ON THE POWER OF NONDETERMINISM IN SMALL TWO-WAY FINITE AUTOMATA

by

CHRISTOS KAPOUTSIS

B.S. Computer Science, University of Thessaloniki, Greece, 1997
M.S. Logic and Algorithms, University of Athens, Greece, 2000

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

at the Massachusetts Institute of Technology, June 2004

©2004 Massachusetts Institute of Technology.
All rights reserved.

Signature of author:

Department of Electrical Engineering and Computer Science
May 15, 2004

Certified by: ____________________________________________________________________________

Michael Sipser
Professor of Applied Mathematics
Thesis Supervisor

Accepted by: ____________________________________________________________________________

Arthur C. Smith
Professor of Electrical Engineering
Graduate Officer, EECS Graduate Office
ON THE POWER OF NONDETERMINISM
IN SMALL TWO-WAY FINITE AUTOMATA

by

CHRISTOS KAPOUTSIS

Submitted to the Department of Electrical Engineering and
Computer Science in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science.

Abstract. We examine the conjecture that one-way nondeterministic fi-
nite automata (1NFAs) can be exponentially more succinct than two-way
deterministic ones (2DFAs); equivalently, that no polynomial-size sequence
of 2DFAs can recognize $B$, for $B$ a particular sequence of regular languages
that is among the hardest of those recognizable by polynomial-size se-
quences of 1NFAs.

We prove that the most natural single-pass 2DFA algorithm for deciding
$B$ fails, “single-pass” meaning that the automaton is bound to terminate
as soon as it reaches an endmarker for the first time. On the way, we
introduce the notion of dilemmas as an interesting general tool for con-
structing hard inputs for 2DFAs.

Thesis supervisor: Michael Sipser, Professor of Applied Mathematics.
§1. Introduction. One of the principal goals of the theory of computation is the comparative study of deterministic and nondeterministic computing. By far, the main focus of this study has been on computations performed by the Turing machine. However, weaker machines have also been studied and several open problems about them have been challenging us, not only as questions we would like to answer in preparation for the harder corresponding Turing machine questions, but also as independent intriguing questions on the power of nondeterminism.

One-way deterministic finite automata (1DFAS) are the weakest of these machines, recognizing exactly the well-studied class of regular languages. Their nondeterministic counterparts (1NFAS) are well-known to be as powerful as 1DFAS themselves [7]; in fact, even one-way alternating finite automata (1AFAS) possess no extra power [2]. Interestingly enough, enhancing all these machines with a bidirectional read head has again no effect on their computational capability: 2DFAS, 2NFAS, and even 2AFAS can recognize only regular languages [7, 9, 3].

The robustness of the way in which these computability issues are resolved leads naturally to complexity questions, the resource of interest here being the size (number of states) of an automaton: given a regular language \( L \), what are the relations among the sizes of the smallest machines of the different types above that recognize \( L \)?

Viewing a machine that recognizes a language as a “description” of that language, we can restate this question as follows: given a regular language \( L \), what is the relative succinctness of the machines of the different types above in describing \( L \)? Some more precise questions include: How much more succinct can 1NFAS and 2DFAS be in comparison with 1DFAS? How much more succinct can 2NFAS be in comparison with 1NFAS and 2DFAS? What is the relative succinctness of 1NFAS and 2DFAS?\(^1\)

Answers to problems of this kind were first given in [1] and more systematically in [5]: both 1NFAS and 2DFAS were shown exponentially more succinct than 1DFAS. Later, the relation between 2NFAS and 1NFAS was resolved [8], via a proof that actually even 2DFAS can be exponentially more succinct than 1NFAS. As for the comparison of 2NFAS with 2DFAS, a similar resolution has been conjectured [8]: it appears that even 1NFAS can be exponentially more succinct than 2DFAS. It is this last problem that we examine. Figure 1(a) summarizes the setting.

Partial progress towards a proof of the conjecture was made already in [8] and more significantly in [10], with the introduction of sweeping automata (SDFAS) — the restricted version of 2DFAS that we get if we demand that the read head can change the direction of its motion only on

\(^1\)Regarding this last question, note that, since neither class of machines contains the other, economy in description may be achievable in both directions.
the two ends of the input—and the proof that these machines can be exponentially less succinct than 1NFAs. (A tighter version of this result was given in [4].) However, this progress is known to be strictly partial: SDFAs are exponentially less succinct than 2DFAs, as well [6]. (See Figure 1(b).)

1.1. A theoretical framework. In [8], a theoretical framework for dealing with these issues was introduced. We partially review it.

1.1. Definition. A language sequence is any infinite sequence of languages. The class 1D contains exactly the language sequences \( L = (L_n)_{n \geq 1} \) for which the size of the smallest 1DFA recognizing \( L_n \) grows polynomially in \( n \). The classes 1N, 2D, and 2N are defined analogously.

Intuitively and informally, 1D is the class of (sequences of) “languages” that have “small” 1DFAs; and similarly for 1N, 2D, and 2N. In this terminology, the setting of Figure 1(a) can also be depicted as in Figure 1(c), and the conjecture is that 1N \( \not\subseteq \) 2D.

All these classes are closed under a natural reducibility relation.

1.2. Definition. For alphabets \( \Delta_1, \Delta_2 \) and languages \( L_1 \subseteq \Delta_1^*, L_2 \subseteq \Delta_2^* \), we say \( L_1 \) homomorphically reduces to \( L_2 \), in symbols \( L_1 \leq_h L_2 \), if there is a mapping \( \mu : \Delta_1 \rightarrow \Delta_2^* \) and two symbols \( \alpha, \beta \in \Delta_2^* \) such that, for all \( s = s_1s_2 \cdots s_k \in \Delta_1^* \),

\[
s \in L_1 \iff \alpha \mu(s_1) \mu(s_2) \cdots \mu(s_k) \beta \in L_2.
\]

For language sequences \( L = (L_n)_{n \geq 1} \) and \( L' = (L'_n)_{n \geq 1} \), we say \( L \) homomorphically reduces to \( L' \), in symbols \( L \leq_h L' \), if there is a polynomially growing \( \nu : \mathbb{N} \rightarrow \mathbb{N} \) such that \( L_n \leq_h L'_\nu(n) \) at all \( n \geq 1 \).

