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

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.

Keywords: String shuffle, circuit complexity, lower bounds, Monge

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.

[*] Corresponding author
Email address: michael.soltys@csuci.edu (Michael Soltys)
URL: http://soltys.cs.csuci.edu (Michael Soltys)

[1] Supported in part by the NSERC Discovery Grant

Mansfield [1] 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 [2], 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 [2], the paper [3] 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 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 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, Shuffle $= \{\langle x, y, w\rangle : \text{Shuffle}(x, y, w)\}$. Given this terminology, the bounds proven in this paper can be stated as: Shuffle $\notin \mathbf{AC}^0$, but 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 Shuffle(x, y, w) and 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 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 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 [3], 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 [4]. Early research with applications to concurrent processes can be found in Riddle [5, 6] and Shaw [7]. A number of authors, including [8, 9, 10, 11, 12, 13, 14, 15, 16, 17] 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 [1, 18], and Warmuth and Haussler [19], studied the computational complexity of the shuffle operator on its own. The paper [1] gave a polynomial time dynamic programming algorithm for computing 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 $u_1, \ldots, u_k$, 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 [19] 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 $u_1, \ldots, u_k$ 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 [3] 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 [20] 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 formula can be seen as a circuit in which every node has fan-out one, and $\wedge, \vee$ have fan-in 2, and $\neg$ has fan-in one. Thus, a Boolean formula is a Boolean circuit whose structure is tree-like.

The *size* of a circuit is its number of gates, and the *depth* of a circuit is the maximum number of gates on any path from an input gate to an output gate.

A *family of circuits* is an infinite sequence $C = \{C_n\} = \{C_0, C_1, C_2, \ldots\}$ of Boolean circuits where C_n has n input variables. We say that a Boolean predicate P has polysize circuits if there exists a polynomial p and a family C such that $|C_n| \leq p(n)$, and $\forall x \in \{0,1\}^*$, $P(x)$ holds iff $C_{|x|}(x) = 1$. In the case of shuffle, the family C computes shuffle if:

$$\text{Shuffle}(x, y, w) \text{ holds} \iff C_{|\langle x,y,w\rangle|}(\langle x, y, w\rangle) = 1.$$

Note that $|\langle x, y, w\rangle| = 4n$, as explained at the end of Section 1.1, and the circuit C_n decides all the different inputs of length $4n$.

Let $\mathbf{P}$/poly be the class of all those predicates which have polysize circuit families. It is a standard result in complexity that all predicates in $\mathbf{P}$ have polysize circuits; that is, if a predicate has a polytime Turing machine, it has polysize circuits. The converse of the above does not hold, unless we put a severe restriction on how the n-th circuit is generated; as it stands, there are undecidable predicates that have polysize circuits. The restriction that we place here is that there is a Turing machine that on input 1^n computes $\{C_n\}$ in space $O(\log n)$. This restriction makes a family C of circuits *uniform*.

The predicates (or Boolean functions) that can be decided (or computed) with polysize, constant fan-in, and depth $O(\log^i n)$ circuits, form the class $\mathbf{NC}^i$. The class $\mathbf{AC}^i$ is defined in the same way, except we allow unbounded fan-in. We set $\mathbf{NC} = \bigcup_i \mathbf{NC}^i$, and $\mathbf{AC} = \bigcup_i \mathbf{AC}^i$, and while it is easy to see that the uniform version of $\mathbf{NC}$ is in $\mathbf{P}$, it is an interesting open question whether they are equal.

We have the following standard result: for all i,

$$\mathbf{AC}^i \subseteq \mathbf{NC}^{i+1} \subseteq \mathbf{AC}^{i+1}.$$

Thus, $\mathbf{NC} = \mathbf{AC}$. Finally, $\mathbf{SAC}^i$ is just like $\mathbf{AC}^i$, except we restrict the $\wedge$ fan-in to be at most two.

$\mathbf{L}$ and $\mathbf{NL}$ stand for deterministic and non-deterministic logarithmic space, respectively, and $\mathbf{UL}$ stands for "Unambiguous Logarithmic Space." We say that a language is in the class $\mathbf{UL}$ if it is decided by a log-space bounded non-deterministic Turing machine, which has the added property that on "yes" instances there is exactly one accepting path (while there may be several in $\mathbf{NL}$). Recall that $\mathbf{NC}^1 \subseteq \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{NC}^2$ and $\mathbf{UL} \subseteq \mathbf{NL}$. It is not known whether any of these containments are strict. In fact, the question whether $\mathbf{UL}$ equals $\mathbf{NL}$ is a long standing problem (see [21, 22]).

2. Upper bound

We start by showing a circuit upper bound for shuffle, that is, we show that Shuffle $\in \mathbf{SAC}^1$, which means that shuffle can be decided with a polysize family of circuits of logarithmic depth (in the size of the input), where all the $\wedge$-gates have fan-in 2. This result relies on the dynamic programming algorithm given in [1] and the complexity result of [23, 24] which shows that $\mathbf{NL} \subseteq \mathbf{SAC}^1$.

