Journal of Discrete Algorithms 31 (2015) 120–128

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

String shuffle: Circuits and graphs

Neerja Mhaskar [a,1], Michael Soltys [b,*,1]

[a] McMaster University, Dept. of Computing & Software, 1280 Main Street West, Hamilton, Ontario L8S 4K1, Canada
[b] California State University Channel Islands, Dept. of Computer Science, One University Drive, Camarillo, CA 93012, USA

ARTICLE INFO

Article history:
Available online 19 January 2015

Keywords:
String shuffle
Circuit complexity
Lower bounds
Monge

ABSTRACT

We show that shuffle, the problem of determining whether a string w can be composed from an order preserving shuffle of strings x and y, is not in $\mathbf{AC}^0$, but it is in $\mathbf{AC}^1$. The fact that shuffle is not in $\mathbf{AC}^0$ is shown by a reduction of parity to shuffle and invoking the seminal result of Furst et al., while the fact that it is in $\mathbf{AC}^1$ is implicit in the results of Mansfield. Together, the two results provide a lower and upper bound on the complexity of this combinatorial problem. We also explore an interesting relationship between graphs and the shuffle problem, namely what types of graphs can be represented with strings exhibiting the anti-Monge condition.

1. Introduction

Suppose that we are given three strings x, y, w over the binary alphabet $\Sigma = \{0, 1\}$. The shuffle problem asks the following question: can we form w as a "shuffle" of x and y? That is, can we compose the third string by weaving together the first two, while preserving the order within each string? For example, over the binary alphabet $\Sigma = \{0, 1\}$, given 000, 111, and 010101, we can obviously answer in the affirmative. We give the formal definition of the shuffle operation in Section 1.1.

Mansfield [17] shows that a clever dynamic programming algorithm can determine whether w, is a shuffle of x, y in time $O(|w|^2)$, and the same paper poses the question of determining a lower bound. In this paper we show a fairly tight upper and lower bound for the shuffling problem in terms of circuit complexity. We show that:

(i) bounded depth circuits of polynomial size *cannot* solve shuffle, but that
(ii) logarithmic depth circuits of polynomial size *can* do so.

This paper is an expanded version of [30], and it contains more detailed proofs, as well as new results in Section 4.2. The reader is encouraged to download the Python code `shuffle.py`, from the corresponding author's web page, in order to experiment with the ideas in Section 4.2. Finally, since the publication of [30], the paper [4] has also appeared, containing the related but independent result that unshuffling a square is **NP**-hard.

The paper is structured as follows: in Section 1.3 we give the background on circuit complexity. In Sections 2 and 3 we give the upper and lower bounds on the circuit complexity of shuffle, respectively. The bounds are summarized in

* Corresponding author.
E-mail address: michael.soltys@csuci.edu (M. Soltys).
URL: http://soltys.cs.csuci.edu (M. Soltys).
[1] Supported in part by the NSERC Discovery Grant.

http://dx.doi.org/10.1016/j.jda.2015.01.003
1570-8667/© 2015 Elsevier B.V. All rights reserved.

Theorem 5. In Section 4.1 we examine other reductions to shuffle, and we remark on the high expressibility of the shuffle predicate, i.e., we show that basic properties of strings can be re-stated in terms of shuffles. In Section 4.2 we examine the connection between graph properties and the shuffle predicate. We finish with open problems in Section 5.

1.1. Definitions

A shuffle is sometimes also called a "merge" or an "interleaving". The intuition for the definition is that w can be obtained from u and v by an operation similar to shuffling two decks of cards.

If x, y, and w are strings over an alphabet Σ, then w is a *shuffle* of x and y provided there are (possibly empty) strings x_i and y_i such that $x = x_1x_2\cdots x_k$ and $y = y_1y_2\cdots y_k$ and $w = x_1y_1x_2y_2\cdots x_ky_k$. Note that $|w| = |x| + |y|$ is a necessary condition for the existence of a shuffle. The advantage of the definition given here is that it is very succinct; the problem is that it can be misleading: there are many ways to shuffle two strings, not just strict alternation of one symbol from each string. But keep in mind that some x_i and y_j may be ε, so for example, if $x_1 \neq \varepsilon$, $x_2 \neq \varepsilon$, and $y_1 = \varepsilon$, then this would mean that we take the first two symbols of x before we take any symbols from y. In short, by choosing certain x_i's and y_j's equal to ε's, we can obtain any shuffle from an ostensibly strictly alternating shuffle.