Under \( \leq_h \), a complete language sequence \( B \) is known for 1N: \( B \in 1N \) and \( L \leq_h B \) for all \( L \in 1N \).

1.3. Definition. Fix any \( n \geq 1 \) and let \( \Sigma_n \) be the set of all graphs consisting of \( n \) left nodes, \( n \) right nodes and directed arcs from left to
right nodes. Figure 2(a) shows three examples of such graphs, for \( n = 5 \).
The concatenation of \( m \geq 0 \) graphs \( g_1, g_2, \ldots, g_m \in \Sigma_n \) is the graph \( g_1 g_2 \cdots g_m \) that we get if we identify every right node of \( g_i \) with the corresponding left node of \( g_{i+1} \), for all \( 1 \leq i < m \). Figure 2(b) shows the concatenation of the graphs of Figure 2(a). For simplicity, we always omit the arrow heads when drawing \( g_1 g_2 \cdots g_m \), as in Figure 2(c). An undirected path of \( g_1 g_2 \cdots g_m \) is any path in this undirected version of it. Moreover, in each column of nodes we talk of the 0-th, 1-st, \ldots, \( (n-1) \)-th node, starting from the topmost one.

For \( g = g_1 g_2 \cdots g_m \in \Sigma_n^* \) any string of symbols (graphs) from \( \Sigma_n \), we say \( g \) is live if there is a (directed) path from a leftmost to a rightmost node in \( g_1 g_2 \cdots g_m \); otherwise, we say \( g \) is dead. We then let

\[
B_n = \{ g \in \Sigma_n^* | \text{g is live} \} \quad \text{and} \quad B = (B_n)_{n \geq 1}.
\]

For example, the string of Figure 2(c) is in \( B_5 \). Note that we freely identify a string over \( \Sigma_n \) with (the undirected version of) the concatenation of its symbols.

It is easy to verify that \( B \nsubseteq 1D \); in fact, \( 2^n \) states are necessary and sufficient for a 1DFA to recognize \( B_n \).

1.2. Our approach. By the results of [8], to prove that \( 1N \nsubseteq 2D \) we can safely invest all our efforts in trying to prove \( B \nsubseteq 2D \). This is the approach of [8, 10] and we also adopt it here. However, our starting point is meaningfully different and we explain why.

To compute on a string with a bidirectional machine instead of a unidirectional one is to have the following two, quite distinct, additional capabilities:
(C1) the ability to make more than one passes (of both directions, possibly) over the string, and
(C2) the ability to change the direction of the read head motion at any point within a pass,
a pass being any computation between two successive readings of an endmarker. The proof that "small" SDFAS cannot recognize $B$ is essentially a proof that adding only (C1) as an extra to "small" 1DFAS does not raise their power to the exponentially higher levels of the demands of $B$. Similarly, the proof that 2DFAS are exponentially more succinct than SDFAS is essentially a proof that by adding (C2) as an extra to "small" SDFAS raises their power to exponentially higher levels—which are still lower than the demands of $B$, if the conjecture is true.

In this report we ask whether enhancing "small" 1DFAS with only the second of the capabilities above boosts their power enough so that the exponentially higher demands of $B$ are met. So, instead of fixing the direction of the read head motion within every pass and allowing more than one passes (as in [10]), we fix the number of passes to 1 and allow a bidirectional read head: can "small", single-pass, two-way deterministic finite automata recognize $B$? In the following, by machine we mean an automaton of this type.\footnote{As a side remark, note that the idea of a single-pass, two-way deterministic finite automaton is not new. This is exactly what a 2DFA was supposed to be when it was first introduced [7]. It seems that the reason why the single-pass restriction was abandoned is the fact that it doesn't affect the computability questions [9, Note 5]. As far as these questions are concerned, the intuitively clear advantage of being able to compute even after the boundaries of the input are encountered for the first time probably gets lost in the dust of the much more drastic collapse of the extra power that the bidirectionality of the read head was supposed to offer. However, for the complexity questions considered here, this advantage may well be of importance.}

Now, how could (such) a machine possibly tell whether a string is live? It appears that the most natural strategy for it would be to actually traverse the graph in order to check if a live path exists. Towards some insight into the general problem, we prove that this strategy fails. The proof is given in Section 3. Until then, we introduce the basic terminology (Section 2.1) and define dilemma (Section 2.2) and fuzzy strings (Section 2.3). Apart from explaining the definitions, the examples of Section 2 also build parts of the final proof.

\section{Preliminaries.}

\subsection{Strings, machines, and computations.}

Fix an alphabet $\Sigma$. For any $x \in \Sigma^+$, we write $|x|$ for the length of $x$ and $x(i)$ for the $i$-th symbol in $x$, $i = 1, 2, \ldots, |x|$. A property of the strings over $\Sigma$ is any $T \subseteq \Sigma^+$. We say $x$ has property $T$ iff $x \in T$.\footnote{As a side remark, note that the idea of a single-pass, two-way deterministic finite automaton is not new. This is exactly what a 2DFA was supposed to be when it was first introduced [7]. It seems that the reason why the single-pass restriction was abandoned is the fact that it doesn't affect the computability questions [9, Note 5]. As far as these questions are concerned, the intuitively clear advantage of being able to compute even after the boundaries of the input are encountered for the first time probably gets lost in the dust of the much more drastic collapse of the extra power that the bidirectionality of the read head was supposed to offer. However, for the complexity questions considered here, this advantage may well be of importance.}
A machine over $\Sigma$ is any triple $M = (Q, \Sigma, \delta)$ where $Q$ is a set of states and $\delta$ is a transition function, totally mapping $Q \times \Sigma$ to $Q \times \{-1, +1\}$. We also write $Q_M$ for $Q$. For any $k \geq 1$, we denote by $M_{\Sigma,k}$ the set of all $k$-state machines over $\Sigma$.

Given any $M = (Q, \Sigma, \delta)$, any $q \in Q$, $x \in \Sigma^+$, and $i \in \{1, \ldots, |x|\}$, the computation of $M$ on $x$ when $M$ is started at $q$ reading the $i$-th symbol of $x$ is the unique sequence

$$\text{COMP}_{M,q,i}(x) := ((q_t, i_t))_{0 \leq t \leq m}$$

that satisfies:

- $(q_0, i_0) = (q, i)$ and $0 \leq m \leq \infty$,
- $0 \leq t < m \Rightarrow 1 \leq i_t \leq |x|$, and $m \neq \infty \Rightarrow i_m \in \{0, |x| + 1\}$,
- $0 \leq t < m \Rightarrow \delta(q_t, x(i_t)) = (q_{t+1}, i_{t+1} - i_t)$.