In this section all our circuit results hold with the uniformity condition imposed, and so we do not mention it explicitly.

In order to show that Shuffle $\in$ **NL**, we use the fact that shuffle can be reduced (in low complexity) to the graph reachability problem [25, 1]. The idea is to construct a grid graph, with $(|x|+1)\times(|y|+1)$ nodes; the lower-left node is represented with $(0,0)$ and the upper-right node is represented with $(|x|,|y|)$. For any $i<|x|$ and $j<|y|$, we have the edges:

$$\begin{cases} ((i,j),(i+1,j)) & \text{if } x_{i+1}=w_{i+j+1} \\ ((i,j),(i,j+1)) & \text{if } y_{j+1}=w_{i+j+1}. \end{cases} \tag{1}$$

Note that both edges may be present, which is what introduces the element of non-determinism in the traversal of the graph, and reflects the fact that two strings can be shuffled in many ways that initially may seem promising to form w.

We explain how to interpret this graph. A path starts at $(0,0)$, and the i-th time it goes up we pick x_i, and the j-th time it goes right we pick y_j. Thus, a path from $(0,0)$ to $(|x|,|y|)$ represents a particular shuffle.

For example, consider Figure 1. On the left we have a shuffle of 000 and 111 that yields 010101, and on the right we have a shuffle of 011 and 011 that yields 001111. The left instance has a unique shuffle that yields 010101, which corresponds to the unique path from $(0,0)$ to $(3,3)$. On the right, there are several possible shuffles of $011, 011$ that yield 001111 — in fact, eight of them, each corresponding to a distinct path from $(0,0)$ to $(3,3)$.

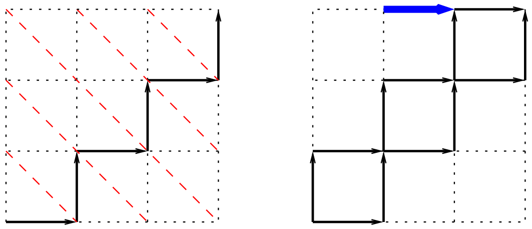

Figure 1: On the left we have a shuffle of 000 and 111 that yields 010101, and on the right we have a shuffle of 011 and 011 that yields 001111. The dynamic programming algorithm in [1] computes partial solutions along the red diagonal lines. The thick blue arrow in the right diagram is there to symbolize that there are other edges beside the black ones; the blue edge is $((1,3),(2,3))$ and it is there because $x_{1+1}=x_2=1=w_5=w_{1+3+1}$. The edges are placed according to (1).

The number of paths is always bounded by:

$$\binom{|x|+|y|}{|x|}$$

and this bound is achieved for $\langle 1^n, 1^n, 1^{2n}\rangle$. Thus, the number of paths can be exponential in the size of the input, and so an exhaustive search is not feasible in general.

Lemma 1. *Shuffle* $\in$ **NL***.*

PROOF: The algorithm in [1] reduces shuffle on $\langle x, y, w\rangle$, $|x| = |y| = n$, $|w| = 2n$, to directed graph reachability. The graph is an $(n+1) \times (n+1)$ grid of nodes, with the lower-left corner labeled $(0, 0)$, and the upper-right corner labeled (n, n), and the edges placed according to (1).

The correctness of the reduction follows from the assertion that given the edges of the grid, defined as in the paragraph above, there is a path from $(0, 0)$ to (i, j) if and only if the first $i+j$ bits of w can be obtained by shuffling the first i bits of x and the first j bits of y. Thus, node (n, n) can be reached from node $(0, 0)$ if and only if Shuffle(x, y, w) is true.

Given $\langle x, y, w\rangle$, this can be checked with three pointers of size $O(\log n)$ each; one pointer for the i-th position in x, one pointer for the j-th position in y, and one pointer for the position $i + j$. If $x_{i+1} = y_{j+1} = w_{i+j+1}$, $i + 1$ or $j + 1$ is picked non-deterministically; if (n, n) is reached, we accept. □

Corollary 2. *Shuffle* $\in$ $\mathbf{SAC}^1$

PROOF: Since $\mathbf{NL} \subseteq \mathbf{SAC}^1$ (see [23, 24]) it follows directly from Lemma 1 that Shuffle $\in \mathbf{SAC}^1 \subseteq \mathbf{AC}^1$. □

The reachability problem for general graphs is in the class **NL**. However, Lemma 1 shows that shuffle reduces to a very particular kind of graph: a grid graph, where all edges point up or to the right. It would be interesting to know if the fact that the graphs are so restricted could improve the upper bound.

