On the Size Complexity of Rotating and Sweeping Automata

Christos Kapoutsis, Richard Královič, and Tobias Mömke

Department of Computer Science, ETH Zürich

Abstract. We examine the succinctness of *one-way*, *rotating*, *sweeping*, and *two-way* deterministic finite automata (1DFAs, RDFAs, SDFAs, 2DFAs). Here, a SDFA is a 2DFA whose head can change direction only on the endmarkers and a RDFA is a SDFA whose head is reset on the left end of the input every time the right endmarker is read. We introduce a list of language operators and study the corresponding closure properties of the size complexity classes defined by these automata. Our conclusions reveal the logical structure of certain proofs of known separations in the hierarchy of these classes and allow us to systematically construct alternative problems to witness these separations.

1 Introduction

One of the most important open problems in the study of the size complexity of finite automata is the comparison between determinism and nondeterminism in the two-way case: Does every two-way nondeterministic finite automaton (2NFA) with n states have a deterministic equivalent (2DFA) with a number of states polynomial in n? [6, 5] Equivalently, if 2N is the class of families of languages that can be recognized by families of polynomially large 2NFAs and 2D is its deterministic counterpart, is it 2D = 2N? The answer is conjectured to be negative, even if all 2NFAs considered are actually one-way (1NFAs). That is, even $\text{2D} \not\supseteq \text{1N}$ is conjectured to be true, where 1N is the one-way counterpart of 2N.

To confirm these conjectures, one would need to prove that some n-state 2NFA or 1NFA requires superpolynomially (in n) many states on every equivalent 2DFA. Unfortunately, such lower bounds for arbitrary 2DFAs are currently beyond reach. They have been established only for certain restricted special cases. Two of them are the *rotating* and the *sweeping* 2DFAs (RDFAs and SDFAs, respectively).

A SDFA is a 2DFA that changes the direction of its head only on the input endmarkers. Thus, a computation is simply an alternating sequence of rightward and leftward one-way scans. A RDFA is a SDFA that performs no leftward scans: upon reading the right endmarker, its head jumps directly to the left end. The subsets of 2D that correspond to these restricted 2DFAs are called SD and RD.

Several facts about the size complexity of SDFAs have been known for quite a while (e.g., $\text{1D} \not\supseteq \text{SD}$ [7], $\text{SD} \not\supseteq \text{2D}$ [7, 1, 4], $\text{SD} \not\supseteq \text{1N}$ [7], $\text{SD} \not\supseteq \text{1N} \cap \text{co-1N}$ [3]) and, often, at the core of their proofs one can find proofs of the corresponding

*Work supported by the Swiss National Science Foundation grant 200021-107327/1.

facts for RDFAs (e.g., 1D $\not\supseteq$ RD, RD $\not\supseteq$ 2D, etc.). Overall, though, our study of these automata has been fragmentary, exactly because they have always been examined only on the way to investigate the 2D vs. 2N question.

In this article we take the time to make the hierarchy 1D $\subseteq$ RD $\subseteq$ SD $\subseteq$ 2D itself our focus. We introduce a list of language operators and study the closure properties of our complexity classes with respect to them. Our conclusions allow us to reprove the separation of [3], this time with a new witness language family, which (i) is constructed by a sequence of applications of our operators to a single, minimally hard, 'core' family and (ii) mimicks the design of the original witness. This uncovers the logical structure of the original proof and explains how hardness propagates upwards in our hierarchy of complexity classes when appropriate operators are applied. It also enables us to construct many other witnesses of the same separation, using the same method but a different sequence of operators and/or a different 'core' family. Some of these witnesses are both *simpler* (produced by operators of lower complexity) and *more effective* (establish a greater exponential gap) than the one of [3].

More generally, our operators provide a systematic way of proving separations by building witnesses out of simpler and easier 'core' language families. For example, given any family $\mathcal{L}$ which is hard for 1DFAs reading from left to right (as usual) but easy for 1DFAs reading from right to left ($\mathcal{L} \notin$ 1D but $\mathcal{L}^{\mathrm{R}} \in$ 1D), one can build a family $\mathcal{L}'$ which is hard for SDFAs but easy for 1NFAs, easy for 1NFAs recognizing the complement, and easy for 2DFAs ($\mathcal{L}' \in (1\mathrm{N} \cap \text{co-}1\mathrm{N} \cap 2\mathrm{D}) \setminus \mathrm{SD}$), a simultaneous witness for the theorems of [7, 3, 1, 4]. We believe that this operator-based reconstruction or simplification of witnesses deepens our understanding of the relative power of these automata.

The next section defines the objects that we work with. Section 3 introduces two important tools for working with parallel automata and uses them to prove hardness propagation lemmata. These are then applied in Sect. 4 to establish the hierarchy and closures map of Fig. 1. Section 5 lists our final conclusions.

2 Preliminaries

Let Σ be an alphabet. If $z \in \Sigma^*$ is a string, then $|z|$, z_t, z^t, and z^{R} are its length, t-th symbol (if $1 \leq t \leq |z|$), t-fold concatenation with itself (if $t \geq 0$), and reverse. If $P \subseteq \Sigma^*$, then $P^{\mathrm{R}} := \{z^{\mathrm{R}} \mid z \in P\}$.

A (*promise*) *problem* over Σ is a pair $L = (L_{\mathrm{Y}}, L_{\mathrm{N}})$ of disjoint subsets of Σ^*. The *promise* of L is $L_{\mathrm{P}} := L_{\mathrm{Y}} \cup L_{\mathrm{N}}$. If $L_{\mathrm{P}} = \Sigma^*$, then L is a *language*. If $L_{\mathrm{Y}}, L_{\mathrm{N}} \neq \emptyset$, then L is *nontrivial*. We write $w \in L$ iff $w \in L_{\mathrm{Y}}$, and $w \notin L$ iff $w \in L_{\mathrm{N}}$. (Note that "$x \notin L$" is equivalent to the negation of "$x \in L$" only when $x \in L_{\mathrm{P}}$.) To *solve* L is to accept all $w \in L$ but no $w \notin L$ (and decide arbitrarily on $w \notin L_{\mathrm{P}}$).

A *family of automata* $\mathcal{M} = (M_n)_{n \geq 1}$ solves a *family of problems* $\mathcal{L} = (L_n)_{n \geq 1}$ iff, for all n, M_n solves L_n. The automata of $\mathcal{M}$ are '*small*' iff, for some polynomial p and all n, M_n has at most $p(n)$ states.

Problem operators. Fix a delimiter # and let L, L_1, L_2 be arbitrary problems. If $\#x_1\#\cdots\#x_l\#$ denotes strings from $\#(L_{\text{P}}\#)^*$ and $\#x\#y\#$ denotes strings from $\#(L_1)_{\text{P}}\#(L_2)_{\text{P}}\#$, then the following pairs are easily seen to be problems, too:

$$\begin{aligned}
\neg L &:= (L_{\text{N}}, L_{\text{Y}}) \qquad L^{\text{R}} := (L^{\text{R}}_{\text{Y}}, L^{\text{R}}_{\text{N}}) \\
L_1 \wedge L_2 &:= (\ \{\#x\#y\# \mid x \in L_1 \wedge y \in L_2\},\ \{\#x\#y\# \mid x \notin L_1 \vee y \notin L_2\}\ ) \\
L_1 \vee L_2 &:= (\ \{\#x\#y\# \mid x \in L_1 \vee y \in L_2\},\ \{\#x\#y\# \mid x \notin L_1 \wedge y \notin L_2\}\ ) \\
L_1 \oplus L_2 &:= (\ \{\#x\#y\# \mid x \in L_1 \Leftrightarrow y \notin L_2\},\ \{\#x\#y\# \mid x \in L_1 \Leftrightarrow y \in L_2\}\ ) \\
\bigwedge L &:= (\ \{\#x_1\#\cdots\#x_l\# \mid (\forall i)(x_i \in L)\},\ \{\#x_1\#\cdots\#x_l\# \mid (\exists i)(x_i \notin L)\}\ ) \\
\bigvee L &:= (\ \{\#x_1\#\cdots\#x_l\# \mid (\exists i)(x_i \in L)\},\ \{\#x_1\#\cdots\#x_l\# \mid (\forall i)(x_i \notin L)\}\ ) \\
\bigoplus L &:= (\ \{\#x_1\#\cdots\#x_l\# \mid \text{the number of } i \text{ such that } x_i \in L \text{ is odd}\}, \\
&\qquad\quad \{\#x_1\#\cdots\#x_l\# \mid \text{the number of } i \text{ such that } x_i \in L \text{ is even}\}\ )
\end{aligned} \tag{1}$$

over the promises, respectively: L_{P}, $(L_{\text{P}})^{\text{R}}$, $\#(L_1)_{\text{P}}\#(L_2)_{\text{P}}\#$ (for $L_1 \wedge L_2$, $L_1 \vee L_2$, $L_1 \oplus L_2$) and $\#(L_{\text{P}}\#)^*$ (for the rest). We call these problems, respectively: the *complement* and *reversal* of L; the *conjunctive*, *disjunctive*, and *parity concatenation* of L_1 with L_2; the *conjunctive*, *disjunctive*, and *parity star* of L.

