ALGORITHMS AND LOWER BOUNDS IN FINITE AUTOMATA SIZE COMPLEXITY

by

CHRISTOS KAPOUTSIS

B.S. Computer Science, Aristotle University of Thessaloniki, Greece, 1997
M.S. Logic and Algorithms, Capodistrian University of Athens, Greece, 2000
M.S. Computer Science, Massachusetts Institute of Technology, 2004

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

DOCTOR OF PHILOSOPHY

at the Massachusetts Institute of Technology, June 2006

Signature of author: [illegible]

Department of Electrical Engineering and Computer Science
May 5, 2006

Certified by: [illegible]

Michael Sipser
Professor of Applied Mathematics
Thesis Supervisor

Accepted by: [illegible]

Arthur C. Smith
Professor of Electrical Engineering
Graduate Officer, EECS Graduate Office

ALGORITHMS AND LOWER BOUNDS IN FINITE AUTOMATA SIZE COMPLEXITY

by

CHRISTOS KAPOUTSIS

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

Abstract

In this thesis we investigate the relative succinctness of several types of finite automata, focusing mainly on the following four basic models: one-way deterministic (1DFAs), one-way nondeterministic (1NFAs), two-way deterministic (2DFAs), and two-way nondeterministic (2NFAs).

First, we establish the exact values of the trade-offs for all conversions from two-way to one-way automata. Specifically, we prove that the functions

$$n(n^n - (n-1)^n), \qquad \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\binom{n}{i}\binom{n}{j}(2^i-1)^j, \qquad \binom{2n}{n+1}$$

return the exact values of the trade-offs from 2DFAs to 1DFAs, from 2NFAs to 1DFAs, and from 2DFAs or 2NFAs to 1NFAs, respectively.

Second, we examine the question whether the trade-offs from 1NFAs or 2NFAs to 2DFAs are polynomial or not. We prove two theorems for *liveness*, the complete problem for the conversion from 1NFAs to 2DFAs. We first focus on *moles*, a restricted class of 2NFAs that includes the polynomially large 1NFAs which solve liveness. We prove that, in contrast, 2DFA moles cannot solve liveness, irrespective of size. We then focus on *sweeping* 2NFAs, which can change the direction of their input head only on the end-markers. We prove that all sweeping 2NFAs solving the complement of liveness are of exponential size. A simple modification of this argument also proves that the trade-off from 2DFAs to sweeping 2NFAs is exponential.

Finally, we examine conversions between two-way automata with more than one head-like devices (e.g., heads, linearly bounded counters, pebbles). We prove that, if the automata of some type A have enough resources to (i) solve problems that no automaton of some other type B can solve, and (ii) simulate any unary 2DFA that has additional access to a linearly-bounded counter, then the trade-off from automata of type A to automata of type B admits no recursive upper bound.

Thesis supervisor: Michael Sipser, Professor of Applied Mathematics.

Contents

Introduction

The main subject of this thesis is the 2D vs. 2N problem, a question on the power of nondeterminism in two-way finite automata. We start by defining it, explaining the motivation for its study, and describing our progress against it.

1. The 2D vs. 2N Problem

A *two-way deterministic finite automaton* (2DFA) is the machine that we get from the common *one-way deterministic finite automaton* (1DFA) when we allow its input head to move in both directions; equivalently, this is the machine that we get from the common *single-tape deterministic Turing machine* (DTM) when we do not allow its input head to write on the input tape. The nondeterministic version of a 2DFA is simply called *two-way nondeterministic finite automaton* (2NFA) and, as usual, is the machine that we get by allowing more than one options at each step and acceptance by any computation branch. The 2D vs. 2N question asks whether 2NFAs can be strictly more efficient than 2DFAs, in the sense that there is a problem for which the best 2DFA algorithm is significantly worse than the best 2NFA one.

Of course, to complete the description of the question, we need to explain how we measure the efficiency of an algorithm on a two-way finite automaton. It is easy to check that, with respect to the length of its input, every algorithm of this kind uses zero space and at most linear time. Therefore, the time and space measures—our typical criteria for algorithmic efficiency on the full-fledged Turing machine—are of little help in this context. Instead, we focus on the size of the program that encodes this algorithm, namely the size of the transition function of the corresponding two-way finite automaton. In turn, a good measure for this size is simply the automaton's number of states.