Since grid graphs are planar, and [26] shows that reachability in planar graphs can be decided in **UL**, it follows that Shuffle $\in$ **UL**. Of course, the fact that Shuffle $\in$ **UL** does not mean that there is a unique path from $(0, 0)$ to (n, n); rather, it means that if a shuffle exists (and there may be many), this is established on a unique computational path.

Further, since all edges point up or to the right, the grid graph in Lemma 1 is also layered. The reachability in such graphs has been studied in [27]. It would be interesting to know whether the results contained therein could improve the upper bound for shuffle.

An interesting question is whether any of the above observations can be used to show Shuffle $\in \mathbf{NC}^1$, i.e., shuffle can be decided with a polysize family of Boolean *formulas*. We conjecture that the answer is affirmative.

3. Lower bound

We show a circuit lower bound for shuffle, that is, Shuffle $\not\in \mathbf{AC}^0$, which means that shuffle cannot be decided with a polysize family of circuits of constant depth where all the $\wedge, \vee$-gates may have arbitrary fan-in.

Our proof relies on the seminal complexity result showing that parity is not in $\mathbf{AC}^0$, due to [28]. A very accessible presentation of this result can be found in [29, Chapters 11 & 12]; many of the details of that presentation are made explicit in [30, section 5.3].

In Section 2 we showed a uniform $\mathbf{SAC}^1$ upper bound for Shuffle. In this section we show a non-uniform $\mathbf{AC}^0$ lower bound. Of course, a uniform upper bound is stronger than a non-uniform one, and a non-uniform lower bound is stronger than a uniform one.

We start with a definition: let $|x|_s$ be the number of occurrences of a symbol s in the string x. Obviously, Shuffle$(0^{|x|_0}, 1^{|x|_1}, x)$ is always true. We can use this observation in order to reduce parity to shuffle, where the reduction itself is $\mathbf{AC}^0$. Let Parity $= \{x : |x|_1 \text{ is odd}\}$.

Here is the outline of the argument: suppose that we have a "black-box" that takes $\langle x, y, w\rangle$ as input bits and computes Shuffle(x, y, w). We could then construct a circuit for Parity with the standard gates $\wedge, \vee, \neg$, plus black-boxes for computing shuffle. If the black-boxes for shuffle were computable with $\mathbf{AC}^0$ circuits, we would then obtain an $\mathbf{AC}^0$ circuit for Parity, giving us a contradiction. The details are given below in Lemma 3, Figure 2, and Corollary 4.

Lemma 3. Parity $\in \mathbf{AC}^0$[Shuffle].

Proof: In order to compute the Parity of x, run the following algorithm: for all odd $i \in \{0, \ldots, |x|\}$, check if Shuffle$(0^{|x|-i}, 1^i, x)$ is true; if it is the case for at least one i, then the Parity of x is 1. Note that if it is true for at least one i, it is true for exactly one i. In terms of circuits, this can be expressed as follows:

$$\text{Parity}(x) = \bigvee_{\substack{0 \leq i \leq |x| \\ i \text{ is odd}}} \text{Shuffle}(0^{|x|-i}, 1^i, x). \tag{2}$$

See Figure 2. This gives us an $\mathbf{AC}^0$ circuits with “black-boxes” for shuffle, and hence the Lemma follows. □

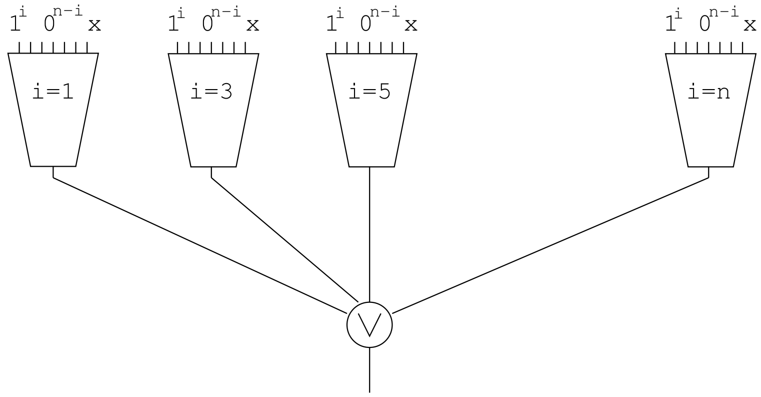

Figure 2: Parity of x computed in terms of Shuffle; note that we assume that n is odd in this Figure. If n were even the last “black box” for shuffle would be for $i = n - 1$.

Corollary 4. Shuffle $\notin \mathbf{AC}^0$.

Proof: Note that the size of the circuit for Parity in the proof of Lemma 3 is $O(|x| \cdot s(|x|))$, where $s(|x|)$ is the size of the circuit for shuffle. Thus, if Shuffle had polynomial size circuits, then so would Parity. Further, if the family of circuits computing Shuffle were of constant depth at most c, then the family of circuits computing Parity would be of constant depth at most $c+1$. Thus, if Shuffle were in $\mathbf{AC}^0$, so would Parity. Since by [28] Parity $\notin \mathbf{AC}^0$, it follows that Shuffle is not in $\mathbf{AC}^0$. □

By Corollaries 2 and 4 we have the following Theorem.