By the definitions, we easily have $\neg(L^{\text{R}}) = (\neg L)^{\text{R}}$, and also:

$$\begin{array}{lll}
\neg(L_1 \wedge L_2) = \neg L_1 \vee \neg L_2 & \neg(\bigwedge L) = \bigvee \neg L & (L_1 \wedge L_2)^{\text{R}} = L_2^{\text{R}} \wedge L_1^{\text{R}} \\
\neg(L_1 \vee L_2) = \neg L_1 \wedge \neg L_2 & \neg(\bigvee L) = \bigwedge \neg L & (L_1 \vee L_2)^{\text{R}} = L_2^{\text{R}} \vee L_1^{\text{R}} \\
\neg(L_1 \oplus L_2) = \neg L_1 \oplus L_2 & (\bigwedge L)^{\text{R}} = \bigwedge L^{\text{R}} & (L_1 \oplus L_2)^{\text{R}} = L_2^{\text{R}} \oplus L_1^{\text{R}}. \\
 & (\bigvee L)^{\text{R}} = \bigvee L^{\text{R}} &
\end{array} \tag{2}$$

Our definitions extend naturally to families of problems: we just apply the problem operator to (corresponding) components. E.g., if $\mathcal{L}, \mathcal{L}_1, \mathcal{L}_2$ are families of problems, then $\neg\mathcal{L} = (\neg L_n)_{n\geq 1}$ and $\mathcal{L}_1 \vee \mathcal{L}_2 = (L_{1,n} \vee L_{2,n})_{n\geq 1}$. Clearly, the identities of (2) remain true when we replace L, L_1, L_2 with $\mathcal{L}, \mathcal{L}_1, \mathcal{L}_2$.

Finite automata. Our automata are *one-way*, *rotating*, *sweeping*, or *two-way*. We refer to them by the naming convention bDFA, where $b = 1, \text{R}, \text{S}, 2$. E.g., RDFAs are rotating (R) deterministic finite automata (DFA). We assume the reader is familiar with all these machines. This section simply fixes some notation.

A SDFA [7] over an alphabet Σ and a set of states Q is a triple $M = (q_s, \delta, q_a)$ of a *start* state $q_s \in Q$, an *accept* state $q_a \in Q$, and a *transition function* δ which partially maps $Q \times (\Sigma \cup \{\vdash, \dashv\})$ to Q, for some *endmarkers* $\vdash, \dashv \notin \Sigma$. An input $z \in \Sigma^*$ is presented to M surrounded by the endmarkers, as $\vdash z \dashv$. The computation starts at q_s and on $\vdash$. The next state is always derived from δ and the current state and symbol. The next position is always the adjacent one in the direction of motion; except when the current symbol is $\dashv$ and the next state is not q_a or when the current symbol is $\vdash$, in which two cases the next position is the adjacent one towards the other endmarker. Note that the computation can either loop, or hang, or fall off $\dashv$ into q_a. In this last case, we say M *accepts* z.

More generally, for any input string $z \in \Sigma^*$ and state p, the *left computation of M from p on z* is the unique sequence $\text{LCOMP}_{M,p}(z) := (q_t)_{1 \le t \le m}$ where: $q_1 := p$; every next state is $q_{t+1} := \delta(q_t, z_t)$, provided that $t \le |z|$ and the value of δ is defined; and m is the first t for which this provision fails. If $m = |z| + 1$, we say the computation *exits z into q_m* or *results in q_m*; otherwise, $1 \le m \le |z|$ and the computation *hangs at q_m* and *results in $\perp$*; the set $Q_\perp := Q \cup \{\perp\}$ contains all possible *results*. The *right computation of M from p on z* is denoted by $\text{RCOMP}_{M,p}(z)$ and defined symmetrically, with $q_{t+1} := \delta(q_t, z_{|z|+1-t})$.

We say M is a RDFA if its next position is decided differently: it is always the adjacent one to the right, except when the current symbol is $\dashv$ and the next state is not q_a, in which case it is the one to the right of $\vdash$.

We say M is a 1DFA if it halts immediately after reading $\dashv$: the value of δ on any state q and on $\dashv$ is always either q_a or undefined. If it is q_a, we say q is a *final* state; if it is undefined, we say q is *nonfinal*. The state $\delta(q_s, \vdash)$, if defined, is called *initial*. If M is allowed more than one next move at each step, we say it is *nondeterministic* (a 1NFA).

Parallel automata. The following additional models will also be useful.

A (*two-sided*) *parallel automaton* ($\text{P}_2$1DFA) [7] is any triple $M = (\mathfrak{L}, \mathfrak{R}, F)$ where $\mathfrak{L} = \{C_1, \ldots, C_k\}$, $\mathfrak{R} = \{D_1, \ldots, D_l\}$ are disjoint families of 1DFAs, and $F \subseteq Q_\perp^{C_1} \times \cdots \times Q_\perp^{C_k} \times Q_\perp^{D_1} \times \cdots \times Q_\perp^{D_l}$, where Q^A is the state set of automaton A. To run M on z means to run each $A \in \mathfrak{L} \cup \mathfrak{R}$ on z from its initial state and record the result, but with a twist: each $A \in \mathfrak{L}$ reads from left to right (i.e., reads z), while each $A \in \mathfrak{R}$ reads from right to left (i.e., reads z^{R}). We say M accepts z iff the tuple of the results of these computations is in F. When $\mathfrak{R} = \emptyset$ or $\mathfrak{L} = \emptyset$, we say M is *left-sided* (a P_L1DFA) or *right-sided* (a P_R1DFA), respectively.

A *parallel intersection automaton* ($\cap_2$1DFA, $\cap_\text{L}$1DFA, or $\cap_\text{R}$1DFA) [5] is a parallel automaton whose F consists of the tuples where *all* results are final states. If F consists of the tuples where *some* result is a final state, the automaton is a *parallel union automaton* ($\cup_2$1DFA, $\cup_\text{L}$1DFA, or $\cup_\text{R}$1DFA) [5]. So, a $\cap_2$1DFA accepts its input iff all components accept it; a $\cup_2$1DFA accepts iff any component does.

We say that a family of parallel automata $\mathcal{M} = (M_n)_{n \ge 1}$ are 'small' if for some polynomial p and all n, each component of M_n has at most $p(n)$ states. Note that this restricts only the *size* of the components—not their *number*.

Complexity classes. The size-complexity class 1D consists of every family of problems that can be solved by a family of small 1DFAs. The classes RD, SD, 2D, $\cap_\text{L}$1D, $\cap_\text{R}$1D, $\cap_2$1D, $\cup_\text{L}$1D, $\cup_\text{R}$1D, $\cup_2$1D, P_L1D, P_R1D, $\text{P}_2$1D, and 1N are defined similarly, by replacing 1DFAs with RDFAs, SDFAs, etc. The naming convention is from [5]; there, however, 1D, 1N, and 2D contain families of *languages*, not problems.(1)

If $\mathcal{C}$ is a class, then re-$\mathcal{C}$ consists of all families of problems whose reversal is in $\mathcal{C}$ and co-$\mathcal{C}$ consists of all families of problems whose complement is in $\mathcal{C}$. Of special interest to us is the class 1N $\cap$ co-1N; we also denote it by 1Δ.

The following inclusions are easy to verify, by the definitions and by [7, Lemma 1], for every side mode $\sigma = {}_{\mathrm{L}}, {}_{\mathrm{R}}, {}_{2}$ and every parallel mode $\pi = \cap, \cup, \mathrm{P}$:

$$\begin{array}{lll} \text{co-}\cap_\sigma\text{1D} = \cup_\sigma\text{1D} & \cap_\sigma\text{1D}, \cup_\sigma\text{1D} \subseteq \mathrm{P}_\sigma\text{1D} & \text{1D} \subseteq \cap_{\mathrm{L}}\text{1D}, \cup_{\mathrm{L}}\text{1D}, \text{RD} \subseteq \mathrm{P}_{\mathrm{L}}\text{1D} \\ \text{re-}\pi_{\mathrm{L}}\text{1D} = \pi_{\mathrm{R}}\text{1D} & \pi_{\mathrm{L}}\text{1D}, \pi_{\mathrm{R}}\text{1D} \subseteq \pi_2\text{1D} & \text{RD} \subseteq \text{SD} \subseteq \mathrm{P}_2\text{1D}, \text{2D}. \end{array} \tag{3}$$

A core problem. Let $[n] := \{1, \ldots, n\}$. All witnesses in Sect. 4 will be derived from the operators of (1) and the following 'core' problem: "Given two symbols describing a set $\alpha \subseteq [n]$ and a number $i \in [n]$, check that $i \in \alpha$." Formally,

$$J_n := (\ \{\alpha i \mid \alpha \subseteq [n]\ \ \&\ \ i \in \alpha\},\ \{\alpha i \mid \alpha \subseteq [n]\ \ \&\ \ i \in \overline{\alpha}\}\ ). \tag{4}$$

Lemma 1. *$\mathcal{J} := (J_n)_{n\geq 1}$ is not in* 1D *but is in* re-1D, 1N, co-1N, $\cap_{\mathrm{L}}$1D, $\cup_{\mathrm{L}}$1D.[(2)]

3 Basic Tools and Hardness Propagation

To draw the map of Fig. 1, we need several lemmata that explain how the operators of (1) can increase the hardness of problems. In turn, to prove these lemmata, we need two basic tools for the construction of hard inputs to parallel automata: the *confusing* and the *generic* strings. We first describe these tools and then use them to prove the hardness propagation lemmata.