We say $m$ is the length of this computation. If $m = \infty$, we say the computation loops. Otherwise, it hits left into $q_m$, if $i_m = 0$; or it hits right into $q_m$, if $i_m = |x| + 1$. When the computation loops or hits right, we say it misses left; similarly, it misses right if it loops or hits left.

As a machine can be started at any one of its states, we often talk of “machine $M_q$" as an abbreviation of “machine $M$ started at state $q$”. When $i = 1$ or $i = |x|$ we get the left computation of $M_q$ on $x$ or the right computation of $M_q$ on $x$,

$$\text{LCOMP}_{M,q}(x) := \text{COMP}_{M,q,1}(x) \quad \text{or} \quad \text{RCOMP}_{M,q}(x) := \text{COMP}_{M,q,|x|}(x),$$

respectively. Finally, $M_q$ recognizes property $T$ iff there exists $F \subseteq Q$ such that for all $x \in \Sigma^+$,

$$x \in T \iff \text{LCOMP}_{M,q}(x) \text{ hits right into a state in } F.$$

Note that other reasonable ways to define recognizability are possible. But it is easy to show that for each one of them a machine that recognizes $B$ exists only if a machine that recognizes $B$ under our definition exists. Hence, working with this particular definition is sufficient for our purposes.

2.2. Dilemmas. In this section we introduce a useful tool for constructing hard strings. We consider an alphabet $\Sigma$ fixed.

2.1. Definition. Let $k \geq 1$ and assume $\emptyset \neq T \subseteq \Sigma^+$. A $(k, T)$-dilemma is any string $u$ that has property $T$ and

$$(\forall M \in M_{\Sigma,k})(\forall q \in Q_M)(\forall u' \in T)[u' \text{ right-extends } u \implies [\text{LCOMP}_{M,q}(u') \text{ hits right } \iff \text{LCOMP}_{M,q}(u) \text{ hits right}]].$$

That is, such a $u"$ encodes" how the right boundary of all its right-extensions in $T$ is treated by the entire $M_{\Sigma,k}$. The name comes from the way we typically use such strings. In the following we explain this usage.
We fix a $u$ as above, some $M \in \mathcal{M}_{\Sigma, k}$, $q \in Q_M$, $x \in \Sigma^+$ and we consider $c = \text{LCOMP}_{M,q}(xu)$.

If $c$ misses right (Figure 3(a)), we know this must also be true for the left computation of $M_q$ on any right-extension of $xu$; in particular, this holds for any $xu'$ such that $u' \in T$ right-extends $u$.

If $c$ hits right (Figure 3(b)), we know $c$ crosses the $x$-$u$ boundary at least once and the last crossing occurs in a left-to-right step. If $p$ is the state $M$ enters in that step, then $c$ ends in the right-hitting computation $d = \text{LCOMP}_{M,p}(u)$. Now, by the selection of $u$, for any right-extension $u'$ of $u$ in $T$, the computation $d' = \text{LCOMP}_{M,q}(xu')$ also hits right. As a consequence, the computation $c' = \text{LCOMP}_{M,q}(xu')$ ends in $d'$, so that $c'$ hits right as well and departs from $x$ (i.e., crosses $x$'s right boundary for the last time) at exactly the same point where $c$ does.

In total, consider what $M_q$ is going through when she computes inside $xu$ and hasn't reached its right boundary yet: She has no clue what lies to the right of it (other than the fact that this is going to keep the entire input in $\{x\}T$) and still she has to decide whether she should keep her computation around the $x$-$u$ boundary (by looping inside $xu$ or hitting left) or cross the right boundary of $u$ and depart from $x$ forever. It's a dilemma. Note that this is true of $u$, no matter what $M \in \mathcal{M}_{\Sigma, k}$, what $q \in Q_M$, and what $x \in \Sigma^+$ we consider.

The next two claims guarantee an abundance of left-$(k,T)$-dilemmas, for every $k$ and $T$. The proof of the first one is trivial.

2.2. Proposition. Let $k \geq 1$ and $\emptyset \neq T \subseteq \Sigma^+$. If $u$ is a left-$(k,T)$-dilemma, so is any right extension of $u$ in $T$.

2.3. Lemma. Left-$(k,T)$-dilemmas exist, for every $k \geq 1$ and every $\emptyset \neq T \subseteq \Sigma^+$. 
PROOF. Fix any $k$ and $T$. Towards a contradiction, suppose left-$(k, T)$-dilemmas do not exist. Then, for all $u \in T$, we know

\[(\exists M \in \mathcal{M}_{\Sigma,k})(\exists q \in Q_M)(\exists u' \in T)[u' \text{ right-extends } u \land]

\[\text{[LCOMP}_{M,q}(u') \text{ hits right } \iff \text{LCOMP}_{M,q}(u) \text{ misses right]}\].

That is, no matter what $u \in T$ we start with (and there exists such $u$, since $T \neq \emptyset$), we can always right-extend it in a way, $u' \in T$, that forces at least one machine $M_q$, with $M \in \mathcal{M}_{\Sigma,k}$ and $q \in Q_M$, to change its right-boundary behaviour. Repeating this with $u'$ as the new start, (* guarantees we can further right-extend $u'$ in $T$ and force one more machine to change its right-boundary behaviour, and so on, ad infinitum.

However, since there are only finitely many machines in $\mathcal{M}_{\Sigma,k}$ and each one has only finitely many states, there are only finitely many behaviours that we can possibly change. Moreover, each of them can change at most once: if a machine hits right when run on $u$, it may miss right when run on some right-extension of $u$; but if it misses right on $u$, there is no way it can hit right when run on some right-extension of it. Hence, the non-terminating process of the first paragraph is not possible.

In a very similar manner, we can define right-$(k, T)$-dilemmas, for any $k$ and $T$, and show an abundance of such strings, as well.