Theorem 5. Shuffle $\notin \mathbf{AC}^0$, *but* Shuffle $\in \mathbf{SAC}^1 \subseteq \mathbf{AC}^1 \subseteq \mathbf{NC}^2$.

The significance of this result is that shuffle cannot be decided with bounded depth circuits of polynomial size. On the other hand, shuffle can be decided with polynomial size circuits of unbounded fan-in and logarithmic depth — which in turn implies that shuffle can be decided in the class $\mathbf{NC}^2$. In general, the classes $\mathbf{NC}^i$ capture those problems that can be solved with polynomially many processors in poly-logarithmic time, which are problems that have fast parallel algorithms. See [31] for a discussion of $\mathbf{NC}^2$.

4. Further properties of shuffle

Although much progress has been done on understanding shuffle, many questions regarding shuffle remain open. For example, does Shuffle Square (given a string w, is it a shuffle of some x with itself), remain **NP**-hard for some alphabets with fewer than seven symbols? See [3]. We explore further properties of shuffle in order to better understand this operation.

In Section 4.1 we show how to express basic string operations with shuffle. In Section 4.2 we show that a particular type of string w gives rise to a bipartite graph on the symbols of w, exhibiting non-nesting properties of its arcs. A bipartite graph G_w on the symbols of the string w is a graph with nodes being the symbols of w, and edges being arcs joining pairs of the same symbol. The "Monge condition" allows nested edges but prohibits crossing edges, and the "anti-Monge condition" allows the opposite: prohibits nesting edges, and allows crossing edges.

The Monge condition has been widely studied for matching problems and transportation problems. Many problems that satisfy the Monge condition, also known as the "quasi-convex" condition, are known to have efficient polynomial time algorithms; for these see [32] and the references cited therein. There are fewer algorithms known for problems that satisfy the anti-Monge property, and some special cases are known to be **NP**-hard [33]. In Section 4.2 we explore the connection between Shuffle and the anti-Monge property.

4.1. Expressiveness of shuffle

It is interesting that several different string predicates reduce to shuffle in an easy and natural way. (2) gives a reduction of parity to shuffle, and we have the same for string equality as $x = y \iff \text{Shuffle}(\varepsilon, x, y)$.

Concatenation also reduces to shuffle. Let p_0, p_1 be "padding" functions on strings defined as follows:

$$p_0(x) = p_0(x_1x_2 \ldots x_n) = 00x_100x_200 \ldots 00x_n00$$
$$p_1(x) = p_1(x_1x_2 \ldots x_n) = 11x_111x_211 \ldots 11x_n11$$

that is, p_b, $b \in \{0,1\}$ pads the string x with a pair of b's between any two bits of x, as well as a pair of b's before and after x.

The third argument of Shuffle is just juxtaposition of two strings. We use "juxtaposition" since we do not want to define concatenation in terms of concatenation.

Claim 6. *$w = u \cdot v$ iff*

$$\text{Shuffle}(p_0(u), p_1(v), p_0(w_1 w_2 \dots w_{|u|}) p_1(w_{|u|+1} w_{|u|+2} \dots w_{|u|+|v|})).$$

Proof: The direction "⇒" is easy to see; for direction "⇐" we use the following notation:

$$\begin{aligned} r &= p_0(u) = 00u_1 00\dots \\ s &= p_1(v) = 11v_1 11\dots \\ t &= p_0(w_1 w_2 \dots w_{|u|}) p_1(w_{|u|+1} w_{|u|+2} \dots w_{|u|+|v|}) = 00w_1 00\dots \end{aligned}$$

If t is a shuffle of r, s, i.e., Shuffle(r, s, t), then we must take the first two bits of r (00) in order to cover the first two bits of t (00). If $u_1 = w_1 = 1$, then we could ostensibly take the first bit of s (1), but the bit following w_1 is 0, and $u_1 = 1$ and the second bit of s is 1; so taking the first bit of s leads to a dead end. Thus, we must use u_1 to cover w_1. We continue showing that we must first take all of r, and then take all of s in order to cover t. This argument can be formalized with induction. It follows that Shuffle(r, s, t) implies $t = r \cdot s$, which in turn implies $w = u \cdot v$. □

4.2. Expressing graphs with shuffles

In this section we explore the connection between general graphs, and the shuffle problem. In particular, we want to explore which graphs can be represented with strings that have the anti-Monge condition. We start with the simplest case, where strings are restricted to have exactly two occurrences of each symbol (so in this section we deal with strings over large alphabets). The shuffling properties of such strings will reflect the "connectedness" properties of the corresponding graph[2]. Let Σ_w denote the set consisting of the symbols in w, and so $|\Sigma_w| < |w|$, unless each symbol occurs exactly once, in which case $|\Sigma_w| = |w|$.