Confusing strings. Let $M = (\mathfrak{L}, \mathfrak{R})$ be a $\cap_2$1DFA and L a problem. We say a string y *confuses M on L* if it is a positive instance but some component hangs on it or is negative but every component treats it identically to a positive one:

$$\text{or} \quad \begin{array}{lll} y \in L & \& & (\exists A \in \mathfrak{L} \cup \mathfrak{R})\big(A(y) = \bot\big) \\ y \notin L & \& & (\forall A \in \mathfrak{L} \cup \mathfrak{R})(\exists \tilde{y} \in L)\big(A(y) = A(\tilde{y})\big) \end{array} \tag{5}$$

where $A(z)$ is the result of $\textsc{lcomp}_A(z)$, if $A \in \mathfrak{L}$, or of $\textsc{rcomp}_A(z)$, if $A \in \mathfrak{R}$. It can be shown that, if some y confuses M on L, then M does not solve L. Note, though, that (5) is independent of the selection of final states in the components of M. So, if $\mathfrak{F}(M)$ is the class of $\cap_2$1DFAs that may differ from M only in the selection of final states, then a y that confuses M on L confuses every $M' \in \mathfrak{F}(M)$, too, and thus no $M' \in \mathfrak{F}(M)$ solves L, either. The converse is also true.

Lemma 2. *Let $M = (\mathfrak{L}, \mathfrak{R})$ be a $\cap_2$*1DFA *and L a problem. Then, strings that confuse M on L exist iff no member of $\mathfrak{F}(M)$ solves L.*

Proof. [$\Rightarrow$] Suppose some y confuses M on L. Fix any $M' = (\mathfrak{L}', \mathfrak{R}') \in \mathfrak{F}(M)$. Since (5) is independent of the choice of final states, y confuses M' on L, too. *If $y \in L$:* By (5), some $A \in \mathfrak{L}' \cup \mathfrak{R}'$ hangs on y. So, M' rejects y, and thus fails. *If $y \notin L$:* If M' accepts y, it fails. If it rejects y, then some $A \in \mathfrak{L}' \cup \mathfrak{R}'$ does not accept y. Consider the $\tilde{y}$ guaranteed for this A by (5). Since $A(\tilde{y}) = A(y)$, we know $\tilde{y}$ is also not accepted by A. Hence, M' rejects $\tilde{y} \in L$, and fails again.

[⇐] Suppose no string confuses M on L. Then, no component hangs on a positive instance; and every negative instance is 'noticed' by some component, in the sense that the component treats it differently than all positive instances:

$$\text{and} \quad \begin{array}{l} (\forall y \in L)(\forall A \in \mathfrak{L} \cup \mathfrak{R})\big(A(y) \neq \bot\big) \\ (\forall y \notin L)(\exists A \in \mathfrak{L} \cup \mathfrak{R})(\forall \tilde{y} \in L)\big(A(y) \neq A(\tilde{y})\big). \end{array} \tag{6}$$

This allows us to find an $M' \in \mathfrak{F}(M)$ that solves L, as follows. We start with all states of all components of M unmarked. Then we iterate over all $y \notin L$. For each of them, we pick an A as guaranteed by (6) and, if the result $A(y)$ is a state, we mark it. When this (possibly infinite) iteration is over, we make all marked states nonfinal and all unmarked states final. The resulting $\cap_2$1DFA is our M'.

To see why M' solves L, consider any string y. *If $y \notin L$:* Then our method examined y, picked an A, and ensured $A(y)$ is either $\bot$ or a nonfinal state. So, this A does not accept y. Therefore, M' rejects y. *If $y \in L$:* Towards a contradiction, suppose M' rejects y. Then some component A^* does not accept y. By (6), $A^*(y) \neq \bot$. Hence, $A^*(y)$ is a state, call it q^*, and is nonfinal. So, at some point, our method marked q^*. Let $\hat{y} \notin L$ be the string examined at that point. Then, the selected A was A^* and $A(\hat{y})$ was q^*, and thus no $\tilde{y} \in L$ had $A^*(\tilde{y}) = q^*$. But this contradicts the fact that $y \in L$ and $A^*(y) = q^*$. □

Generic strings [7]. Let A be a 1DFA over alphabet Σ and states Q, and $y, z \in \Sigma^*$. The (*left*) *views of A on y* is the set of states produced on the right boundary of y by left computations of A:

$$\text{LVIEWS}_A(y) := \{q \in Q \mid (\exists p \in Q)[\text{LCOMP}_{A,p}(y) \text{ exits into } q]\}.$$

The (*left*) *mapping of A on y and z* is the partial function

$$\text{LMAP}_A(y, z) : \text{LVIEWS}_A(y) \to Q$$

which, for every $q \in \text{LVIEWS}_A(y)$, is defined only if $\text{LCOMP}_{A,q}(z)$ does not hang and, if so, returns the state that this computation exits into. It is easy to verify that this function is a *partial surjection* from $\text{LVIEWS}_A(y)$ to $\text{LVIEWS}_A(yz)$.[3] This immediately implies Fact 1. Fact 2 is equally simple.[4]

Fact 1 *For all A, y, z as above*: $|\text{LVIEWS}_A(y)| \geq |\text{LVIEWS}_A(yz)|$.

Fact 2 *For all A, y, z as above*: $\text{LVIEWS}_A(yz) \subseteq \text{LVIEWS}_A(z)$.

Now consider any PL1DFA $M = (\mathfrak{L}, \emptyset, F)$ and any problem L which is *infinitely right-extensible*, in the sense that every $u \in L$ can be extended into a $uu' \in L$. By Fact 1, if we start with any $u \in L$ and keep right-extending it ad infinitum into $uu', uu'u'', uu'u''u''', \cdots \in L$ then, from some point on, the corresponding sequence of tuples of sizes $(|\text{LVIEWS}_A(\cdot)|)_{A \in \mathfrak{L}}$ will become constant. If y is any of the extensions after that point, then y satisfies

$$y \in L \quad \& \quad (\forall yz \in L)(\forall A \in \mathfrak{L})\big(|\text{LVIEWS}_A(y)| = |\text{LVIEWS}_A(yz)|\big) \tag{7}$$

and is called L-*generic* (*for M*) *over L*. The next lemma uses such strings.

Lemma 3. *Suppose a* P_L1DFA $M = (\mathfrak{L}, \emptyset, F)$ *solves* $\bigwedge L$ *and* y *is* L*-generic for* M *over* $\bigwedge L$*. Then,* $x \in L$ *iff* $\text{LMAP}_A(y, xy)$ *is total and injective for all* $A \in \mathfrak{L}$*.*

Proof. [$\Rightarrow$] Let $x \in L$. Then $yxy \in \bigwedge L$ (since $y \in \bigwedge L$ and $x \in L$). So, yxy right-extends y inside $\bigwedge L$. Since y is L-generic, $|\text{LVIEWS}_A(y)| = |\text{LVIEWS}_A(yxy)|$, for all $A \in \mathfrak{L}$. Hence, each partial surjection $\text{LMAP}_A(y, xy)$ has domain and codomain of the same size. This is possible only if the function is both total and injective.

[$\Leftarrow$] Suppose each partial surjection $\text{LMAP}_A(y, xy)$ is total and injective. Then it bijects the set $\text{LVIEWS}_A(y)$ into the set $\text{LVIEWS}_A(yxy)$, which is actually a subset of $\text{LVIEWS}_A(y)$ (Fact 2). Clearly, this is possible only if this subset is the set itself. So, $\text{LMAP}_A(y, xy)$ is a permutation π_A of $\text{LVIEWS}_A(y)$.

Now pick $k \geq 1$ so that each π_A^k is an identity, and let $z := y(xy)^k$. It is easy to verify that $\text{LMAP}_A\big(y, (xy)^k\big)$ equals $\text{LMAP}_A(y, xy)^k = \pi_A^k$, and is therefore the identity on $\text{LVIEWS}_A(y)$. This means that, reading through z, the left computations of A do not notice the suffix $(xy)^k$ to the right of the prefix y. So, no A can distinguish between y and z: it either hangs on both or exits both into the same state.(5) Thus, M does not distinguish between y and z, either: it either accepts both or rejects both. But M accepts y (because $y \in \bigwedge L$), so it accepts z. Hence, every #-delimited infix of z is in L. In particular, $x \in L$. □

If $M = (\mathfrak{L}, \mathfrak{R}, F)$ is a $\text{P}_2$1DFA, we can also work symmetrically with right computations and left-extensions: we can define $\text{RVIEWS}_A(y)$ and $\text{RMAP}_A(z, y)$ for $A \in \mathfrak{R}$, derive Facts 1, 2 for $\text{RVIEWS}_A(y)$ and $\text{RVIEWS}_A(zy)$, and define R-generic strings. We can then construct strings, called *generic*, that are simultaneously L- and R-generic, and use them in a counterpart of Lemma 3 for $\text{P}_2$1DFAs:

Lemma 4. *Suppose a* $\text{P}_2$1DFA $M = (\mathfrak{L}, \mathfrak{R}, F)$ *solves* $\bigwedge L$ *and* y *is generic for* M *over* $\bigwedge L$*. Then,* $x \in L$ *iff* $\text{LMAP}_A(y, xy)$ *is total and injective for all* $A \in \mathfrak{L}$ *and* $\text{RMAP}_A(yx, y)$ *is total and injective for all* $A \in \mathfrak{R}$*.*

Hardness Propagation. We are now ready to show how the operators of (1) can allow us to build harder problems out of easier ones.