2.4. Example. Consider the subset of $\Sigma_5$ that contains exactly the symbols that “permute the non-zero nodes”, i.e., the symbols of the form

\[\{(i \to \pi(i)) \mid 1 \leq i \leq 4\},\]

for $\pi$ any permutation of $\{1, 2, 3, 4\}$. For instance, the symbol describing the identity permutation is the leftmost symbol in Figure 2(a).

The set of all strings we can build using only these symbols is a property $\Pi$ over $\Sigma_5$. Figure 2(d) contains an example of a string in this property. Clearly, $\Pi$ can also be characterized as the property of containing exactly 4 disjoint live paths, neither of which passes through a 0-th node. Evidently, every string in $\Pi$ describes a permutation on $\{1, 2, 3, 4\}$ — in the example of Figure 2(d) this is the permutation $\{(1, 2), (2, 3), (3, 1), (4, 4)\}$.

For any $k \geq 1$, Lemma 2.3 guarantees we can find a left-$(k, \Pi)$-dilemma. Note that we can assume this describes the identity permutation, without loss of generality: If it describes some other permutation $\pi$, we can append to it the symbol that describes $\pi^{-1}$. The new string will describe the identity permutation and, by Proposition 2.2, will also be a left-$(k, \Pi)$-dilemma.

Similarly, a right-$(k, \Pi)$-dilemma that describes the identity permutation can be found. Let us fix the names $\iota_k$ and $\eta_k$ for these two dilemmas, respectively, and represent them graphically as in Figure 2 (the arrows show to what extensions the guarantee of each dilemma applies).
2.3. Fuzzy strings. A fuzzy string (on $n$) is any non-empty family $x$ of strings over $\Sigma_n$ that have the same length. The length $|x|$ of $x$ is this common length of its members. If we let $\subseteq$ be the relation

$$x \subseteq y \iff |x| = |y| \land x \text{ is a subgraph of } y,$$

between ordinary strings, then clearly any fuzzy string $x$ has a greatest lower bound $\cap x$ and a least upper bound $\cup x$ with respect to $\subseteq$. The edges of $x$ are exactly the edges of $\cup x$. We distinguish them to skeleton edges, that appear in $\cap x$, and fuzzy edges, that do not appear in $\cap x$. Hence, an edge of $x$ is fuzzy iff there are $x, y \in x$ such that $x$ contains the edge but $y$ doesn’t. A fuzzy string with members of length 1 is a fuzzy symbol.

2.5. Example. Consider the fuzzy symbol $a = \{a_0, a_1, \ldots, a_6\}$ containing the first 7 symbols of the second row in Figure 2. Clearly, $\cap a = a_0$ and $\cup a$ is as shown in the same figure (skeleton and fuzzy edges show as double and single lines, respectively). Note that $a$ contains all symbols that are at least $a_0$ and describe an injective partial mapping from $\{1, 2, 3, 4\}$ to $\{1, 2, 3, 4\}$. Doing the same with any of $b_0, c_0, d_0$, and $s_0$ of Figure 2, we can define 4 more 7-member fuzzy symbols and denote them $b, c, d, s$, respectively.

The concatenation operation is naturally extended to allow for fuzzy operands: $xy = \{xy \mid x \in x \land y \in y\}$, $xy = x[y]$, $xy = \{x\}y$; the result is always a fuzzy string. Moreover, $x$ is a prefix of $z$ iff $z = xy$ for some $y$. A property of the fuzzy strings (on $n$) is any family of fuzzy strings (on $n$). The Kleene-star operator can also be applied naturally on a fuzzy string; the result is a property.

2.6. Example. For any $k \geq 1$, we can use the dilemmas of Example 2.4, the fuzzy symbols of Example 2.5 and the symbol $\theta$ of Figure 2 to define the concatenations

$$a_k = a_n^\theta \theta \xi_k, b_k = b_n^\theta \theta \xi_k, c_k = c_n^\theta \theta \xi_k, d_k = d_n^\theta \theta \xi_k, s_k = \theta \xi_k s_n^\theta \theta \xi_k.$$

Figure 4 graphically depicts (the skeletons of) these strings. Using them, we can define the property

$$L_k = (a_k b_k c_k d_k)^*.$$

Figure 5(a) contains (the skeleton of) one of the simplest members of $L_k$, the fuzzy string $x = a_k b_k c_k d_k$.

Note that the two colored paths in $x$ are live, contain no fuzzy edges, and can possibly communicate with each other only via undirected paths that contain fuzzy edges. Figure 5(b) shows such a path. Moreover, the left end of any fuzzy edge can be driven to far right, in the sense that there is an $x \in x$ that contains an isolated directed path starting at this node and ending in the rightmost column. Figure 5(c) proves this for a particular node of this kind. Similarly, the right end of any fuzzy edge
can be driven to far left, as in Figure 5(d). The same facts are true of every member of \( L_k \), as the following lemma explains. We omit a detailed proof.

2.7. Lemma. For all \( k \geq 1 \), \( x \in L_k \), \( \cap x \) contains two disjoint connected components, joining the 1-st and 3-rd leftmost nodes with their rightmost counterparts. Moreover, for every node \( v \) in \( x \):

1. If \( v \) is the left end of a fuzzy edge, there exists \( x \in x \) that contains an isolated directed path from \( v \) to the 2-nd rightmost node.

2. If \( v \) is the right end of a fuzzy edge, there exists \( x \in x \) that contains an isolated directed path from the 2-nd leftmost node to \( v \).

We also consider the family \( L'_k \) of fuzzy strings that we get by prepending \( s_k \) to every string in \( L_k \). A simple member of \( L'_k \), the string \( s_k x \) is shown in Figure 7(b) (ignore the red path). Note that the two disjoint colored paths of \( x \) now meet at the 0-th leftmost node. Driving the ends of fuzzy edges to the string boundaries is still possible, in every member of \( L'_k \). \( \Box \)

The computation of a machine \( M \) on a fuzzy string \( x \) when started at state \( q \) reading its \( i \)-th symbol is simply the family of computations

\[
\text{COMP}_{M,q,i}(x) := \{ \text{COMP}_{M,q,i}(x) \mid x \in x \}.
\]

In the particular case where this family is a singleton, we say this is a stable computation. Bringing stability of computations into play, we can define fuzzy dilemmas.
Figure 5. (a) The skeleton of $a_k b_k c_k d_k$. (b) How to jump between the two colored paths. (c) Driving the left end of a fuzzy edge to far right. (d) Driving the right end of a fuzzy edge to far left.