The predicate $\text{Shuffle}(x, y, w)$ holds if and only if w is a shuffle of x, y, as described in the above paragraph. We define the language Shuffle as, $\text{Shuffle} = \{\langle x, y, w\rangle : \text{Shuffle}(x, y, w)\}$. Given this terminology, the bounds proven in this paper can be stated as: $\text{Shuffle} \notin \mathbf{AC}^0$, but $\text{Shuffle} \in \mathbf{AC}^1$.

Shuffling can be defined over any alphabet, but in this paper we work mostly with the binary alphabet $\Sigma = \{0, 1\}$. The naming convention we use is that lower case "shuffle" and "parity" denote the generic problems, while $\text{Shuffle}(x, y, w)$ and $\text{Parity}(x)$ denote the corresponding predicates, and Shuffle and Parity without arguments denote the corresponding languages.

We will work with circuits that compute the predicate $\text{Shuffle}(x, y, w)$, or alternatively decide the language Shuffle. Let $a \cdot b$ denote the *concatenation* of two strings, and let $\langle x, y, w\rangle$ denote the encoding of three strings. For shuffle we can simply let $\langle x, y, w\rangle = x \cdot y \cdot w$, as for a well formed input $|x| = |y| = n$ and $|w| = 2n$, so we can extract x, y, w from $x \cdot y \cdot w$. Thus, a family of circuits $C = \{C_n\}$ that computes $\text{Shuffle}(x, y, w)$ is parametrized by $4n$, that is, the circuit C_n has $4n$ inputs, corresponding to the $4n$ bits of $x \cdot y \cdot w$. We define circuits in Section 1.3.

1.2. History

Following the presentation of the history of shuffle in [4], we mention that the initial work on shuffles arose out of abstract formal languages. Shuffles were later motivated by applications to modeling sequential execution of concurrent processes. The shuffle operation was first used in formal languages by Ginsburg and Spanier [8]. Early research with applications to concurrent processes can be found in Riddle [23,24] and Shaw [27]. A number of authors, including [9,10, 12–16,19,20,28] have subsequently studied various aspects of the complexity of the shuffle and iterated shuffle operations in conjunction with regular expression operations and other constructions from the theory of programming languages.

In the early 1980's, Mansfield [17,18], and Warmuth and Haussler [33], studied the computational complexity of the shuffle operator on its own. The paper [17] gave a polynomial time dynamic programming algorithm for computing $\text{Shuffle}(x, y, w)$.

In [18] this was extended to give polynomial time algorithms for deciding whether a string w can be written as the shuffle of k strings $u_1, \dots, u_k$, for a *constant* integer k. The paper [18] further proved that if k is allowed to vary, then the problem becomes **NP**-complete (via a reduction from Exact Cover with 3-Sets).

Warmuth and Haussler [33] gave an independent proof of the above result, and went on to give a rather striking improvement by showing that this problem remains **NP**-complete even if the k strings $u_1, \dots, u_k$ are equal. That is to say, the question of, given strings u and w, whether w is equal to an *iterated shuffle* of u is **NP**-complete. Their proof used a reduction from 3-Partition.

In [4] we show that square shuffle, i.e., the problem of determining whether a given string w is a shuffle of some x with itself, that is, whether the predicate $\exists x, |x| < |w| \wedge \text{Shuffle}(x, x, w)$ holds, is **NP**-hard. The paper [25] gives an alternative proof of the same result.

1.3. Background on circuits

A *Boolean circuit* can be seen as a directed, acyclic, connected graph in which the input nodes are labeled with variables x_i and constants 1, 0, representing true and false, respectively, and the internal nodes are labeled with standard Boolean connectives $\wedge$, $\vee$, $\neg$, that is, AND, OR, NOT, respectively. We often use $\bar{x}$ to denote $\neg x$, and the circuit nodes are often called *gates*.

The *fan-in*, i.e., number of incoming edges, of a $\neg$-gate is always one, and the fan-in of $\wedge$, $\vee$ can be arbitrary, even though for some complexity classes, such as $\mathbf{SAC}^1$ defined below, we require that the fan-in be bounded by a constant. The *fan-out*, i.e., number of outgoing edges, of any node can also be arbitrary. Note that when the fan-out is restricted to be exactly one, circuits become Boolean formulas. Each node in the graph can be associated with a Boolean function in the obvious way. The function associated with the output gate(s) is the function computed by the circuit. Note that a Boolean