Lemma 5. *If no* m*-state* 1DFA *can solve problem* L*, then no* $\cap_\text{L}$1DFA *with* m*-state components can solve problem* $\bigvee L$*. Similarly for* $\bigoplus L$*.*

Proof. Suppose no m-state 1DFA can solve L. By induction on k, we prove that no $\cap_\text{L}$1DFA with k m-state components can solve $\bigvee L$ (the proof for $\bigoplus L$ is similar).

If $k = 0$: Fix any such $\cap_\text{L}$1DFA $M = (\mathfrak{L}, \emptyset)$. By definition, $\# \notin \bigvee L$. But M accepts #, because all components do (vacuously, since $\mathfrak{L} = \emptyset$). So M fails.

If $k \geq 1$: Fix any such $\cap_\text{L}$1DFA $M = (\mathfrak{L}, \emptyset)$. Pick any $D \in \mathfrak{L}$ and remove it from M to get $M_1 = (\mathfrak{L}_1, \emptyset) := (\mathfrak{L} - \{D\}, \emptyset)$. By the inductive hypothesis, no member of $\mathfrak{F}(M_1)$ solves $\bigvee L$. So (Lemma 2), some y confuses M_1 on $\bigvee L$.

Case 1: $y \in \bigvee L$. Then some $A \in \mathfrak{L}_1$ hangs on y. Since $A \in \mathfrak{L}$, too, y confuses M as well. So, M does not solve $\bigvee L$, and the inductive step is complete.

Case 2: $y \notin \bigvee L$. Then every $A \in \mathfrak{L}_1$ treats y identically to a positive instance:

$$(\forall A \in \mathfrak{L} - \{D\})(\exists \tilde{y} \in \bigvee L)\big(A(y) = A(\tilde{y})\big). \tag{8}$$

Let M_2 be the single-component $\cap_{\mathrm{L}}$1DFA whose only 1DFA, call it D', is the one derived from D by changing its initial state to $D(y)$. By the hypothesis of the lemma, no member of $\mathfrak{F}(M_2)$ solves L. So (Lemma 2), some x confuses M_2 on L. We claim that $yx\#$ confuses M on $\bigvee L$. Thus, M does not solve $\bigvee L$, and the induction is again complete. To prove the confusion, we examine cases:

Case 2a: $x \in L$. Then $yx\# \in \bigvee L$, since $y \in (\bigvee L)_{\mathrm{P}}$ and $x \in L$. And D' hangs on x (since x is confusing and D' is the only component), thus $D(yx\#) = D'(x\#) = \bot$. So, component D of M hangs on $yx\# \in \bigvee L$. So, $yx\#$ confuses M on $\bigvee L$.

Case 2b: $x \notin L$. Then $yx\# \notin \bigvee L$, because $y \notin \bigvee L$ and $x \in L_{\mathrm{P}}$. And, since x is confusing, D' treats it identically to some $\tilde{x} \in L$: $D'(x) = D'(\tilde{x})$. Then, each component of M treats $yx\#$ identically to a positive instance of $\bigvee L$:

- D treats $yx\#$ as $y\tilde{x}\#$: $D(y\tilde{x}\#) = D'(\tilde{x}\#) = D'(x\#) = D(yx\#)$. And we know $y\tilde{x}\# \in \bigvee L$, because $y \in (\bigvee L)_{\mathrm{P}}$ and $\tilde{x} \in L$.
- each $A \neq D$ treats $yx\#$ as $\tilde{y}x\#$, where $\tilde{y}$ the string guaranteed for A by (8): $A(\tilde{y}x\#) = A(yx\#)$. And we know $\tilde{y}x\# \in \bigvee L$, since $\tilde{y} \in \bigvee L$ and $x \in L_{\mathrm{P}}$.

Overall, $yx\#$ is again a confusing string for M on $\bigvee L$, as required. □

Lemma 6. *If L_1 has no $\cap_{\mathrm{L}}$1DFA with m-state components and L_2 has no $\cap_{\mathrm{R}}$1DFA with m-state components, then $L_1 \vee L_2$ has no $\cap_2$1DFA with m-state components. Similarly for $L_1 \oplus L_2$.*

Proof. Let $M = (\mathfrak{L}, \mathfrak{R})$ be a $\cap_2$1DFA with m-state components. Let $M_1 := (\mathfrak{L}', \emptyset)$ and $M_2 := (\emptyset, \mathfrak{R}')$ be the $\cap_2$1DFAs derived from the two 'sides' of M after changing the initial state of each $A \in \mathfrak{L} \cup \mathfrak{R}$ to $A(\#)$. By the lemma's hypothesis, no member of $\mathfrak{F}(M_1)$ solves L_1 and no member of $\mathfrak{F}(M_2)$ solves L_2. So (Lemma 2), some y_1 confuses M_1 on L_1 and some y_2 confuses M_2 on L_2. We claim that $\#y_1\#y_2\#$ confuses M on $L_1 \vee L_2$ and thus M fails. (Similarly for $L_1 \oplus L_2$.)

Case 1: $y_1 \in L_1$ or $y_2 \in L_2$. Assume $y_1 \in L_1$ (if $y_2 \in L_2$, we work similarly). Then $\#y_1\#y_2\# \in L_1 \vee L_2$ and some $A' \in \mathfrak{L}'$ hangs on y_1. The corresponding $A \in \mathfrak{L}$ has $A(\#y_1\#y_2\#) = A'(y_1\#y_2\#) = \bot$. So, $\#y_1\#y_2\#$ confuses M on $L_1 \vee L_2$.

Case 2: $y_1 \notin L_1$ and $y_2 \notin L_2$. Then $\#y_1\#y_2\# \notin L_1 \vee L_2$, and each component of M_1 treats y_1 identically to a positive instance of L_1, and same for M_2, y_2, L_2:

$$(\forall A' \in \mathfrak{L}')(\exists \tilde{y}_1 \in L_1)\big(A'(y_1) = A'(\tilde{y}_1)\big), \tag{9}$$

$$(\forall A' \in \mathfrak{R}')(\exists \tilde{y}_2 \in L_2)\big(A'(y_2) = A'(\tilde{y}_2)\big). \tag{10}$$

It is then easy to verify that every $A \in \mathfrak{L}$ treats $\#y_1\#y_2\#$ as $\#\tilde{y}_1\#y_2\# \in L_1 \vee L_2$ ($\tilde{y}_1$ as guaranteed by (9)), and every $A \in \mathfrak{R}$ treats $\#y_1\#y_2\#$ as $\#y_1\#\tilde{y}_2\# \in L_1 \vee L_2$ ($\tilde{y}_2$ as guaranteed by (10)). Therefore, $\#y_1\#y_2\#$ confuses M on $L_1 \vee L_2$, again. □

Lemma 7. *Let L' be nontrivial, $\pi \in \{\cap, \cup, \mathrm{P}\}$, $\sigma \in \{_{\mathrm{L}}, _{\mathrm{R}}, _2\}$. If L has no π_σ1DFA with m-state components, then neither $L \wedge L'$ has. Similarly for $\neg L$ and $L \oplus L'$.*

Proof. We prove only the first claim, for $\pi = \cap$ and $\sigma = {}_{\mathrm{L}}$. Fix any $y' \in L'$. Given a $\cap_{\mathrm{L}}$1DFA M' solving $L \wedge L'$ with m-state components, we build a $\cap_{\mathrm{L}}$1DFA M solving L with m-state components: We just modify each component A' of M' so that the modified A' works on y exactly as A' on $\#y\#y'\#$. Then, M accepts y $\Leftrightarrow$ M' accepts $\#y\#y'\#$ $\Leftrightarrow$ $y \in L$. The modifications are straightforward. □

Lemma 8. *If L has no $\cap_{\text{L}}$1DFA with $\binom{m}{2}$-state components, then $\bigwedge L$ has no* P_{L}1DFA *with m-state components.*

Proof. Let $M = (\mathfrak{L}, \emptyset, F)$ be a P_{L}1DFA solving $\bigwedge L$ with m-state components. Let y be L-generic for M over $\bigwedge L$. We will build a $\cap_{\text{L}}$1DFA M' solving L.