2.8. Definition. Let $k \geq 1$ and $T$ be any non-empty property of the fuzzy strings on $n$. A left-$(k, T)$-dilemma is any $u \in T$ such that

$$\forall M \in \mathcal{M}_{\Sigma, k} \forall q \in Q_M \forall u' \in T \left[u' \text{ right-extends } u \implies \begin{array}{c} [LCOMP_{M,q}(u')] \text{ is stable} \& \text{hits right} \\ \iff LCOMP_{M,q}(u) \text{ is stable} \& \text{hits right}\end{array} \right].$$

That is, such a $u$ "encodes" how the fuzziness and right boundary of all its right-extensions in $T$ are treated by the entire $\mathcal{M}_{\Sigma, k}$. Direct analogues of Proposition 2.2 and Lemma 2.3 hold.

2.9. Example. For any $k \geq 1$, let $L_k$ be the property of Example 2.6. We fix a left-$(k, L_k)$-dilemma, denote it by $i_k$ and graphically represent it as in Figure 2. Note the reminder of the two disjoint connected components that join the 1-st and 3-rd leftmost nodes with their rightmost counterparts. 

§3. Machines that respect the graph. We now give a formal definition of the class of machines that try to distinguish live from dead strings by "traversing their graph". Intuitively, on input $x$ such a machine should
behave much like a finite-memory robot that starts at one of the leftmost
tnodes of $x$ and travels through the graph being able to reach only what
is accessible via the existing edges and see only what is visible via them.
For example, if at the 0-th left node of $\theta$ (Figure 2), such a robot could
only move to any of the 1-st or 4-th nodes (either left or right), but not to
a 2-nd or 3-rd node. Moreover, the robot’s next move would not depend
on whether edges $(2 \rightarrow 2)$ and $(3 \rightarrow 3)$ are present, since the robot could
not see beyond the connected component of its position in the graph.

So, whenever at state $q$ reading a symbol $a$, a machine of this type is
not actually reading the entire bipartite graph of $a$ but is rather focusing
on one of its nodes, basing its decision for what to do next only on the
connected component (in $a$) of that node, and moving only to a state at
which the focus is a node in the same component.

In order to present the formal definition, we need some names. A focus
is any pair $f = (i, d) \in \{0, 1, \ldots, n - 1\} \times \{-1, +1\}$. We write $f$ for
the focus $(i, -d)$. The $f$-th node of a string $x$ is its $i$-th leftmost node, if
d $= -1$; it is its $i$-th rightmost node, if $d = +1$. The $f$-th component of $x$
is the connected component of its $f$-th node. By $x \leadsto f$ we denote the unique
string that has the same length as $x$, has its $f$-th component identical to
the $f$-th component of $x$, and has no edges beyond that component. We
write $f \leadsto f'$ if $x$ contains an undirected path from its $f$-th node to its
$f'$-th node.

3.1. Definition. A foci assignment for a machine $M = (Q, \Sigma_n, \delta)$ is
any mapping $f : Q \rightarrow \{0, 1, \ldots, n - 1\} \times \{-1, +1\}$ such that, for all
$p, q \in Q, a \in \Sigma_n, d \in \{-1, +1\}$,

1. $\delta(p, a) = (q, d) \Rightarrow f(q)_2 \neq d \ & \ f(p) \leadsto a, \ f(q)$.
2. $\delta(p, a) = \delta(p, a : f(p))$

We say $f(p)$ is the focus of $p$. A machine respects the graph if there exists
a foci assignment for it.

So, $M$ respects the graph if there is a way to assign a focus to each
one of its states so that, whenever at state $p$, $M$ behaves as if it focused
on the $f(p)$-th node of the current input symbol, in the sense described
above. Condition (1) of the definition states that $M$ moves only to nodes
accessible from its current focus. Condition (2) states $M$ sees only what
is visible from its current focus.

Whenever a machine $M = (Q, \Sigma_n, \delta)$ that respects the graph is consid-
ered, a foci assignment $f$ for it will also be considered fixed. According
to this fixed $f$, the set $Q$ can be split into left and right (focusing) states,

$$Q_L = \{q \in Q \mid f(q)_2 = -1\}, \quad Q_R = \{q \in Q \mid f(q)_2 = +1\},$$
and we write \( \text{COMP}_{M, q}(x) \) to mean \( \text{LCOMP}_{M, q}(x) \) or \( \text{RCOMP}_{M, q}(x) \), if \( q \) is in \( Q_L \) or in \( Q_R \), respectively. Each \( q \in Q \) defines the equivalence relation,

\[
x \equiv_q y \iff x : f(q) = y : f(q)
\]

that equates strings with identical \( f(q) \)-th components. A computation of \( M \) on \( x \) visits an edge \( e \) iff at some point in it \( M \) reads the symbol containing \( e \) while at a state \( q \) such that the \( f(q) \)-th component of the symbol contains \( e \).

Evidently, if \( M \) respects the graph, its computations (from the same start state \( q \) and the same start position \( i \)) on two different strings \( x, y \) of the same length ought to be identical up to and including the first point where an edge of difference (i.e., a fuzzy edge of the fuzzy string \( \{x, y\} \)) is visited. If no such edge is ever visited, the machine computes the same on both strings, and hence treats them identically. In particular, the machine treats identically two strings in which all edges of difference are inaccessible from the starting focus (i.e., the \( f(q) \)-th node of \( x(i) \)).

We consider these observations obvious and we omit any proofs.

3.2. Lemma. Suppose \( x \) is a fuzzy string, \( M \) respects the graph, \( q \in Q_M \) and \( 1 \leq i \leq |x| \). If for all \( x \in x \) the computation \( \text{COMP}_{M, q, i}(x) \) visits no fuzzy edges, then the computation \( \text{COMP}_{M, q, i}(x) \) is stable.

3.3. Lemma. If \( M \) respects the graph, then for all \( q \in Q_M \), \( x, y \in \Sigma_+^n \),

\[ x \equiv_q y \implies \text{COMP}_{M, q}(x) = \text{COMP}_{M, q}(y) \]

As an immediate corollary to Lemma 3.3, we note that, for machines that respect the graph, the guarantee offered by a left-(\( k, T \))-dilemma holds not only for its right-extensions in \( T \), but also for their equivalents with respect to the starting state. A similar corollary holds for right-dilemmas.