[2] The reader is encouraged to download the Python program `shuffle.py` from the corresponding author's web page, which implements many of the constructions given here.

A *pair-string* is a string where each symbol occurs exactly twice, i.e., where $\forall s \in \Sigma_w, |w|_s = 2$. A pair-string can also be viewed as a shuffle of some x with some permutation of the symbols of x, when every symbol of x is distinct. A pair-string w is said to be *anti-Monge* with respect to two symbols u, v, if neither $uvvu$ nor $vuuv$ is a subsequence of w. This is related to the usual anti-Monge condition in that the arcs that join u to u and v to v are nested if $uvvu$ or $vuuv$ is a subsequence of w.

Given a graph $G = (V, E)$, we want to construct a string w_G with the following property: if (u, v) is an edge in E, then w_G is anti-Monge with respect to u, v, and, conversely, if (u, v) is not an edge in E, then w_G is not anti-Monge with respect to u, v. Since we are going to be representing graphs with strings, we define Σ_V to be the alphabet of symbols corresponding to the vertices in V. Without confusion, given $V = \{v_1, v_2, \ldots, v_m\}$, we let $\Sigma_V = \{v_1, v_2, \ldots, v_m\}$.

We say that a pair-string w_G *represents* G if given any two vertices u, v in V, and given the corresponding positions $u^1 < u^2$ and $v^1 < v^2$ of the symbols u and v in w_G, the following holds depending on whether or not (u, v) is an edge in E:

Case $(u, v) \in E$: Exactly one of the following four conditions on the indices $u^1 < u^2$ and $v^1 < v^2$ holds:

$$\begin{gathered} u^1 < v^1 < u^2 < v^2 \\ v^1 < u^1 < v^2 < u^2 \\ u^1 < u^2 < w^1 < w^2 \\ v^1 < v^2 < u^1 < u^2 \end{gathered}$$

See Figure 3, and notice that the arcs are not nested.

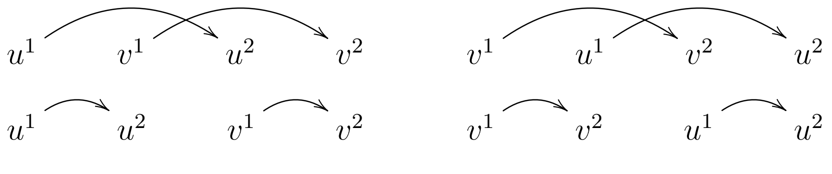

Figure 3: The four ways in which the edge (u, v) can be represented in w_G.

Case $(v, w) \notin E$**:** Exactly one the following two conditions on the indices $u^1 < u^2$ and $v^1 < v^2$ holds:

$$u^1 < v^1 < v^2 < u^2$$
$$v^1 < u^1 < u^2 < v^2$$

See Figure 4, and notice that the arcs are nested.

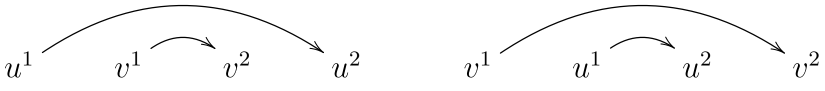

Figure 4: The two ways in which the two nodes (u_1, u_2) and (v_1, v_2) can be represented in w_G as *not* being connected.

The idea, as is apparent from Figures 3 and 4, is that vertices which are not connected in G have *nested* arcs in w_G — each vertex corresponds to an arc. On the other hand, when two vertices are connected, their arcs are not nested, which can happen in the four ways depicted in Figure 3. In essence, two vertices $u, v \in V$ are connected if and only if uu, vv are properly shuffled in the string that represents the graph.

Many classes of graphs can be represented with strings. For example, a clique $G = (V, E)$ can be represented with

$$w_G = (v_1 v_2 \dots v_n)(v_1 v_2 \dots v_n),$$

and an independent set $G = (V, E)$ can be represented as

$$w_G = (v_1 v_2 \dots v_n)(v_1 v_2 \dots v_n)^R,$$

where $(w_1 w_2 \dots w_m)^R = w_m w_{m-1} \dots w_1$, i.e., the reverse of a strings. In fact, all graphs of up to four vertices can be represented with strings. However, there are graphs that cannot be represented with a string, for example the cycle on 5 vertices shown in Figure 5.

The complement of a graph $G = (V, E)$, denoted as $G^c = (V, E^c)$ is a graph obtained from G such that, $e \in E$ if and only if $e \notin E^c$. The transitive closure of a graph $G = (V, E)$, is the graph $G' = (V, E')$, such that, E' is the smallest superset of E with the following property: if $(v, w), (w, u)$ are two edges in E', then $(v, u) \in E'$.