By Lemma 3, an arbitrary x is in L iff $\text{LMAP}_A(y, xy)$ is total and injective for all $A \in \mathfrak{L}$; i.e., iff for all $A \in \mathfrak{L}$ and every two distinct $p, q \in \text{LVIEWS}_A(y)$,

$$\text{LCOMP}_{A,p}(xy) \text{ and } \text{LCOMP}_{A,q}(xy) \text{ exit } xy, \text{ into different states.} \qquad (11)$$

So, checking $x \in L$ reduces to checking (11) for each A and two-set of states of $\text{LVIEWS}_A(y)$. The components of M' will perform exactly these checks. To describe them, let us first define the following relation on the states of an $A \in \mathfrak{L}$:

$$r \asymp_A s \iff \text{LCOMP}_{A,r}(y) \text{ and } \text{LCOMP}_{A,s}(y) \text{ exit } y, \text{ into different states,}$$

and restate our checks as follows: for all $A \in \mathfrak{L}$ and all distinct $p, q \in \text{LVIEWS}_A(y)$,

$$\text{LCOMP}_{A,p}(x) \text{ and } \text{LCOMP}_{A,q}(x) \text{ exit } x, \text{ into states that relate under } \asymp_A. \quad (11')$$

Now, building 1DFAs to perform these checks is easy. For each $A \in \mathfrak{L}$ and $p, q \in \text{LVIEWS}_A(y)$, the corresponding 1DFA has 1 state for each two-set of states of A. The initial state is $\{p, q\}$. At each step, the automaton applies A's transition function on the current symbol and each state in the current two-set. If either application returns no value or both return the same value, it hangs; otherwise, it moves to the resulting two-set. A state $\{r, s\}$ is final iff $r \asymp_A s$. □

Lemma 9. *If L has no $\cap_2$*1DFA *with $\binom{m}{2}$-state components, then $\bigwedge L$ has no* $\text{P}_2$1DFA *with m-state components.*

4 Closure Properties and a Hierarchy

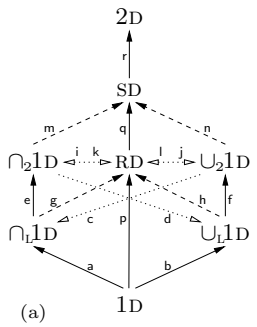
We are now ready to confirm the information of Fig. 1. We start with the positive cells of the table, continue with the diagram, and finish with the negative cells of the table. On the way, Lemma 11 proves a few useful facts.

Lemma 10. *Every '+' in the table of* Fig. 1b *is correct.*

Proof. Each closure can be proved easily, by standard constructions.(6) We also use the fact that every m-state RDFA (resp., SDFA) can be converted into an equivalent one with $O(m^2)$ states that keeps track of the number of rotations (resp., sweeps), and thus never loops. Similarly for 2DFAs and $O(m)$ [2]. □

Lemma 11. *The following separations and fact hold:*

[I] $\cap_{\text{L}}$1D $\not\supseteq$ re-1D, [II] P_{L}1D $\not\supseteq$ re-1D, [III] $\cap_2$1D $\not\supseteq \cup_{\text{L}}$1D $\cap$ RD, [IV] $\cap_{\text{L}}$1D $\cup \cap_{\text{R}}$1D $\not\supseteq \cap_2$1D $\cap$ SD, [V] there exists $\mathcal{L} \in \cup_{\text{L}}$1D $\cap$ RD such that $\bigwedge \mathcal{L} \notin \text{P}_2$1D.



(a)

(b)

	1D	∩L1D	∪L1D	RD	∩2 1D	∪2 1D	SD	2D	
¬	+	−	−	+	−	−	+	+	1
.R	−	−	−	−	+	+	+	+	2
∧	+	+	+	+	+	−	+	+	3
∨	+	+	+	+	−	+	+	+	4
⊕	+	−	−	+	−	−	+	+	5
⋀	+	+	−	−	+	−	−	+	6
⋁	+	−	+	−	−	+	−	+	7
⨁	+	−	−	?	−	−	?	+	8
	A	B	C	D	E	F	G	H	

Fig. 1. (a) A hierarchy from 1D to 2D: a solid arrow $\mathcal{C} \to \mathcal{C}'$ means $\mathcal{C} \subseteq \mathcal{C}'$ & $\mathcal{C} \not\supseteq \mathcal{C}'$; a dashed arrow means the same, but $\mathcal{C} \subseteq \mathcal{C}'$ only for the part of $\mathcal{C}$ that can be solved with polynomially many components; a dotted arrow means only $\mathcal{C} \not\supseteq \mathcal{C}'$. (b) Closure properties: '+' means closure; '−' means non-closure; '?' means we do not know.

Proof. [I] Let $\mathcal{L} := \bigvee\mathcal{J}$. We prove $\mathcal{L}$ is a witness. *First*, $\mathcal{J} \not\in$ 1D (Lemma 1) implies $\bigvee\mathcal{J} \not\in \cap_{\text{L}}$1D (Lemma 5). *Second*, $\mathcal{J}^{\text{R}} \in$ 1D (Lemma 1) implies $\bigvee\mathcal{J}^{\text{R}} \in$ 1D (by A7 of Fig. 1b), and thus $(\bigvee\mathcal{J})^{\text{R}} \in$ 1D (by (2)).

[II] Let $\mathcal{L} := \bigwedge\bigvee\mathcal{J}$. We prove $\mathcal{L}$ is a witness. *First*, $\bigvee\mathcal{J} \not\in \cap_{\text{L}}$1D (by I) implies $\bigwedge\bigvee\mathcal{J} \not\in$ P$_{\text{L}}$1D (Lemma 8). *Second*, $(\bigvee\mathcal{J})^{\text{R}} \in$ 1D (by I) implies $\bigwedge(\bigvee\mathcal{J})^{\text{R}} \in$ 1D (by A6 of Fig. 1b), and thus $(\bigwedge\bigvee\mathcal{J})^{\text{R}} \in$ 1D (by (2)).

[III] Let $\mathcal{L} := (\bigvee\mathcal{J}) \vee (\bigvee\mathcal{J}^{\text{R}})$. We prove $\mathcal{L}$ is a witness. *First*, $\bigvee\mathcal{J} \not\in \cap_{\text{L}}$1D (by I) implies $(\bigvee\mathcal{J})^{\text{R}} \not\in$ re-$\cap_{\text{L}}$1D or, equivalently, $\bigvee\mathcal{J}^{\text{R}} \not\in \cap_{\text{R}}$1D (by (2), (3)). Overall, both $\bigvee\mathcal{J} \not\in \cap_{\text{L}}$1D and $\bigvee\mathcal{J}^{\text{R}} \not\in \cap_{\text{R}}$1D, and thus $\mathcal{L} \not\in \cap_2$1D (Lemma 6). *Second*, $\mathcal{J} \in \cup_{\text{L}}$1D via $\cup_{\text{L}}$1DFAs with few components (Lemma 1) and thus $\bigvee\mathcal{J} \in \cup_{\text{L}}$1D also via $\cup_{\text{L}}$1DFAs with few components (by C7); therefore $\bigvee\mathcal{J} \in$ RD via the RDFA that simulates these components one by one. Hence, $\bigvee\mathcal{J} \in \cup_{\text{L}}$1D $\cap$ RD. In adddition, $\mathcal{J}^{\text{R}} \in$ 1D (Lemma 1) implies $\bigvee\mathcal{J}^{\text{R}} \in$ 1D (by A7), and thus $\bigvee\mathcal{J}^{\text{R}} \in \cup_{\text{L}}$1D $\cap$ RD as well (since 1D $\subseteq \cup_{\text{L}}$1D, RD). Overall, both $\bigvee\mathcal{J}$ and $\bigvee\mathcal{J}^{\text{R}}$ are in $\cup_{\text{L}}$1D $\cap$ RD. Hence, $\mathcal{L} \in \cup_{\text{L}}$1D $\cap$ RD as well (by C4,D4).

[IV] Let $\mathcal{L} := (\bigvee\mathcal{J}) \wedge (\bigvee\mathcal{J}^{\text{R}})$. We prove $\mathcal{L}$ is a witness. However, given that $\mathcal{L}^{\text{R}} = (\bigvee\mathcal{J}^{\text{R}})^{\text{R}} \wedge (\bigvee\mathcal{J})^{\text{R}} = \bigvee(\mathcal{J}^{\text{R}})^{\text{R}} \wedge \bigvee\mathcal{J}^{\text{R}} = \mathcal{L}$, we know $\mathcal{L} \in \cap_{\text{L}}$1D $\iff$ $\mathcal{L} \in \cap_{\text{R}}$1D, and thus it enough to prove only that $\mathcal{L} \in (\cap_2$1D $\cap$ SD$) \setminus \cap_{\text{L}}$1D. Here is how. *First*, $\bigvee\mathcal{J} \not\in \cap_{\text{L}}$1D (by I) and $\bigvee\mathcal{J}^{\text{R}}$ is nontrivial, so $\mathcal{L} \not\in \cap_{\text{L}}$1D (by Lemma 7). *Second*, $\bigvee\mathcal{J}^{\text{R}} \in$ 1D (by I) implies $(\bigvee\mathcal{J})^{\text{R}} \in \cap_2$1D $\cap$ SD (since 1D $\subseteq \cap_2$1D, SD) and thus $\bigvee\mathcal{J} \in \cap_2$1D $\cap$ SD as well (by E2,G2). Since both $\bigvee\mathcal{J}$ and $\bigvee\mathcal{J}^{\text{R}}$ are in $\cap_2$1D $\cap$ SD, the same is true of $\mathcal{L}$ (by E3,G3).

