing braces and a suffix of opening braces.*

We say that a string is *complementary* if it satisfied the hypotheses of Lemma 3. For any complementary string t we can denote two functions $\overline{\phi}(t), \phi(t) : \Sigma^* \to \Sigma^*$ as follows:

$$\overline{\phi}(t) = \overline{a}_{i_1}\overline{a}_{i_2}\dots\overline{a}_{i_k}, \phi(t) = a_{j_1}a_{j_2}\dots a_{j_m}$$

Lemma 4. *Let $s = xy$ such that $x, y, s \in \Sigma^*$. The string s is complementary if and only if x and y is complementary strings, and one of the following conditions:*

1. *If $\phi(x) = x_1x_2$, where $x_1, x_2 \in \Sigma^*$ and $(x_1)^{-1} = \overline{\phi}(y)$, then $\overline{\phi}(s) = \overline{\phi}(x)$ and $\phi(s) = x_2\phi(y)$.*
2. *$\overline{\phi}(y) = \overline{y}_1\overline{y}_2$, where $\overline{y}_1, \overline{y}_2 \in \Sigma^*$ and $(\overline{y}_2)^{-1} = \phi(x)$, then $\overline{\phi}(s) = \overline{\phi}(x)\overline{y}_1$ and $\phi(s) = \phi(y)$.*

All proof contains in the Frandsen et al. [1] and Harrison [6].

2.3 The Main Data Structure

Let $\mathcal{D}$ be the structure under string s. We maintain a balanced binary tree in $\mathcal{D}$ whose v-th vertex represents substring $s_{l\dots r}$ and where each internal node represents the concatenation of its children's strings $s_{l\dots m}$ and $s_{m+1\dots r}$. If $s_{l\dots r}$ is complementary string, then we will save $\phi(s_{l..r})$ and $\overline{\phi}(s_{l..r})$ in v. In other case, we will save only one bit about uncomplementary.

We can easily compute the vertex data. If substrings $s_{l\dots m}$ or $s_{m\dots r}$ is not complementary, then $s_{l\dots r}$ is not complementary by the Lemma 4. After, we need to check two conditions from statement of Lemma 4 using the Data Structure for String Equality Verification $\mathcal{S}$. If each of them unsatisfied, then $s_{l\dots r}$ is not complementary. In other case, we will compute $\phi(s_{l..r})$ and $\overline{\phi}(s_{l..r})$ by the formulas from Lemma 4.

Calculating this information takes $O(\log|\mu(s_{l\dots m})| + \log|\mu(s_{m\dots r})|)$ time using our data structure $\mathcal{S}$. If the maximum nesting depth of brackets is h, then updating of one vertex can be completed in $O(\log h)$ time. Therefore, member, split and merge requests can be completed in time $O(\log n \log h)$ per operation and $O(n \log n)$ memory total, because of structure $\mathcal{S}$ is persistent.

References

1. Frandsen et al. Dynamic Algorithms for the Dyck Languages, pp. 98–108 (1995).
2. Seidel, R., Aragon, C.R. Randomized search trees. Algorithmica 16, pp. 464–497 (1996).
3. Driscoll, J., Sarnak, N., Sleator, D., Tarjan, R. Making data structures persistent. Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing. pp. 109–121 (1986).
4. Andersson, A. General Balanced Trees. Journal of Algorithms, pp. 1-18 (1999).
5. Dietzfelbinger, M. Polynomial hash functions are reliable. Automata, Languages and Programming, Springer Berlin Heidelberg, pp. 235–246 (1992).
6. Harrison, A. Introduction to Formal Language Theory. Addison-Wesley (1978).