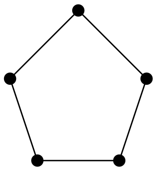

Figure 5: The Hamiltonian cycle on five vertices is the smallest graph without a string representation.

Let the product notation $\Pi_{i=1}^{n} x_i$ denote the concatenation $x_1 \cdot x_2 \cdot \ldots \cdot x_n$. Given a string w of even length, $|w| = n = 2k$, we define $l(w)$ to be the left half of w and $r(w)$ to be the right half of w, i.e., $l(w) = w_1 w_2 \ldots w_k$ and $r(w) = w_{k+1} w_{k+2} \ldots w_n$. Thus, $w = l(w)r(w)$, i.e., w is the concatenation of its left and right halves.

The next lemma shows that a large family of graphs can be represented with strings.

Lemma 7. *If G is a graph such that either G or G^c is isomorphic to a transitive closure of a tree, then G can be represented with a string w_G.*

Proof: A rooted tree is a tree with the root singled out. Thus, a rooted tree is represented by a pair T, R where T is the tree, and R is the root. Given any node v in T, we let T_v be the subtree of T rooted at v. In particular, the whole tree can be represented as T_R. We view our rooted trees as implicitly directed, where the direction is from the root to the leaves.

Let T be a rooted tree and consider its representation in Figure 6. Here R is the root of T, and $R_1, R_2, \ldots, R_n$ are the children of R, with their respective subtrees $T_{R_1}, T_{R_2}, \ldots, T_{R_n}$. Note that since the tree is considered to be directed, R is (for example) connected to every node in T_{R_1}, but node R_1 is not connected to node R_2.

Let $\hat{T}$ represent the transitive closure of T. Thus, $\hat{T}$ has all the edges of T plus given any node v in T, v has an edge to every node in T_v (except itself). In particular, following the naming in Figure 6, R is connected to every node in T by an edge, and R_i is connected to every node in T_{R_i} by an edge, and so on. Note that if T were not directed, its transitive closure would simply be the clique on its nodes. Since it is directed, a node is connected to any other node on a path from it to some leaf. However, once all edges are computed their direction is dropped, and so $\hat{T}$ is undirected.

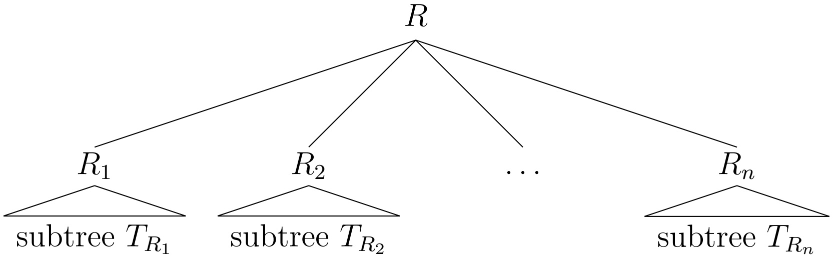

Figure 6: Tree T rooted at R.

We are going to show that given an undirected graph G such that either G or G^c is isomorphic to some $\hat{T}$, the graph G can be represented with a string w_G. We show how to construct w_G over Σ_V. Note that we work with three graphs: $G, T, \hat{T}$, where G is undirected, T is implicitly directed, and $\hat{T}$ is obtained from T by a transitive closure, and then the directions on the edges of $\hat{T}$ are dropped.

Suppose that G is isomorphic to some $\hat{T}$, and throughout the construction keep in mind Figure 6. We build $w_G = w_{\hat{T}}$ inductively, and our procedure maintains the following property for each subtree: each subtree T_i (i.e., the subtree rooted at R_i) has a corresponding $w_{\hat{T}_i}$ such that $\Sigma_{l(w_{\hat{T}_i})} = \Sigma_{r(w_{\hat{T}_i})}$. Thus, exactly half of the symbols of $w_{\hat{T}_i}$ are in the left half (and therefore, exactly half of the symbols are in the right half of $w_{\hat{T}_i}$). Note that the symbols do *not* necessarily occur in the same order in the two halves.

We are going to construct $w_G = w_{\hat{T}}$ by structural induction on T. In the basis case T consists of a single node R, and $w_{\hat{T}} = RR$. Note that the property $l(w_{\hat{T}}) = l(RR) = R = r(RR) = r(w_{\hat{T}})$ is maintained. In the inductive step we build $w_{\hat{T}}$ as follows:

$$w_G = w_{\hat{T}} = R\left[\Pi_{i=1}^{n} l(w_{\hat{T}_i})\right] R \left[\Pi_{i=n}^{1} r(w_{\hat{T}_i})\right]. \tag{3}$$

Note that $\Sigma_{l(w_{\hat{T}})} = \Sigma_{r(w_{\hat{T}})}$, since inductively, for all i, $\Sigma_{l(w_{\hat{T}_i})} = \Sigma_{r(w_{\hat{T}_i})}$, and one copy of R is in $l(w_{\hat{T}})$ and the other in $r(w_{\hat{T}})$. Also note that the product inside the right square brackets of (3) is given in reverse order.