[V] Let $\mathcal{L} := (\bigvee\mathcal{J}) \vee (\bigvee\mathcal{J}^{\text{R}})$. By III, $\mathcal{L} \in (\cup_{\text{L}}$1D $\cap$ RD$) \setminus \cap_2$1D. By Lemma 9, $\mathcal{L} \not\in \cap_2$1D implies $\bigwedge\mathcal{L} \not\in$ P$_2$1D. □

Lemma 12. *Every arrow in the hierarchy of* Fig. 1a *is correct.*

Proof. All inclusions are immediate, either by the definitions or by easy constructions. Note that g, h, m, n refer only to the case of parallel automata with polynomially many components. The non-inclusions are established as follows.

[a,b] By Lemma 1. [d,k,m] By III. [c,l,n] By d,k,m, respectively, and (3),D1,G1. [g,p,h] By k,l. [e] By I, since re-1D $\subseteq \cap_2$1D. [f] By e and (3). [i,q,j] By II and since re-1D $\subseteq \cap_2$1D, $\cup_2$1D, SD and RD $\subseteq$ P$_L$1D. [r] Pick $\mathcal{L}$ as in V. Then $\bigwedge\mathcal{L} \notin$ SD (since SD $\subseteq$ P$_2$1D) but $\bigwedge\mathcal{L} \in$ 2D (by H6 and since $\mathcal{L} \in$ RD $\subseteq$ 2D). □

Lemma 13. *Every '−' in the table of* Fig. 1b *is correct.*

Proof. We examine the cells visiting them row by row, from top to bottom.

[B1] By C1 and (3). [C1] Pick $\mathcal{L}$ as in III. Then $\mathcal{L} \in \cup_L$1D but $\mathcal{L} \notin \cap_2$1D, so $\neg\mathcal{L} \notin \cup_2$1D and thus $\neg\mathcal{L} \notin \cup_L$1D. [E1] Pick $\mathcal{L}$ as in III. Then $\mathcal{L} \notin \cap_2$1D. But $\neg\mathcal{L} \in \cap_2$1D (because $\mathcal{L} \in \cup_L$1D, so $\neg\mathcal{L} \in \cap_L$1D). [F1] By E1 and (3).

[A2] By Lemma 1, $\mathcal{J}^R \in$ 1D but $\mathcal{J} \notin$ 1D. [B2] Pick $\mathcal{L}$ as in I. Then $\mathcal{L} \notin \cap_L$1D but $\mathcal{L}^R \in$ 1D $\subseteq \cap_L$1D. [C2] Pick $\mathcal{L}$ as in I. Since $\mathcal{L} \notin \cap_L$1D, we know $\neg\mathcal{L} \notin \cup_L$1D. Since $\mathcal{L}^R \in$ 1D, we know $\neg(\mathcal{L}^R) \in$ 1D (by A1) and thus $(\neg\mathcal{L})^R \in \cup_L$1D. [D2] Pick $\mathcal{L}$ as in II. Then $\mathcal{L}^R \in$ 1D $\subseteq$ RD but $\mathcal{L} \notin$ P$_L$1D $\supseteq$ RD.

[F3] Let $\mathcal{L}_1, \mathcal{L}_2$ be the witnesses for E4. Then $\mathcal{L}_1, \mathcal{L}_2 \in \cap_2$1D, hence $\neg\mathcal{L}_1, \neg\mathcal{L}_2 \in \cup_2$1D. But $\mathcal{L}_1 \vee \mathcal{L}_2 \notin \cap_2$1D, hence $\neg(\mathcal{L}_1 \vee \mathcal{L}_2) \notin \cup_2$1D or, equivalently $\neg\mathcal{L}_1 \wedge \neg\mathcal{L}_2 \notin \cup_2$1D. [E4] Pick $\mathcal{L}$ as in I. Then $\mathcal{L}^R \in$ 1D, hence $\mathcal{L}^R \in \cap_2$1D (since 1D $\subseteq \cap_2$1D), and thus $\mathcal{L} \in \cap_2$1D (by E2). But $\mathcal{L} \vee \mathcal{L}^R \notin \cap_2$1D (by III).

[B5,E5] Let $\mathcal{L}$ be the complement of the family of III. Then $\mathcal{L} \in \cap_L$1D $\subseteq \cap_2$1D. But $\neg\mathcal{L} \notin \cap_2$1D, and thus $\mathcal{L} \oplus \mathcal{L} \notin \cap_2$1D $\supseteq \cap_L$1D (Lemma 7). [C5,F5] Pick $\mathcal{L}$ as in III. Then $\mathcal{L} \in \cup_L$1D $\subseteq \cup_2$1D. But $\mathcal{L} \notin \cap_2$1D, hence $\neg\mathcal{L} \notin \cup_2$1D, and thus $\mathcal{L} \oplus \mathcal{L} \notin \cup_2$1D $\supseteq \cup_L$1D (Lemma 7).

[C6,D6,F6,G6] Pick $\mathcal{L}$ as in V. Then $\mathcal{L} \in \cup_L$1D$\cap$RD $\subseteq \cup_2$1D, SD. But $\bigwedge\mathcal{L} \notin$ P$_2$1D and thus $\bigwedge\mathcal{L} \notin \cup_L$1D, RD, $\cup_2$1D, SD. [B7,D7,E7,G7] Let $\mathcal{L}$ be the complement of the family of V. Then $\mathcal{L} \in \cap_L$1D $\cap$ RD (by D1), and thus also $\mathcal{L} \in \cap_2$1D, SD. But $\neg\bigvee\mathcal{L} = \bigwedge\neg\mathcal{L} \notin$ P$_2$1D, so $\neg\bigvee\mathcal{L} \notin \cap_L$1D, RD, $\cap_2$1D, SD, and same for $\bigvee\mathcal{L}$ (by D1,G1).

[B8,C8,E8,F8] By B5,C5,E5,F5. The witnesses there, are problems of the form $\mathcal{L} \oplus \mathcal{L}$, for some $\mathcal{L}$. Such problems simply restrict the corresponding $\bigoplus\mathcal{L}$. □

5 Conclusions

For each $n \geq 1$, let S_n be the problem: "Given a set $\alpha \subseteq [n]$ and two numbers $i, j \in [n]$ exactly one of which is in α, check that the one in α is j." Formally:

$$S_n := (\ \{\alpha ij \mid \alpha \subseteq [n]\ \ \&\ \ i \in \overline{\alpha}\ \ \&\ \ j \in \alpha\},\ \{\alpha ij \mid \alpha \subseteq [n]\ \ \&\ \ i \in \alpha\ \ \&\ \ j \in \overline{\alpha}\}\ ).$$

For $\mathcal{S} := (S_n)_{n\geq 1}$ the corresponding family, consider the family

$$\mathcal{R} = (R_n)_{n\geq 1} := \bigwedge\big((\bigoplus\mathcal{S}) \oplus (\bigoplus\mathcal{S}^R)\big).$$

It is easy to see that $\mathcal{S} \in 1\Delta = 1\text{N} \cap \text{co-1N}$ and that 1Δ is closed under $\cdot^R, \bigoplus, \oplus, \bigwedge$. Hence, $\mathcal{R} \in 1\Delta$ as well. At the same time, $\mathcal{S} \notin$ 1D (easily), so $\bigoplus\mathcal{S} \notin \cap_L$1D (Lemma 5) and $\bigoplus\mathcal{S}^R = (\bigoplus\mathcal{S})^R \notin \cap_R$1D, which implies $(\bigoplus\mathcal{S}) \oplus (\bigoplus\mathcal{S}^R) \notin \cap_2$1D (Lemma 6) and thus $\mathcal{R} \notin$ P$_2$1D (Lemma 9). Hence, $\mathcal{R} \notin$ SD either. Overall, $\mathcal{R}$ witnesses that $1\Delta \not\subseteq$ SD. This separation was first proven in [3]. There, it was witnessed by a language family $(\Pi_n)_{n\geq 1}$ that restricted liveness [5].(7)

We claim that, for all n, Π_n and R_n are '*essentially the same*': For each direction (left-to-right, right-to-left), there exists a $O(n)$-state one-way transducer that converts any well-formed instance u of Π_n into a string v in the promise of R_n such that $u \in \Pi_n \iff v \in R_n$. Conversely, it is also true that for each direction some $O(n)$-state one-way transducer converts any v from the promise of R_n into a well-formed instance u of Π_n such that $u \in \Pi_n \iff v \in R_n$.[8]

Therefore, using our operators, we essentially 'reconstructed' the witness of [3] in a way that identifies the source of its complexity (the witness of $1\Delta \not\subseteq 1\text{D}$ at its core) and reveals how its definition used reversal, parity, and conjuction to propagate its deterministic hardness upwards from 1D to SD without increasing its hardness with respect to 1Δ.

At the same time, using our operators, we can easily show that the witness of [3] is, in fact, unnecessarily complex. Already from the proof of Lemma 11[V] (and the easy closure of 1Δ under $\cdot^{\text{R}}, \bigvee, \vee, \bigwedge$), we know that even

$$\mathcal{L} = (L_n)_{n\geq 1} := \bigwedge\big((\bigvee \mathcal{J}) \vee (\bigvee \mathcal{J}^{\text{R}})\big)$$

witnesses $1\Delta \not\subseteq \text{SD}$. Indeed, $\mathcal{L}$ is both *simpler* than $\mathcal{R}$ (uses $\mathcal{J}, \bigvee, \vee$ instead of $\mathcal{S}, \bigoplus, \oplus$) and *more effective* (we can prove it needs $O(n)$ states on 1NFAs and co-1NFAs and $\Omega(2^{n/2})$ states on SDFAs, compared to $\mathcal{R}$'s $O(n^2)$ and $\Omega(2^{n/2}/\sqrt{n})$ [3]).