So, the 2D vs. 2N question asks whether there is a problem that, although it can be solved both by 2DFAs and by 2NFAs, the smallest possible 2DFA for it (i.e., the 2DFA that solves it with the fewest possible states) is still significantly larger than the smallest possible 2NFA for it. To fully understand the question, two additional clarifications are needed.

First, it is well-known that the problems that are solvable by 2DFAs are exactly those described by the regular languages, and that the same is true for 2NFAs [**53**]. Hence, efficiency considerations aside, the two types of automata have the same power—in the same way that 1DFAs or DTMs have the same power with their nondeterministic counterparts. Therefore, the above reference to problems that "can be solved both by 2DFAs and by 2NFAs" is a reference to exactly the regular problems.

Second, although we explained that efficiency is measured with respect to the number of states, we still have not defined what it means for the number of states in one automaton to be "significantly larger" than the number of states in another one. The best way to clarify this aspect of the question is to give a specific example.

Consider the problem in which we are given a list of subsets of $\{0, 1, \ldots, n-1\}$ and we are asked whether this list can be broken into sublists so that the first set of every sublist contains the number of sets after it [**51**]. More precisely, the problem is defined on the alphabet $\Delta_n := \mathcal{P}(\{0, 1, \ldots, n-1\})$ of all sets of numbers smaller than n. A string $a_0 a_1 \cdots a_l$ over this alphabet is a *block* if its first symbol is a set that contains the number of symbols after it, that is, if $a_0 \ni l$. The problem consists in determining whether a given string over Δ_n can be written as a concatenation of blocks. For example, for $n = 8$ and for the string

$$\{1,2,4\}\emptyset\{4\}\{0,4\}\{2,4,6\}\{4\}\{4,6\}\emptyset\{3,6\}\emptyset\{2,4\}\{5,7\}\{0,3\}\{4,7\}\emptyset\{4\}\emptyset\{4\}\{0,1\}\{2,5,6\}\{1\}$$

the answer should be "yes", since this is the concatenation of the substrings

$$\{1,\mathbf{2},4\}\emptyset\{4\}\quad\{0,\mathbf{4}\}\{2,4,6\}\{4\}\{4,6\}\emptyset\quad\{\mathbf{3},6\}\emptyset\{2,4\}\{5,7\}\quad\{\mathbf{0},3\}\quad\{4,\mathbf{7}\}\emptyset\{4\}\emptyset\{4\}\{0,1\}\{2,5,6\}\{1\}$$

where the first set in each substring indeed contains the number of sets after it in the same substring, as indicated by boldface. In contrast, for the string

$$\{1,2,7\}\{4\}\{5,6\}\emptyset\{3,6\}\{2,4,6\}$$

the answer should be "no", as there is obviously no way to break it into blocks.

Is there a 2DFA algorithm for this problem? Is there a 2NFA algorithm? Indeed, the problem is regular. The best 2NFA algorithm for it is a rather obvious one:

> We scan the list of sets from left to right. At the start of each block, we read the first set. If it is empty, we just hang (in this branch of the computation). Otherwise, we nondeterministically select from this set the correct number of remaining sets in the block. We then consume as many sets, counting from that number down. When the count reaches 0, we know the block is over and a new one is about to start. In the end, we accept if the list and our last count-down finish simultaneously.

It is easy to see that this algorithm can be implemented on a 2NFA (which does not actually use its bidirectionality) with exactly 1 state per possible value of the counter, for a total number of n states. As for a 2DFA algorithm, here is the best *known* one:

> We scan the list of sets from left to right. At each step, we remember all possibilities about how many more sets there are in the most recent block. (E.g., right after the first set, every number in it is a possibility.) After reading each set, we decrease each possibility by 1, to account for the set just consumed; if a possibility becomes 0, we replace it by all numbers of the next set. If at any point the set of possibilities gets empty, we just hang. Otherwise, we eventually reach the end of the list. There, we check whether 0 is among the possibilities. If so, we accept.

Easily, this algorithm can be implemented on a 2DFA (which does not actually use its bidirectionality) that has exactly 1 state per possible non-empty[1] set of possibilities, for a total number of $2^n - 1$ states. Overall, we see that the difference in size between the automata implementing the two algorithms is exponential. Such a difference, we surely want to call "significant".

[1]Note that a 2DFA is allowed to reject by just hanging anywhere along its input. Without this freedom, the number of states required to implement our algorithm would actually be 2^n.