Note that the first occurrence of R is to the left of all other symbols, and the second occurrence of R falls in the middle of all the "subtrees"; thus any arc associated with any vertex in the subtree rooted at R overlaps in the right way with the R-arc. On the other hand, the subtrees associated with each $R_1, R_2, \dots, R_n$ are such that any two arcs from different subtrees are nested. See Figure 7.

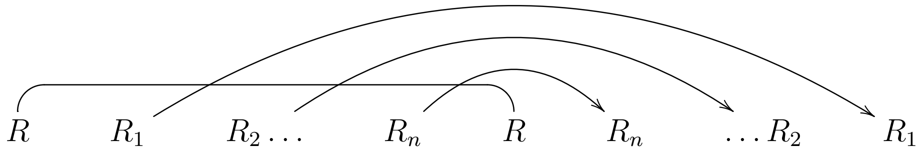

Figure 7: Recursive construction of w_G. Note that in the left half the R_i's are given in increasing order, and in the right half they are given in decreasing order.

We now give a small example of the construction. Consider the tree T in Figure 8. We are going to construct the corresponding $w_{\hat{T}}$. We start with the tree rooted at R_1, where $w_{\hat{T}_1} = R_1R_{11}R_{12}R_1R_{12}R_{11}$, and $w_{\hat{T}_2} = R_2R_2$. We now combined, inductively, the two sub-trees rooted at R_1 and R_2 into one tree rooted at R:

$$\begin{aligned} w_{\hat{T}} &= Rl(w_{\hat{T}_1})l(w_{\hat{T}_2})Rr(w_{\hat{T}_2})r(w_{\hat{T}_1}) \\ &= RR_1R_{11}R_{12}R_2RR_2R_1R_{12}R_{11} \end{aligned}$$

and now note that, for example, R_2 and R_{12} are not connected in $\hat{T}$, and so arc $(5,7)$ is nested under arc $(4,9)$, where the numbers indicate the position of the corresponding symbol in $w_{\hat{T}}$. On the other hand, R is connected to R_{11} in $\hat{T}$, and so are arcs $(3,10)$ and $(1,6)$ are not nested.

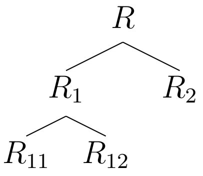

Figure 8: Example of a small tree T.

Suppose now that G^c, where an edge is in G^c if and only if it is not in G, is isomorphic to $\hat{T}$. The basis case, where G^c consists of a single node, is the same as in the first case (G is isomorphic to $\hat{T}$). The difference is in the inductive step.

Let $w_{\hat{T}_1^c}, w_{\hat{T}_2^c}, \ldots, w_{\hat{T}_n^c}$ be strings that correspond to the following subtrees: $\hat{T}_1^c, \hat{T}_2^c, \ldots, \hat{T}_n^c$, and hence they represent faithfully the connections in the corresponding vertices in G. We now complete the construction of w_G by adding the root R, so that:

$$w_G = w_{\hat{T}^c} = R(\Pi_{i=1}^n w_{\hat{T}_i^c})R.$$

To see why this works note that if G^c is isomorphic to $\hat{T}$, then G is isomorphic to $\hat{T}^c$, and any arc in $\Pi_{i=1}^{n} w_{\hat{T}_i^c}$ is nested under the arc between the R's at the ends of the string. This means that R is not connected to any of the nodes in $T_1, T_2, \ldots, T_n$. □

5. Open problems

It follows from the results of [1] that shuffle can be decided in $\mathbf{SAC}^1$. Can shuffle be decided in $\mathbf{NC}^1$? If shuffle were in $\mathbf{NC}^1$ it would mean that shuffle can be decided with a polysize family of Boolean formulas, which would be a very interesting result. Does the converse of Lemma 7 hold? Can we characterize all graphs with shuffle on strings where more repetitions of symbols are allowed? Also regarding Lemma 7, can we test in polynomial time whether a given graph or its complement are isomorphic to the transitive closure of a tree?

6. Acknowledgments

The authors are grateful to Bill Smyth, Franya Franek and Dragan Rakas for comments on this paper, and especially to Bill Smyth for introducing the second author to the "shuffle problem". This paper has benefited greatly from the insights of anonymous referees.

References

[1] A. Mansfield, An algorithm for a merge recognition problem, Discrete Applied Mathematics 4 (1982) 193–197.

[2] M. Soltys, Circuit complexity of shuffle, in: T. Lecroq, L. Mouchard (Eds.), International Workshop on Combinatorial Algorithms 2013, volume 8288 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 402—411.