Finally, using our operators, we can systematically produce many different witnesses for each provable separation. The following corollary is indicative.

Corollary 1. *Let $\mathcal{L}$ by any family of problems.*

- *If $\mathcal{L} \in 1\Delta \setminus 1\text{D}$, then $\bigwedge\bigvee\mathcal{L} \in 1\Delta \setminus \text{RD}$.*
- *If $\mathcal{L} \in 1\Delta \setminus (1\text{D} \cup \text{re-1D})$, then $\bigwedge\bigvee\mathcal{L} \in 1\Delta \setminus \text{SD}$.*
- *If $\mathcal{L} \in \text{re-1D} \setminus 1\text{D}$, then $\bigwedge\bigvee(\mathcal{L} \vee \mathcal{L}^{\text{R}}) \in (1\Delta \cap 2\text{D}) \setminus \text{SD}$.*

Note how the alternation of $\bigwedge$ and $\bigvee$ (in 'conjunctive normal form' style) increases the hardness of a core problem; it would be interesting to further understand its role in this context. Answering the ?'s of Fig. 1b would also be very interesting—they seem to require tools other than the ones currently available.

References

1. P. Berman. A note on sweeping automata. In *Proc. of ICALP*, pages 91–97, 1980.
2. V. Geffert, C. Mereghetti, and G. Pighizzini. Complementing two-way finite automata. *Information and Computation*, 205(8):1173–1187, 2007.
3. C. Kapoutsis, R. Královič, and T. Mömke. An exponential gap between LasVegas and deterministic sweeping finite automata. In *Proc. of SAGA*, pages 130–141, 2007.
4. S. Micali. Two-way deterministic finite automata are exponentially more succinct than sweeping automata. *Information Processing Letters*, 12(2):103–105, 1981.
5. W. J. Sakoda and M. Sipser. Nondeterminism and the size of two way finite automata. In *Proc. of STOC*, pages 275–286, 1978.
6. J. I. Seiferas. Untitled manuscript. Communicated to Michael Sipser, Oct. 1973.
7. M. Sipser. Lower bounds on the size of sweeping automata. *Journal of Computer and System Sciences*, 21(2):195–202, 1980.

Appendix: technical comments (not to appear in final version)

(1) Working with *promise problems* instead of *languages* allows us to stop worrying about strings that do not encode legal inputs. Our automata become easier to design and describe, as they do not need to include the distracting "check that the input is in the correct form". Our arguments become more direct: e.g., we can directly write $\neg\bigwedge L = \bigvee\neg L$ without worrying which of the two languages, $\bigwedge L$ or its complement, contains the strings that are not #-separated instances of L (with languages, the equation $\neg\bigwedge L = \bigvee\neg L$ would be false).

At the same time, none of our upper and lower bounds is harmed: For every problem $\mathcal{L} = (L_n)_{n\geq 1}$ in this article, the corresponding family of promises $\mathcal{L}_{\text{P}} = ((L_n)_{\text{P}})_{n\geq 1}$ is at most as hard as $\mathcal{L}$ (for every class $\mathcal{C}$, $\mathcal{L} \in \mathcal{C} \implies \mathcal{L}_{\text{P}} \in \mathcal{C}$) and every class $\mathcal{C}$ in this article is closed under intersection. Therefore, a membership $\mathcal{L} \in \mathcal{C}$ is true or false irrespective of whether we consider $\mathcal{L}$ and the members of $\mathcal{C}$ to be families of promise problems or families of languages.

In short, promise problems allow us to work directly at the combinatorial core of a computational problem, by removing the distracting formalities imposed by languages. And, if used properly, they preserve the validity of our conclusions.□

(2) **Proof of Lemma 1:** It is easy to verify that J_n needs $\geq 2^n$ states on any 1DFA, but $\leq n$ states on a 1NFA, $\leq n$ components of ≤ 2 states each on a $\cap_{\text{L}}$1DFA, and $\leq n$ components of ≤ 1 state each on a $\cup_{\text{L}}$1DFA. Also, $(J_n)^{\text{R}}$ needs $\leq n$ states on a 1DFA and $\neg J_n$ needs $\leq n$ states on a 1NFA. □

(3) First, the values of $\text{LMAP}_A(y,z)$ are all in $\text{LVIEWS}_A(yz)$. Indeed: Let r be a value of $\text{LMAP}_A(y,z)$. Then some $q \in \text{LVIEWS}_A(y)$ is such that $\text{LMAP}_A(y,z)(q) = r$. Since $q \in \text{LVIEWS}_A(y)$, we know some $c := \text{LCOMP}_{A,p}(y)$ exits into q. Since $\text{LMAP}_A(y,z)(q) = r$, we know $d := \text{LCOMP}_{A,q}(z)$ exits into r. Overall, the computation $\text{LCOMP}_{A,p}(yz)$ must be exactly the concatenation of c and d. So, it exits into the same state as d, namely r. Therefore $r \in \text{LVIEWS}_A(yz)$.

Second, the values of $\text{LMAP}_A(y,z)$ cover the entire $\text{LVIEWS}_A(yz)$. Indeed: Suppose $r \in \text{LVIEWS}_A(yz)$. Then some $c' := \text{LCOMP}_{A,p}(yz)$ exits into r. Let q be the state of c' right after crossing the y-z boundary. Clearly, (i) the computation $\text{LCOMP}_{A,p}(y)$ exits into q, and (ii) the computation $\text{LCOMP}_{A,q}(z)$ exits into the same state as c', namely r. By (i), we know that $q \in \text{LVIEWS}_A(y)$. By (ii), we know that $\text{LMAP}_A(y,z)(q) = r$. Therefore, r is a value of $\text{LMAP}_A(y,z)$.

Therefore, $\text{LMAP}_A(y,z)$ partially surjects $\text{LVIEWS}_A(y)$ onto $\text{LVIEWS}_A(yz)$. □

(4) **Proof of Fact 2:** Suppose $r \in \text{LVIEWS}_A(yz)$. Then some computation $c := \text{LCOMP}_{A,p}(yz)$ exits into r. If q is the state of c after crossing the y-z boundary, then $\text{LCOMP}_{A,q}(z)$ is a suffix of c and exits into r. So, $r \in \text{LVIEWS}_A(z)$. □

(5) Suppose A accepts $z = y(xy)^k$ and let $c := \text{LCOMP}_{A,p}\big(y(xy)^k\big)$ be its computation, where p the initial state. Then c exits into a final state r. Easily, c can be split into subcomputations $c' := \text{LCOMP}_{A,p}(y)$, which exits into some state q,

and $c'' := \textsc{lcomp}_{A,q}\big((xy)^k\big)$, which exits into r. By the selection of q and r and the fact that π_A^k is an identity, we know

$$r = \textsc{lmap}_A\big(y, (xy)^k\big)(q) = \pi_A^k(q) = q.$$

Hence, c exits $z = y(xy)^k$ into the same state into which c' exits y. (Intuitively, in reading $(xy)^k$ to the right of y, the full computation c achieves nothing more than what is already achieved on y by its prefix c'.) Since r is final, A accepts y.

Conversely, any accepting computation of A on y can be extended into an accepting computation on z—this time by pumping up (as opposed to pumping down) and by using the computations that cause π_A^k to be an identity. □

(6)**Proof of Lemma 10:** None of the constructions is hard. We briefly sketch the ideas involved. We examine the cells column by column, from left to right.

Suppose m-state 1DFAs M_1, M_2 solve L_1, L_2, respectively. [A1] To solve $\neg L_1$, an $(m+1)$-state 1DFA simulates M_1 and accepts iff M_1 does not accept. [A3] To solve $L_1 \wedge L_2$, a $O(m)$-state 1DFA simulates M_1 between the first # and the second #, then M_2 between the second # and the third #, then accepts iff both simulations accepted. [A4-A5] Similarly to A3. [A6] To solve $\bigwedge L_1$, a $O(m)$-state 1DFA simulates M_1 between every two successive #, then accepts iff all simulations accepted. [A7-A8] Similarly to A6.

Suppose $\cap_{\text{L}}$1DFAs M_1, M_2 solve L_1, L_2, respectively, with k m-state components each. [B3] To solve $L_1 \wedge L_2$, a $\cap_{\text{L}}$1DFA uses k $O(m)$-state components, each constructed as in A3 out of a component of M_1 and the corresponding component of M_2. [B4] To solve $L_1 \vee L_2$, a $\cap_{\text{L}}$1DFA uses k^2 $O(m)$-state components, each constructed as in A4 out of a component of M_1 and a component of M_2. [B6] To solve $\bigwedge L_1$, a $\cap_{\text{L}}$1DFA uses k $O(m)$-state components, each constructed as in A6 out of a component of M_1. [C3,C4,C7] As in B4, B3, and B6, respectively.

