Circuit complexity of shuffle

Michael Soltys[⋆]

McMaster University
Dept. of Computing & Software
1280 Main Street West
Hamilton, Ontario L8S 4K1, CANADA
soltys@mcmaster.ca

Abstract. We show that Shuffle(x, y, w), 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 [FSS84], while the fact that it is in $\mathbf{AC}^1$ is implicit in the results of [Man82a]. Together, the two results provide a strong complexity bound for this combinatorial problem.

Keywords: String shuffle, circuit complexity, lower bounds

1 Introduction

Given three strings over the binary alphabet, it is a natural question to ask whether the third string can be composed from a "shuffle" of the first two. That is, can we compose the third string by weaving together the first two, while preserving the order within each string? For example, given 000, 111, and 010101, we can obviously answer in the affirmative. [Man82a] shows that a clever dynamic programming algorithm can determine Shuffle(x, y, w) in time $O(|w|^2)$, and the same paper poses the question of determining a lower bound.

In this paper we show a strong 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. In the nomenclature of circuit complexity this can be stated as follows: Shuffle $\notin \mathbf{AC}^0$ but Shuffle $\in \mathbf{AC}^1$, which provides a good characterization of the circuit complexity of shuffle.

As a side remark, we also show a lower bound for shuffle on single-tape Turing machines; for this model of computation, we can show that a number of steps in the order of $\Omega(n^2)$ is *necessary* to solve the problem. Both lower bounds, the one in terms of bounded depth circuits, and the one in terms of single tape Turing machines, are obtained by reductions. In the former case, the reduction is from the "parity problem" and in the latter case, the reduction is from the "palindromes problem."

[⋆] Supported in part by the NSERC Discovery Grant

This paper is structured as follows: in section 2 we give the background on circuit complexity; in section 3 we give an upper bound and in section 4 we give the lower bound on the complexity of shuffle. The bounds are summarized in Theorem 1 in the conclusion, and we finish with some open problems.

Formal definition of shuffle 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 $x = x_1 x_2 \cdots x_k$ and $y = y_1 y_2 \cdots y_k$ and $w = x_1 y_1 x_2 y_2 \cdots x_k y_k$. Note that $|w| = |x| + |y|$ is a necessary condition for the existence of a shuffle. Also note that [Man82a] gives a different but equivalent definition.

A shuffle is sometimes instead 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. In this paper we assume the binary alphabet, i.e., $x, y, w \in \{0,1\}^*$.

The predicate $\mathrm{Shuffle}(x, y, w)$ holds iff w is a shuffle of x, y, as described in the above paragraph. We are going to present circuits and Turing machines that compute the $\mathrm{Shuffle}(x, y, w)$, and so they must take the three binary strings x, y, w as input. We let $\langle x, y, w \rangle$ denote the encoding of the three strings; this encoding can be just a concatenation of the strings, so that for a properly formed input, where $|x| = |y| = n$, we have $|\langle x, y, w \rangle| = 4n$, and the n can be extracted by the machine. We can also use demarcation of the strings, by encoding 0 with 00, and 1 with 01, and use 11 as separators. In that case, a well formed input where $|x| = m, |y| = n, |w| = m + n$, would be such that $|\langle x, y, w \rangle| = 2m + 1 + 2n + 1 + 2(m + n)$. The point is that it does not matter how we do it, as long as we do it "reasonably."

History Following the presentation of the history of shuffle in [BS13], 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 [GS65]. Early research with applications to concurrent processes can be found in Riddle [Rid73,Rid79] and Shaw [Sha78]. A number of authors, including [Gis81,GH09,Jan81,Jan85,Jed99,JS01,JS05,MS94,ORR78,Sho02] 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 [Man82b,Man83] and Warmuth and Haussler [WH84] studied the computational complexity of the shuffle operator on its own. The paper [Man82b] gave a polynomial time dynamic programming algorithm for deciding $\mathrm{Shuffle}(x, y, w)$. In [Man83], 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 [Man83] 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 [WH84] gave an independent proof of this last 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 [BS13] we show that square shuffle, i.e., the problem of determining whether some string w is a shuffle of some x with itself, that is, $\exists x \text{Shuffle}(x, x, w)$, is NP-hard.