3.4. Corollary. Suppose \( k \geq 1 \), \( \emptyset \neq T \subseteq \Sigma_+^n \), \( u \) is a left-(\( k, T \))-dilemma and \( M = (Q, \Sigma_n, \delta) \) respects the graph, with \( |Q| = k \). Then,

\[
(\forall q \in Q_L)(\forall u')(\exists v \in T)(u' \equiv_q v \land v \text{ right-extends } u \implies [\text{LCOMP}_{M, q}(u') \text{ hits right } \iff \text{LCOMP}_{M, q}(u) \text{ hits right}]).
\]

3.5. Example. Suppose \( M, q \in M_{\Sigma_n, k} \) respects the graph, \( f(q) = (1, -1) \) for some \( q \in Q_M \), and we know that \( \text{LCOMP}_{M, q}(\iota_k) \) hits right, where \( \iota_k \) is the left-(\( k, II \))-dilemma from Example 2.4. Can we conclude anything for the corresponding computation on the right-extension \( u' \) of \( \iota_k \) that appears in Figure 6? This extension is clearly not in \( II \). However, it is \( \equiv_q \)-equivalent to the right-extension \( v \) of \( \iota_k \), also in Figure 6, which is in \( II \). Since the left-computation of \( M_q \) on \( v \) must hit right, the same should be true for the computation on \( u' \).

We are now ready to prove the main lemma.
3.6. **Lemma.** Assume $k \geq 1$, $M \in \mathcal{M}_{\Sigma_0,k}$ respects the graph, and $r \in Q_M$ has focus $(0, -1)$. Then $M_r$ does not recognize $B_5$.

**Proof.** Towards a contradiction, suppose $M_r$ recognizes $B_5$. We first prove that $M_r$ is stable on all strings in the property $L'_k$ of Example 2.6.

3.6.1. **Claim.** For all $x \in L'_k$, $\text{COMP}_{M,r}(x)$ is stable.

**Proof.** Towards a contradiction, assume $\text{COMP}_{M,r}(x)$ is not stable. Then, there exists $x' \in x$ such that $\text{COMP}_{M,r}(x')$ visits a fuzzy edge. Let $(q, j)$ be the point where this happens for the first time and let $v$ be the $f(q)$-th node of $x(j)$. We distinguish cases on whether $q \in Q_L$ or $q \in Q_R$.

Figure 7 sketches the arguments for the special case $x = s_k a_k b_k c_k d_k$ and for some arbitrary $v$.

*Case 1: q is a left focusing state.* (See Figure 7(a).) Then $v$ is not in $s$, since $M_r$ cannot visit a fuzzy edge of $s$ from the left unless it has already visited other fuzzy edges. We also know some occurrence of $i_k$ precedes $x'(j)$ (this is true of every fuzzy symbol in $x$). Obviously, the left boundary of this occurrence must have been crossed at least once before the $(q, j)$ point, and the last crossing must have been left-to-right. Let $(p, i)$ be the point in the computation immediately after this last crossing, let $c$ be the computation up to that point, and $d$ the computation from that point up to $(q, j)$. Clearly, $d = \text{LCOMP}_{M,r}(i_k)$.

Since $v$ is the left end of a fuzzy edge, Lemma 2.7 guarantees some $x \in x$ contains an isolated path from $v$ to the 2-nd rightmost node; and, since $v$ is not in $s$, we can assume $s$ instantiates to $s_0$ in that particular $x$. Moreover, since the computations $c$ and $d$ on $x'$ visit no fuzzy edges, we know the left computation of $M_r$ on $x$ also starts with $cd$.

Now, let $x_0 = xe$ and $x_1 = xe'$, where $e$, $e'$ are the symbols from Figure 2. It is easy to verify that $x_0$ is dead (remember, $s$ instantiates to $s_0$ in $x_0$), $x_1$ is live, and $M_r$ still starts with $cd$ on both strings. If we prove $M_r$ continues to treat $x_0$ and $x_1$ the same even after the $(q, j)$
point, we will have the contradiction: $M_r$ treats a live and dead string identically, so it fails to recognize $B_5$.

So, let $y_0$ (resp. $y_1$) be the suffix of $x_0$ (resp. $x_1$) from the $i$-th symbol to the right. Clearly, $y_0 \equiv_p y_1$ (both $y_0 : f(p)$, $y_1 : f(p)$ consist of exactly the red path in Figure 7(a)). Moreover, if we start with $y_0 : f(p)$ and fill in 3 more disjoint live paths that left-start exactly as in $x_k$ and don't use 0-th nodes, we can actually construct a $y^*$ such that (i) $y^*$ right-extends $\iota_k$ in $\Pi$, and (ii) $y_0, y_1 \equiv_p y^*$. By (i) and the fact that $d$ hits right, we know $d' := \text{lcomp}_{M,p}(y^*)$ hits right, as well. By (ii), we know $\text{lcomp}_{M,p}(y_0) = \text{lcomp}_{M,p}(y_1) = d'$. Hence, from $(p, i)$ on, $M_q$ continues to behave the same on both $x_0, x_1$.

Case 2: $q$ is a right focusing state. (See Figure 7(b).) Since $x'(j)$ is a fuzzy symbol, there is an occurrence of $\eta_k$ to the right of it. The right boundary of this occurrence must have been crossed at least once before the $(q, j)$ point (otherwise, a fuzzy edge needs to have been visited earlier) and the last crossing must have been right-to-left. Let $(p, i)$ be the point in the computation just after this last crossing. As before, name $c$ the computation up to $(p, i)$ and $d$ the computation from there up to $(q, j)$. Clearly, $d = \text{rcomp}_{M,p}(\eta_k)$.

We know $v$ is the right end of a fuzzy edge. If $v$ is not in $s$, Lemma 2.7 says there is $x \in \mathcal{X}$ that contains an undirected path from $v$ to the 2-nd leftmost node of the first occurrence of $a_k$. Clearly, we can pick $x$ to have its $s$ instantiation so that this path actually reaches the 2-nd leftmost node of the entire string. In the case $v$ is in $s$, an $x$ containing an undirected path from $v$ to the 2-nd leftmost node of the entire string can be very easily found, again by proper instantiation of $s$.

