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 Dyck language, the class of strings of properly balanced parentheses. We present non-deterministic algorithm and data structure which maintain a string under replacements of symbols, insertions, deletions of symbols, and language membership queries.

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

1 Introduction

1.1 Dyck Language

Dyck Language is a formal language describing balanced bracket sequences.

Let $A = \{a_1, a_2, \ldots, a_k\}$ and $\overline{A} = \{\overline{a}_1, \overline{a}_2, \ldots, \overline{a}_k\}$ 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 type that contains a string $x \in (A \cup \overline{A})^n$ of even length, initially empty s, with the following operations:

member(x): return 'yes' if and only if $x \in D'_k$
split(x, i): split string x into two strings $x' = x_1 x_2 \ldots x_i$ and $x'' = x_{i+1} x_{i+2} \ldots x_n$
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 witch solve the Dynamic Membership Problem. Each query is completed in $O(\log n \log h)$ time, where h is maximum nesting depth of brackets and $O(n \log n)$ memory. In worst case our result is equal the last by Frandsen et al. [1].

2 Algorithm

2.1 A Data Structure for String Equality

We need a structure that handles the following operations for strings over a finite alphabet Σ:

create(ϵ): create a new empty string $s = \epsilon$ 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 $O(1)$

For example, we can use persistent treap with polynomial hashing to comply with all requirements.

2.2 Theoretical Background

By definition, put inverse string $s^{-1} = \overline{s}_n \overline{s}_{n-1} \dots \overline{s}_1$ with the convention $\overline{\overline{a}}_i = a_i$ and $\overline{\epsilon} = \epsilon$

We call a string *reduced* if it contains no neighbouring pair of matching parentheses.

Let introduce 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 the map from Σ^* to Σ^* such that $\mu(s)$ derived from s in one-sided associative calculus and $\mu(s)$ is not reduced.

Lemma 1. *The mapping $\mu(s)$ is a function.*

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

Lemma 3. *Suppose any substring s of string t does not satisfies lemma 2; then $\mu(t) = \overline{a}_{i_1} \overline{a}_{i_2} \dots \overline{a}_{i_k} a_{j_1} a_{j_2} \dots a_{j_m}$, where $1 \le i_1 < i_2 < \dots < i_k < j_1 < j_2 < \dots < j_m \le n$. This means that $\mu(t)$ consists of prefix of closing braces and suffix of opening braces.*

If lemma 3 is satisfies then we can denote two functions $\overline{\phi}(t), \phi(t) : \Sigma^* \to \Sigma^*$:

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

Lemma 4. *Let $s = xy$ such that $x, y, s \in \Sigma^*$. For the string s to satisfies lemma 3 it is necessary and sufficient to have x and y is satisfies lemma 3 and one of two conditions is satisfied:*

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

In first case $\overline{\phi}(s) = \overline{\phi}(x)$ and $\phi(s) = x_2\phi(y)$.
In second case $\overline{\phi}(s) = \overline{\phi}(x)\overline{y}_1$ and $\phi(s) = \phi(y)$.

2.3 A Main Data Structure

We maintain a balanced binary tree whose v-th leaf represents $s_{l..r}$ and where each internal node represents the concatenation of its children's strings $s_{l..m}$ and $s_{m+1..r}$. If $s_{l..r}$ satisfied lemma 3 then we will save $\overline{\phi}(s_{l..r})$ and $\phi(s_{l..r})$. In other case we will save only one bit about unsatisfying.

We can update vertex info using lemma 2, lemma 4 and the Data Structure for String Equality in $O(\log|\mu(s_{l..r})|)$ time. If the maximum nesting depth of brackets is h then update 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.

References

1. Gudmund Skovbjerg Frandsen, Thore Hu feldt, Peter Bro Miltersen, Theis Rauhe, and Seren Skyum: Dynamic Algorithms for the Dyck Languages (1995), https://link.springer.com/chapter/10.1007/3-540-60220-8_54.