2 Background on complexity

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 $1, 0$ (or T, F), 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 1, 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 1, 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 1 (and $\wedge, \vee$ have fan-in 2, and $\neg$ has fan-in 1).

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 $\forall x \in \{0,1\}^*$, $x \in P$ iff $C_{|x|}(x) = 1$. In order to make this more concrete, note that in the case of shuffle, the family C computes it if

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

Note that $|\langle x, y, w\rangle|$ only depends on the length of the inputs x, y, w and so the same circuit decided all the inputs of a given fixed length.

Let $\mathbf{P}$/poly be the class of all those predicates which have polysize circuits. 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 of circuits C *uniform*. All our circuit results hold with and without the condition of uniformity.

Those predicates (or Boolean functions) that can be decided 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. Recall that $\mathbf{NC}^1 \subseteq \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{NC}^2$, where $\mathbf{L}$ and $\mathbf{NL}$ are deterministic and non-deterministic logarithmic space, respectively. It is not known whether any of these containments are strict. For more details see any complexity textbook; for example [Pap94,Sip06,Sol09].

3 Upper bound

In this section we show 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 [Man82a] and the complexity result of [Sud78,Ven91] which shows that $\mathbf{NL} \subseteq \mathbf{SAC}^1$.

In order to show that Shuffle $\in \mathbf{NL}$, we show that shuffle can be reduced (in low complexity) to the graph reachability problem. We start with an 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; there is a unique path from $(0,0)$ to $(3,3)$. On the right, there are several possible shuffles — in fact, eight of them, each corresponding to a distinct path from $(0,0)$ to $(3,3)$.

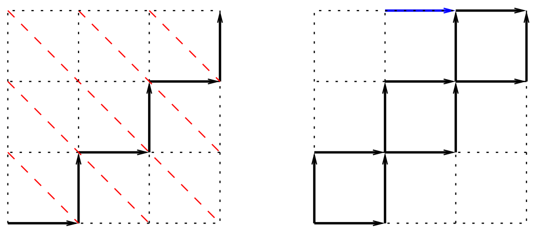

Fig. 1. On the left we have a shuffle of 000 and 111 that yields 010101, and on the right we have a shuffle of 011 and 011 that yields 001111. The dynamic programming algorithm in [Man82a] computes partial solutions along the red diagonal lines.

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 dynamic programming algorithm proposed in [Man82a] works by reducing shuffle 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). For any $i \leq n$ and $j \leq n$, we have edge

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

Note that both edges may be present, which is what introduces the element of non-determinism.

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.

Thus Shuffle $\in$ **NL**. $\square$

Corollary 1 Shuffle $\in$ $\mathbf{SAC}^1$

Proof. Since $\mathbf{NL} \subseteq \mathbf{SAC}^1$ (see [Sud78,Ven91]) it follows directly from Lemma 1 that Shuffle $\in \mathbf{SAC}^1 \subseteq \mathbf{AC}^1$. $\square$

Note that since the graph resulting from the shuffle reduction is planar, it follows that Shuffle $\in$ **UL**. [BTV07] shows that directed planar reachability is in the class **UL** (for "Unambiguous Logarithmic Space"; this class is just like **NL** except that we can design a non-deterministic log-space bounded machine such that "no" instances of the problem have no accepting paths, while "yes" instances have exactly one accepting path).