Now, let $y$ be the prefix of $x$ up to the $i$-th symbol. By completing $y : f(p)$ (that contains exactly the red path in Figure 7(b)) with three more disjoint live paths that right-start exactly as in $\eta_k$ and don't use 0-th nodes, we can construct a $y^*$ such that (i) $y^*$ left-extends $\eta_k$ in $\Pi$, and (ii) $y \equiv_p y^*$. By (i) and the fact that $d$ hits left, we know $d' := \text{rcomp}_{M,p}(y^*)$ also hits left. By (ii) we know $\text{rcomp}_{M,p}(y) = d'$ and therefore hits left, as well. As a consequence, the entire $\text{lcomp}_{M,r}(x)$ hits left. But, as a live string, $x$ has both live and dead right-extensions and our conclusion implies $M$ treats all these extensions the same. So, again, $M$ fails to recognize $B_5$.

We will now reach a contradiction by constructing a string that $M$ fails to recognize correctly. To this end, let $u = s_k i_k$.

3.6.2. Claim. $\text{comp}_{M,r}(u)$ is stable and hits right into a state of focus either $(1, -1)$ or $(3, -1)$.

Proof. Let $c$ be this computation. Since $u \in L'_k$, by Claim 3.6.1 we know $c$ is stable. So, $c = \text{comp}_{M,r}(u)$ for the live member $u = \cap u$ of $u$. 

\end{proof}
Figure 7. Claim 3.6.1: (a) Case 1. Subscript $b$ is 0 or 1, and the dotted edges in the rightmost symbol are present iff $b = 1$. (b) Case 2.
(Figure 8(a)). If $c$ hits left or loops, then $M_r$ treats the live and dead extensions of $u$ the same, which contradicts the fact $M_r$ recognizes $B_5$. Hence, $c$ must be hitting right, into some state $p$ of focus $(i, -1)$ for $i \in \{1, 2, 3, 4\}$.

Suppose $i = 2$. Then $c$ actually ends in a left computation inside $t_k$ that hits right into $p$. Letting $e$, $e''$ be the symbols of Figure 2, $c_0 = \text{LCOMP}_{M_r}(ue)$, $c_1 = \text{LCOMP}_{M_r}(ue'')$, and reasoning as in Case 1 of the proof of Claim 3.6.1, we conclude that $c_0$ and $c_1$ must be identical, starting as $c$ and ending in a left computation inside $t_k e$ and $t_k e''$ respectively that hits right into a state of focus $(2, -1)$. But then $M_r$ treats the dead $ue$ exactly as the live $ue''$, which contradicts the assumption $M_r$ recognizes $B_5$.

In a similar manner, we can prove $i \neq 4$.

So, let $p$ be the state into which $\text{COMP}_{M_r}(u)$ hits right. We distinguish cases based on the focus of $p$.

Case 1: $p$ has focus $(1, -1)$. We then define the following sequence of extensions of $u$:

$$v_m = u(d_k d_k i_k)^m,$$

for all $m \geq 0$. Figure 8(b) shows the second member of this sequence.
3.6.3. Claim. For all \( m \geq 0 \), \( \text{COMP}_{M,r}(v_m) \) is stable and hits right into some state of focus \((1,-1)\).

Proof. Fix an \( m \geq 0 \) (and follow the proof for \( m = 1 \) on Figure 8(b)). Since \( c = \text{LCOMP}_{M,r}(u) \) is stable and hits right, the \( s_k-i_k \) boundary in \( u \) must have been crossed at least once, and the last time must have been left-to-right, into some state \( q \). Then \( c \) ends in \( d = \text{LCOMP}_{M,q}(i_k) \) and clearly \( d \) is stable and hits right into \( p \). Since \( d \) is stable, the foci of \( q \) and \( p \) must be in the same component in \( \cap i_k \). Knowing \( p \) has focus \((1,-1)\), we conclude the focus of \( q \) must be \((1,-1)\), as well.

Since \( d \) is stable and hits right, and \( i_k \) is a left-\((k,L_k)\)-dilemma, and \( i_k(d_k d_k i_k)^m \) right-extends \( i_k \) in \( L_k \), we know \( d' = \text{LCOMP}_{M,q}(i_k(d_k d_k i_k)^m) \) is also stable and hits right, into some state \( p' \). Again, the foci of \( q \) and \( p' \) must be in the same connected component in \( \cap (i_k(d_k d_k i_k)^m) \). Since \( q \) has focus \((1,-1)\), \( p' \) must have focus \((1,-1)\), too. \( \square \)

Looking at Figure 8(b), note that Claims 3.6.2 and 3.6.3 imply \( M_r \) doesn't visit the part of the blue path that lies to the right of the first occurrence of \( i_k \) in \( v_1 \). The same is true for all the \( v_m \)'s.

Now, since \( M \) has only finitely many states, there exist \( 0 \leq m_1 < m_2 \) such that \( \text{COMP}_{M,r}(v_{m_1}) \) and \( \text{COMP}_{M,r}(v_{m_2}) \) hit right into the same state, say \( p^* \), of focus \((1,-1)\). Figure 9 shows the rest of the argument for the case \( m_1 = 1 \), \( m_2 = 2 \). Let us also use \( v \) and \( v' \) instead of \( v_{m_1} \) and \( v_{m_2} \), respectively.

Let \( \xi \) be the symbol from Figure 2 and consider appending it to \( v \) and \( v' \). Then in both strings the green path dies, while the blue path stays live. Moreover, the computations

\[
c = \text{COMP}_{M,r}(v \xi) \quad \text{and} \quad c' = \text{COMP}_{M,r}(v' \xi)
\]

end in

\[
d = \text{RCOMP}_{M,p^*}(v \xi) \quad \text{and} \quad d' = \text{RCOMP}_{M,p^*}(v' \xi),
\]

respectively. Both \( d \) and \( d' \) must be stable (or else we can prove that \( M_r \) fails to recognize liveness, exactly as we did in the proof of Claim 3.6.1) and hit right (so that \( M_r \) can distinguish between the live and dead extensions of \( v \xi, v' \xi \) into states of focus \((3,-1)\) (because the rightmost symbol is \( \xi \) and this contains only the \((3 \rightarrow 3)\) edge).