Suppose m-state RDFAs M_1, M_2 solve L_1, L_2, respectively. Assume that they never loop (if one does, we can modify it so as to reject if its number of rotations exceeds m; the result is a $O(m^2)$-state RDFA). [D1] As in A1. [D3] To solve $L_1 \wedge L_2$, a $O(m)$-state RDFA simulates M_1 on the first part of the input ignoring the second part, then M_2 on the second part of the input ignoring the first part, then accepts iff both simulations accepted. [D4,D5] As in D3.

Suppose $\cap_2$1DFAs M_1, M_2 solve L_1, L_2, respectively, with m-state components. [E2] To solve L_1^{R}, an $\cap_2$1DFA simulates M_1 but with left and right components swapped. [E3,E6,F2,F4,F7] As in B3, B6, E2, C4, and C7, respectively.

Suppose m-state SDFAs M_1, M_2 solve L_1, L_2, respectively. Assume that they never loop (as for RDFAs, they can count the number of sweeps). [G1] As in A1. [G2] To solve L_1^{R}, an $(m+1)$-state SDFA moves its head to $\dashv$, then simulates M_1 but with left and right motions and endmarkers swapped. [G3-G5] As in D3-D5.

Suppose m-state 2DFAs M_1, M_2 solve L_1, L_2, respectively. Assume that they never loop (by [2]). [H1] As in A1. [H2] As in G2. [H3-H8] As in A3-A8. □

(7)**Definition of $(\Pi_n)_{n\geq 1}$.**For convenience, we include here the definition of the language family $(\Pi_n)_{n\geq 1}$, as it appeared in [3].

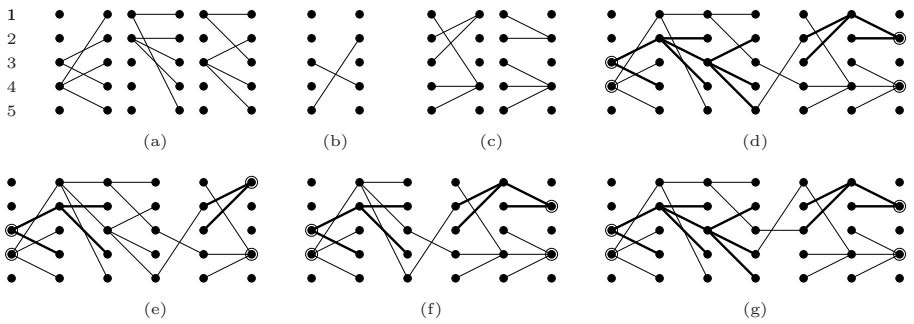


Fig. 2. (a) Three symbols of Γ_5; e.g., the leftmost one is $(3, 4, \{2, 4\})$. (b) The symbol $\{(3, 4), (5, 2)\}$ of X_5. (c) Two symbols of Δ_5. (d) The string defined by the six symbols of (a)-(c); in circles: the roots of the four trees; in bold: the two upper trees; the string is in Π'_5. (e) The upper left tree vanishes. (f) No tree vanishes, but the middle edges miss the upper left tree. (g) A well-formed string that does not respect the tree order.

Language Π_n consists of all #-delimited concatenations of the strings of another language, Π'_n. That is, $\Pi_n := \#(\Pi'_n\#)^*$. So, we need to present Π'_n.

Language Π'_n is defined over the alphabet $\Sigma'_n := \Gamma_n \cup X_n \cup \Delta_n$, where:

$$\Gamma_n := \{\ (i, j, \alpha)\ \mid\ i, j \in [n]\ \ \&\ \ i < j\ \ \&\ \ \emptyset \neq \alpha \subsetneq [n]\ \},$$
$$X_n := \{\ \{(i, r), (j, s)\}\ \mid\ i, j, r, s \in [n]\ \ \&\ \ i \neq j\ \ \&\ \ r \neq s\ \},$$
$$\Delta_n := \{\ (\alpha, j, i)\ \mid\ i, j \in [n]\ \ \&\ \ i < j\ \ \&\ \ \emptyset \neq \alpha \subsetneq [n]\ \}.$$

Intuitively, each $(i, j, \alpha) \in \Gamma_n$ represents a two-column graph (Fig. 2a) that has n nodes per column and contains exactly the edges that connect the ith left node to all right nodes inside α and the jth left node to all right nodes outside α. Symmetrically, each $(\alpha, j, i) \in \Delta_n$ represents a similar graph (Fig. 2c) containing exactly the edges that connect the ith and jth right nodes to the left nodes inside α and outside α, respectively. Finally, each $\{(i, r), (j, s)\} \in X_n$ represents a graph (Fig. 2b) containing only the edges connecting the ith and jth left nodes to the rth and sth right nodes, respectively. In all cases, we say that i and j (and r and s, in the last case) are the *roots* of the given symbol.

Of all strings over Σ'_n, consider those following the pattern $\Gamma_n^* X_n \Delta_n^*$. Each of them represents the multi-column graph (Fig. 2d) that we get from the corresponding sequence of two-column graphs when we identify adjacent columns. The symbol of X_n is called 'the middle symbol'—although it may very well not be in the middle position. If we momentarily hide the edges of that symbol, we easily see that the graph consists of exactly four disjoint trees, stemming out of the roots of the leftmost and rightmost columns. The tree out of the upper root of the leftmost column is naturally referred to as "the upper left tree". Similarly, the other trees are called "lower left", "upper right", and "lower right". Notice that, starting from the leftmost column, the two left trees may or may not both reach the left column of the middle symbol, as one of them may at some point 'cover all nodes' (Fig. 2e). Similarly, at least one of the two right trees reaches

the right column of the middle symbol, but not necessarily both. Also observe that, in the case where all four trees make it to the middle symbol, the two edges of that symbol may or may not collectively 'touch' all trees (Fig. 2f). A string over Σ'_n is called *well-formed* if it belongs to $\Gamma_n^* X_n \Delta_n^*$ and is such that each of the four trees contains exactly one of the roots of the middle symbol (Fig. 2dg).

Of all well-formed strings over Σ'_n, problem Π'_n consists of those that 'respect the tree order', in the sense that the two edges of the middle symbol do not connect an upper tree to a lower one (Fig. 2d). In other words, this is the set

$$\Pi'_n := \{z \in (\Sigma'_n)^* \mid z \text{ is well-formed and respects the tree order}\}.$$

Hence, to solve $\Pi_n = \#(\Pi'_n\#)^*$ means to check that the input string (over $\Sigma_n := \Sigma'_n \cup \{\#\}$) starts and ends with # and is such that *every* infix between two successive copies of # is well-formed and respects the tree order. □

(8) Intuitively, every instance of S_n inside v simulates and is simulated by a one-level extension of two left trees inside u, and is positive (resp., negative) iff the extension swaps (resp., preserves) the order of the trees; similarly for instances of S_n^{R} and right trees. Hence, for either conversion, a transducer simply 'translates' between instances of S_n or S_n^{R} and one-level extensions of left or right trees.

More carefully, the transducer from Π_n to R_n converts the symbols of a string $u \in \Sigma_n^*$ as follows:

- each symbol (i, j, α) is converted into the string $ij\#\alpha$;
- each symbol (α, j, i) is converted into the string $\alpha\#ji$;
- each symbol $\{(i, r), (j, s)\}$ is converted into the string $ij\#\#\#rs$;
- the symbol # is converted into a string of the form $x\#y$, where
  - x is $\{i\}\#\#$, if u contains to the left of # a symbol of the form (α, j, i); otherwise, x is the empty string;
  - y is $\#\#\{i\}$, if u contains to the right of # a symbol of the form (i, j, α); otherwise, y is the empty string.

It should be clear that a one-way transducer can indeed perform these conversions, irrespective of whether it is scanning u from left to right or from right to left. Moreover, non-constant memory is required only for the conversion of #, and then only i needs to be remembered. Overall, $O(n)$ states are enough. For the correctness of the conversion, one can prove that, if each #-delimited infix of u is well-formed, then the resulting v is in the promise of R_n; and then, $u \in \Pi_n$ iff $v \in R_n$. We omit a careful proof.

The transducer from R_n to Π_n converts the symbols of a $v \in (R_n)_{\mathrm{P}}$ (recall that the alphabet of R_n consists of all $\alpha \subseteq [n]$, all $i \in [n]$, and #) as follows:

- each symbol α within the left operand of a $\oplus$ converts into $(1, n, \alpha)$;
- each substring ij within the left operand of a $\oplus$ converts into $(i, j, [n-1])$;
- each symbol α within the right operand of a $\oplus$ converts into $(\alpha, n, 1)$;
- each substring ji within the right operand of a $\oplus$ converts into $([n-1], j, i)$;
- each substring ### between the operands of a $\oplus$ converts into $\{(1, n), (n, 1)\}$; if next to an endmarker, it converts into #;
- each substring ##### converts into #.

It should be clear that a one-way transducer can indeed perform these conversions, irrespective of whether it is scanning v from left to right or from right to left. Moreover, non-constant memory is required only for converting a substring ij (or ji), and then only i or j needs to be remembered. So, $O(n)$ states suffice. For the correctness of the conversion, one can prove that, if v is in the promise of R_n, then each #-delimited infix of the resulting u is well-formed; and then, $u \in \mathit{\Pi}_n$ iff $v \in R_n$. We omit a careful proof. □