Note that Shuffle $\in$ **UL** does not mean that there is a unique path from $(0,0)$ to (n,n); rather, there exists a machine deciding the problem in log-space such that, while the machine is non-deterministic, at most one computational path accepts. Further, the planar graph in the proof of Lemma 1 is layered — and such graphs have been studied in [ABC$^+$09]. It would be interesting to know whether the results contained therein could improve the upper bound for shuffle. In particular, can this be used to show that Shuffle $\in \mathbf{NC}^1$, and hence shuffle can be decided with a polysize family of Boolean *formulas*? The fact that $\mathbf{NC}^1$ circuits are computationally equivalent to Boolean formulas follows from Spira's theorem (see [Sol09, Theorem 6.3]).

4 Lower bound

In this section we show a circuit lower bound for shuffle — that is, we show that Shuffle $\notin \mathbf{AC}^0$, 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. This result relies on the seminal complexity result showing that parity is not in $\mathbf{AC}^0$, due to [FSS84]. A very accessible presentation of this result can be found in [SP95, Chapters 11 & 12]; many of the details of that presentation are made explicit in [Sol09, section 5.3]. We also show that shuffle requires $\Omega(n^2)$ steps on a single-tape Turing machines.

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

Intuitively, what we claim is the following: suppose that we have a "black-box" that takes $\langle x, y, w\rangle$ as input bits and computes $\text{Shuffle}(x, y, w)$. 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 in Lemma 2 below and in Figure 2.

Lemma 2 Parity $\in \mathbf{AC}^0[\text{Shuffle}]$.

Proof. In order to compute the parity of x, run the following algorithm: for all odd $i \in \{0, \ldots, |x|\}$, check if $\text{Shuffle}(0^{|x|-i}, 1^i, x)$ 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:

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

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

Corollary 2 Shuffle $\notin \mathbf{AC}^0$.

Proof. Since by [FSS84] Parity $\notin \mathbf{AC}^0$, and by Lemma 2 we know that parity $\mathbf{AC}^0$-reduces to shuffle, it follows that shuffle is not in $\mathbf{AC}^0$. □

Turing machine lower bound The string x is a palindrome if it reads the same backward as forward. If x^R is the reverse of x, i.e., $x^R = x_n x_{n-1} \ldots x_1$, then x is a palindrome if and only if $x = x^R$. It is a folklore result in complexity that given a single tape Turing machine as the model of computation, testing for palindromes

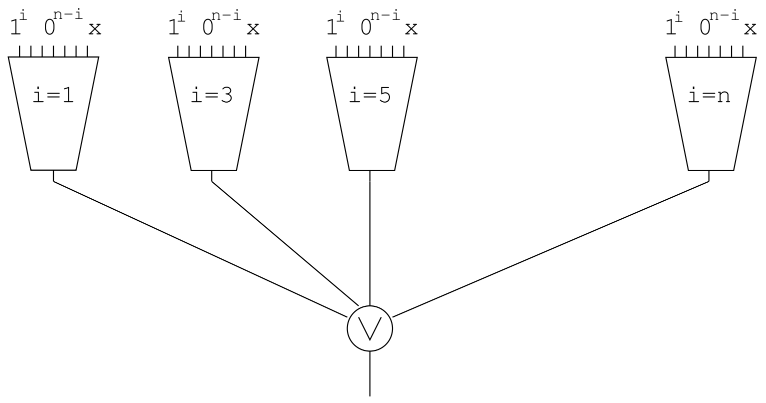

Fig. 2. Parity of x computed in terms of Shuffle; note that we assume that n is odd in this Figure. If n were even the last "black box" for shuffle would be for $i = n - 1$.

requires $\Omega(|x|^2)$ many steps. This result uses Kolmogorov complexity and the "crossing sequences technique." The interested reader can check, for example, [SP95, Chapter 9] or [Sol09, §1.3].

We can use this lower bound for palindromes in order to show that shuffle also requires $\Omega(n^2)$ many steps on a single tape Turing machine. Let Palindrome(x) be the eponymous predicate and note that we can use shuffle to express that a string is a palindrome as follows:

$$\text{Palindrome}(x) \iff \text{Shuffle}(\varepsilon, x, x^R). \tag{2}$$

As the first string is empty, shuffle will hold iff $x_i = x_{n+1-i}$, for $i \in [n]$, which is true iff $x = x^R$.