Now, let \( j \) be such that \( v(|v| - j + 1) = s \); that is, \( j \) is the right-to-left index of \( s \) in \( v \). It is easy to check that the corresponding position in \( v' \) is occupied by a \( d \): \( v'(|v'| - j + 1) = d \). Consider the string \( v^* \) that we get from \( v' \) if we first instantiate \( s \) to \( s_0 \), and then replace the aforementioned \( d \) with an \( s \). The instantiation kills all paths other than the blue and the green one, while the replacement kills the blue path. So, only the green path is live in \( v^* \). And \( v \) is a suffix of \( v^* \).
Figure 10. Tricking the machine when she chooses to exit u from the blue path: For at least two different strings, $w\xi'$ and $w'\xi'$, she will reach the blue deadend in $\xi'$ having the same memory, $p^*$, of what the past has been. In the intermediate string, $w^*\xi'$, she will compute exactly as in $w'\xi'$ until the blue deadend in $\xi'$ is reached. She will then backtrack exactly as in $w\xi'$ and choose to follow the red path, fooled into thinking it is the green one she is following.
Remark that, going from \( v' \) to \( v^* \), we change no edge among those visited by \( \text{COMP}_{M,r}(v') \). As a result, \( \text{COMP}_{M,r}(v^*) \) is exactly \( \text{COMP}_{M,r}(v') \). In particular, \( \text{COMP}_{M,r}(v^*) \) hits right into \( p^* \). Consequently, 
\[
c^* = \text{COMP}_{M,r}(v^* \xi)
\]
ends in 
\[
d^* = \text{RCOMP}_{M,p^*}(v^* \xi).
\]
Since \( v^* \xi \) is a suffix of \( v^* \xi \), \( M_{p^*} \) on \( v^* \xi \) will behave exactly as on \( v^* \xi \), as long as it doesn't move beyond this suffix. But the behaviour of \( M_{p^*} \) on \( v^* \xi \) is \( d \), and we already know \( d \) hits right, so that \( M_{p^*} \) indeed moves only inside the \( v^* \xi \) suffix of \( v^* \xi \). Hence, \( d^* = d \). Which implies \( c^* \), \( c \) end identically, hence \( M_r \) treats \( v^* \xi \), \( v^* \xi \) the same.

Now, as we have already mentioned, the only live path in \( v^* \) is the green one, and we know appending \( \xi \) kills this path. In total, \( v^* \xi \) is a dead string. On the other hand, \( v^* \xi \) is live. Since \( M_r \) treats them the same, \( M_r \) fails to recognize \( B_5 \).

\textbf{Case 2:} \( p \) has focus \((3, -1)\). As above, we can define a sequence of extensions of \( u \),
\[
\text{for all } m \geq 0, \quad w_m = u(a_k c_k i_k)^m,
\]
(see Figure 8(c) for the second member of the sequence) and prove the following.

\textbf{3.6.4. Claim.} \textit{For all } \( m \geq 0 \), \( \text{COMP}_{M,r}(w_m) \text{ is stable and hits right into some state of focus } (3, -1) \).

\textbf{Proof.} Very similar to that of Claim 3.6.3.

Then, we proceed as we did for the \( v_m \)'s: We find two strings \( w, w' \) in the sequence that lead \( M_r \) to hit right into the same state. Starting with the longest of them, \( w' \), we instantiate \( s \) to \( s_0 \) and replace the appropriate occurrence of \( c \) by \( s \), to construct a new string \( w^* \). We then prove that, for \( \xi' \) the symbol from Figure 2, \( M_r \) treats \( w^* \xi' \), which is dead, exactly as \( w' \xi' \), which is live. Figure 10 explains the setting for the particular case \( w = w_1, w' = w_2 \).

The main theorem is now easy to see.

\textbf{3.7. Theorem.} \textit{No machine that respects the graph can recognize } \( B_5 \). Hence, no sequence of machines that respect the graph can recognize } \( B \).

\textbf{Proof.} Suppose \( M \) respects the graph and recognizes \( B_5 \). Let \( k \) be the number of states in \( M \), let \( r \) be the state from which \( M \) must be started so that it recognizes \( B_5 \), and let \( f \) be the focus of \( r \).

If \( f = (0, -1) \), then Lemma 3.6 has already proven the claim. If \( f \) is any other focus such that the \( f \)-th and \( (0, -1) \)-th nodes of \( \theta \) are in the same component in \( \theta \), then the claim holds again, obviously.
For any other \( f = (i, d) \), we can fix a permutation \( \pi \) of \( \{0, 1, \ldots, n - 1\} \) such that \( \pi(0) = i \), if \( d = -1 \), or \( \pi(1) = i \), if \( d = +1 \). Using \( \pi \), we can replace every edge \((i \rightarrow j)\) with the edge \((\pi(i) \rightarrow \pi(j))\), in every symbol in every string we defined so far. Redoing the proofs for the new strings, we will again conclude \( M_f \) fails to recognize \( B_5 \).

§4. Conclusion and further work. By the remark at the end of Section 2.1, Theorem 3.7 actually states that there is no general way to distinguish whether a string is dead or live via a single-pass, graph-respecting, two-way, read-only deterministic computation on it.

What happens if we remove “single-pass” from this last statement? We do not know. The above proof fails, very easily. Conceivably, the ability to compute after the detection of the ends of the input could allow a graph-respecting 2DFA (maybe even a small one) to correctly decide liveness. However, we conjecture this is not so. The restriction to respect the graph seems to be so severe, that even huge 2DFAs that obey it will fail to solve \( B_n \), probably even for very small \( n \).

What happens if we remove “graph-respecting” from this statement? Clearly, the statement becomes false, as even a 1DFA can decide the liveness of its input. Reintroducing the restriction that the automaton be small, we return to the core of our investigations: is there a small, single-pass 2DFA that solves liveness?

Trying to extend Theorem 3.7 by removing the “single-pass” restriction (but keeping our attention on graph-respecting machines) is probably the next most natural and less difficult step we could pursue. On the other hand, removing the “graph-respecting” restriction (while focusing on small 2DFAs) seems to be much harder, as it is clearly very close to the heart of the general problem of the comparison between small 2DFAs and small 1NFAs.

REFERENCES