[3] S. R. Buss, M. Soltys, Unshuffling a square is NP-hard, Journal of Computer and System Sciences 80 (2013) 766–776.

[4] S. Ginsburg, E. Spanier, Mapping of languages by two-tape devices, Journal of the Association of Computing Machinery 12 (1965) 423–434.

[5] W. E. Riddle, A method for the description and analysis of complex software systems, SIGPLAN Notices 8 (1973) 133–136.

[6] W. E. Riddle, An approach to software system modelling and analysis, Computer Languages 4 (1979) 49–66.

[7] A. C. Shaw, Software descriptions with flow expressions, IEEE Transactions on Software Engineering SE-4 (1978) 242–254.

[8] J. Gischer, Shuffle languages, Petri nets, and context-sensivite grammars, Communications of the ACM 24 (1981) 597–605.

[9] H. Gruber, M. Holzer, Tight bounds on the descriptional complexity of regular expressions, in: Proc. Intl. Conf. on Developments in Language Theory (DLT), Springer Verlag, 2009, pp. 276–287.

[10] M. Jantzen, The power of synchronizing operations on strings, Theoretical Computer Science 14 (1981) 127–154.

[11] M. Jantzen, Extending regular expressions with iterated shuffle, Theoretical Computer Science 38 (1985) 223–247.

[12] J. Jedrzejowicz, Structural properties of shuffle automata, Grammars 2 (1999) 35–51.

[13] J. Jedrzejowicz, A. Szepietowski, Shuffle languages are in P, Theoretical Computer Science 250 (2001) 31=53.

[14] J. Jedrzejowicz, A. Szepietowski, On the expressive power of the shuffle operator matched with intersection by regular sets, Theoretical Informatics and Applications 35 (2005) 379–388.

[15] A. J. Mayer, L. J. Stockmeyer, The complexity of word problems — this time with interleaving, Information and Computation 115 (1994) 293–311.

[16] W. F. Ogden, W. E. Riddle, W. C. Rounds, Complexity of expressions allowing concurrency, in: Proc. 5th ACM Symposium on Principles of Programming Languages (POPL), 1978, pp. 185–194.

[17] T. Shoudai, A P-complete language describable with iterated shuffle, Information Processing Letters 41 (1992) 233–238.

[18] A. Mansfield, On the computational complexity of a merge recognition problem, Discrete Applied Mathematics 1 (1983) 119–122.

[19] M. K. Warmuth, D. Haussler, On the complexity of iterated shuffle, Journal of Computer and System Sciences 28 (1984) 345–358.

[20] R. Rizzi, S. Vialette, On recognizing words that are squares for the shuffle product, in: Computer Science – Theory and Applications, 8th International Computer Science Symposium in Russia, CSR, Lecture Notes in Computer Science 7913, 2013, pp. 235–245.

[21] K. Reinhardt, E. Allender, Making nondeterminism unambiguous., SIAM Journal on Computing 29 (2000) 1118–1131.

[22] G. Herman, M. Soltys, Unambiguous functions in logarithmic space, Fundamenta Informaticae 114 (2012) 129–147.

[23] I. H. Sudborough, On the tape complexity of deterministic context-free languages, Journal of the Association of Computing Machinery 25 (1978) 405–415.

[24] H. Venkateswaran, Properties that characterize LOGCFL, Journal of Computer and System Science 43 (1991) 380–404.

[25] C. H. Papadimitriou, Computational Complexity, Addison-Wesley, 1994.

[26] C. Bourke, R. Tewari, N. V. Vinodchandran, Directed planar reachability is in unambiguous log-space, ACM Trans. Comput. Theory 1 (2009) 4:1–4:17.

[27] E. Allender, D. A. M. Barrington, T. Chakraborty, S. Datta, S. Roy, Planar and grid graph reachability problems, Theory of Computing Systems 45 (2009) 675–723.

[28] M. Furst, J. B. Saxe, M. Sipser, Parity, circuits, and the polynomial-time hierarchy, Math. Systems Theory 17 (1984) 13–27.

[29] U. Schöning, R. Pruim, Gems of Theoretical Computer Science, Springer, 1995.

[30] M. Soltys, An introduction to computational complexity, Jagiellonian University Press, 2009.

[31] S. A. Cook, A taxonomy of problems with fast parallel algorithms, Information and Computation 64 (1985) 2–22.

[32] S. R. Buss, P. N. Yianilos, Linear and $O(n \log n)$ time minimum-cost matching algorithms for quasi-convex tours, SIAM J. Comput. 27 (1998) 170–201.

[33] R. E. Burkhard, E. Çela, G. Rote, G. J. Woeginger, The quadratic assignment problem with a monotone anti-Monge and a symmetric Toeplitz matrix: Easy and hard cases, Mathematical Programming 82 (1998) 125–158.