Lemma 3 Shuffle *takes $\Omega(n^2)$ many steps on a single-tape Turing machine.*

Proof. Suppose that a single-tape Turing machine can decide, on input $\langle x, y\rangle$ whether Shuffle(ε, x, y). Then, the same machine can decide on input

$$\langle w_1 \dots w_{\lfloor \frac{n}{2} \rfloor}, w_n \dots w_{\lceil \frac{n}{2} \rceil + 1}\rangle$$

whether w, where $n = |w|$, is a palindrome. As palindromes require $\Omega(n^2)$ steps (in the worst-case), so does Shuffle(ε, x, y). As Shuffle(ε, x, y) is a special case of the general shuffle problem, the Lemma follows. □

Other reductions to shuffle It is interesting that several different string predicates reduce to shuffle in a natural way. We have (1) which gives a reduction of parity to shuffle; we have that equality of strings reduces to shuffle: $x = y \iff$ Shuffle(ε, x, y); we have (2) which shows that palindromes reduce to shuffle.

We end by showing that concatenation reduces to shuffle. Let p_0, p_1 be "padding" functions on strings defined as follows:

$$\begin{aligned} p_0(x) &= p_0(x_1x_2\dots x_n) = 00x_100x_200\dots 00x_n00 \\ p_1(x) &= p_1(x_1x_2\dots x_n) = 11x_111x_211\dots 11x_n11 \end{aligned}$$

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. Now note that

$$w = u{\cdot}v \iff \text{Shuffle}(p_0(u), p_1(v), p_0(w_1w_2\dots w_{|u|}){\cdot}p_1(w_{|u|+1}w_{|u|+2}\dots w_{|u|+|v|})),$$

where "$\cdot$" denotes concatenation of strings. The direction "$\Rightarrow$" is easy to see; for direction "$\Leftarrow$" we use the following notation:

$$\begin{aligned} r &= p_0(u) = 00u_100\dots \\ s &= p_1(v) = 11v_111\dots \\ t &= p_0(w_1w_2\dots w_{|u|}) \cdot p_1(w_{|u|+1}w_{|u|+2}\dots w_{|u|+|v|}) = 00w_100\dots \end{aligned}$$

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 $w = u \cdot v$.

5 Conclusion

Putting everything together we have the following Theorem.

Theorem 1 Shuffle $\notin$ $\mathbf{AC}^0$, *but* Shuffle $\in \mathbf{SAC}^1 \subseteq \mathbf{AC}^1$. *Also, shuffle requires $\Omega(n^2)$ many steps on a single tape Turing machine.*

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 [Coo85] for a discussion of $\mathbf{NC}^2$.

6 Open problems

It follows from the results of [Man82a] that shuffle can be decided in $\mathbf{SAC}^1$. Can shuffle be decided in $\mathbf{NC}^1$? We know that $\mathbf{NC}^1 \subseteq \mathbf{SAC}^1 \subseteq \mathbf{NC}^2$, and $\mathbf{SAC}^1$

is almost the same as $\mathbf{NC}^1$ except that $\mathbf{SAC}^1$ allows unbounded fan-in for the $\vee$-gates (and bounded fan-in for the $\wedge$-gates), whereas $\mathbf{NC}^1$ has bounded fan-in for all gates. 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.

7 Acknowledgments

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

References

[ABC$^+$09] Eric Allender, David A. Mix Barrington, Tanmoy Chakraborty, Samir Datta, and Sambuddha Roy. Planar and grid graph reachability problems. *Theory of Computing Systems*, 45:675–723, 2009.

[BS13] Samuel R. Buss and Michael Soltys. Unshuffling a square is NP-hard. Submitted for publication, March 2013.

[BTV07] Chris Bourke, Raghunath Tewari, and N. V. Vinodchandran. Directed planar reachability is in unambiguous log-space. In *Proceedings of IEEE Conference on Computational Complexity CCC*, 2007.

[Coo85] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Computation*, 64(13):2–22, 1985.

[FSS84] M. Furst, J. B. Saxe, and M. Sipser. Parity, circuits, and the polynomial-time hierarchy. *Math. Systems Theory*, 17:13–27, 1984.

[GH09] Hermann Gruber and Markus Holzer. Tight bounds on the descriptional complexity of regular expressions. In *Proc. Intl. Conf. on Developments in Language Theory (DLT)*, pages 276–287. Springer Verlag, 2009.

[Gis81] J. Gischer. Shuffle languages, petri nets, and context-sensivite grammars. *Communications of the ACM*, 24(9):597–605, 1981.

[GS65] Seymour Ginsburg and Edwin Spanier. Mappings of languages by two-tape devices. *Journal of the A.C.M.*, 12(3):423–434, 1965.

[Jan81] Matthias Jantzen. The power of synchronizing operations on strings. *Theoretical Computer Science*, 14:127–154, 1981.

[Jan85] Matthias Jantzen. Extending regular expressions with iterated shuffle. *Theoretical Computer Science*, 38:223–247, 1985.

[Jed99] Joanna Jedrzejowicz. Structural properties of shuffle automata. *Grammars*, 2(1):35–51, 1999.

[JS01] Joanna Jedrzejowicz and Andrzej Szepietowski. Shuffle languages are in P. *Theoretical Computer Science*, 250(1-2):31=53, 2001.

[JS05] Joanna Jedrzejowicz and Andrzej Szepietowski. On the expressive power of the shuffle operator matched with intersection by regular sets. *Theoretical Informatics and Applications*, 35:379–388, 2005.

[Man82a] Anthony Mansfield. An algorithm for a merge recognition problem. *Discrete Applied Mathematics*, 4(3):193–197, June 1982.

[Man82b] Anthony Mansfield. An algorithm for a merge recognition problem. *Discrete Applied Mathematics*, 4(3):193–197, June 1982.

[Man83] Anthony Mansfield. On the computational complexity of a merge recognition problem. *Discrete Applied Mathematics*, 1(3):119–122, January 1983.

[MS94] Alain J. Mayer and Larry J. Stockmeyer. The complexity of word problems — this time with interleaving. *Information and Computation*, 115:293–311, 1994.

[ORR78] W. F. Ogden, W. E. Riddle, and W. C. Rounds. Complexity of expressions allowing concurrency. In *Proc. 5th ACM Symposium on Principles of Programming Languages (POPL)*, pages 185–194, 1978.

[Pap94] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[Rid73] William E. Riddle. A method for the description and analysis of complex software systems. *SIGPLAN Notices*, 8(9):133–136, September 1973.

[Rid79] William E. Riddle. An approach to software system modelling and analysis. *Computer Languages*, 4(1):49–66, 1979.

[Sha78] Alan C. Shaw. Software descriptions with flow expressions. *IEEE Transactions on Software Engineering*, SE-4(3):242–254, 1978.

[Sho02] Takayoshi Shoudai. A P-complete language describable with iterated shuffle. *Information Processing Letters*, 41(5):233–238, 1002.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation.* Thompson, 2006. Second Edition.

[Sol09] Michael Soltys. *An introduction to computational complexity.* Jagiellonian University Press, 2009.

[SP95] Uwe Schöning and Randall Pruim. *Gems of Theoretical Computer Science.* Springer, 1995.

[Sud78] I. H. Sudborough. On the tape complexity of deterministic context-free languages. *Journal of the Association of Computing Machinery*, 25:405–415, 1978.

[Ven91] H. Venkateswaran. Properties that characterize LOGCFL. *Journal of Computer and System Science*, 43:380–404, 1991.

[WH84] Manfred K. Warmuth and David Haussler. On the complexity of iterated shuffle. *Journal of Computer and System Sciences*, 28(3):345–358, June 1984.