At this point, after the above clarifications, we may want to restate the 2D vs. 2N question as the question whether there exists a regular problem for which the smallest possible 2DFA is still super-polynomially larger than the smallest possible 2NFA. However, a further clarification is due.

Given any fixed regular problem, the sizes of the smallest possible 2DFA and the smallest possible 2NFA for it are nothing more than just two numbers. So, asking whether their difference is polynomial or not makes little sense. What the previous example really describes is a *family of regular problems* $\Pi = (\Pi_n)_{n\geq 0}$, one for each natural value of n; then, a *family of* 2NFAs $N = (N_n)_{n\geq 0}$ which solve these problems and whose sizes grow linearly in n; and, finally, a *family of* 2DFAs $D = (D_n)_{n\geq 0}$ which also solve these problems and whose sizes grow exponentially in n. Therefore, our reference to "the difference in size between the automata" was really a reference to the difference in the rate of growth of the sizes of the automata in the two families. It is this difference that can be characterized as polynomial or not. Naturally, we decide to call it significant if one of the two rates can be bounded by a polynomial but the other one cannot.

So, the 2D vs. 2N question asks whether there exists a family of regular problems such that the family of the smallest 2NFAs that solve them have sizes that grow polynomially in n, whereas the family of the smallest 2DFAs that solve them have sizes that grow super-polynomially in n. Equivalently, the question is *whether there exists a family of regular problems that can be solved by a polynomial-size family of* 2NFAs *but no polynomial-size family of* 2DFAs. With this clarification, we are ready to explain the name "2D vs. 2N". We define 2D as the class of families of regular problems that can be solved by polynomial-size families of 2DFAs, and 2N as the corresponding class for 2NFAs [**48**]. Under these definitions, we obviously have 2D $\subseteq$ 2N, and the question is *whether* 2D *and* 2N *are actually different.* Observe how the nature of the question resembles both circuit and Turing machine complexity: like circuits, we are concerned with the rate of growth of size in families of programs; unlike circuits and like Turing machines, each program in a family can work on inputs of any length.

Concluding this description of the problem, let us also remark that its formulation in terms of families is not really part of our every-day vocabulary in working on the problem. Instead, we think and speak as if n were a built-in parameter of our world, so that only the n-th members of the three families (of problems, of 2NFAs, and of 2DFAs) were visible. Under this pretense, the 2D vs. 2N question asks whether there is a regular problem that can be solved by a polynomially large 2NFA but no polynomially large 2DFA—and it is redundant to mention what parameter "polynomially large" refers to. In addition, we also use "small" as a more intuitive substitute for "polynomially large", and drop the obviously redundant characterization "regular". So, the every-day formulation of the question is *whether there exists a problem that can be solved by a small* 2NFA *but no small* 2DFA. Throughout this thesis, we will occasionally be using this kind of talk, with the understanding that it is a substitute for its formal interpretation in terms of families.

2. Motivation

Our motivation for studying the 2D vs. 2N question comes from two distinct sources: the theory of *computational complexity* and the theory of *descriptional complexity*. We discuss these two different contexts separately.

2.1. Computational complexity. From the perspective of computational complexity, the 2D vs. 2N question falls within the general major goal of understanding the power of nondeterminism. For certain computational models and resources, this quest has been going on for more than four decades now. Of course, the most important (and correspondingly famous) problem of this kind is P vs. NP, defined on the Turing machine and for time which is bounded by some polynomial of the length of the input. The next most important question is probably L vs. NL, also defined on the Turing machine and for space which is bounded by the logarithm of some polynomial of the length of the input.

It is perhaps fair to say that *our progress against the core of these problems has been slow.* Although our theory is constantly being enriched with new concepts and new connections between the already existing ones, major advances of our understanding are rather sparse. At the same time, the common conceptual origin as well as certain combinatorial similarities between these problems has led some to suspect that essentially *the same elusive idea lies at the core of all problems of this kind.* In particular, the suspicion goes, this idea may be independent of the specifics of the underlying computational model and resource.

In this context, a possibly advantageous approach is to focus on weak models of computation. The simple setting provided by these models allows us to work closer to the set-theoretic objects that are produced by their computations. This serves us in two ways. First, it obviously helps our arguments become cleaner and more robust. Second, it helps our reasoning become more objective, by neutralizing our often misleading algorithmic intuitions about what a machine may or may not do.

The only problem with this approach is that it may very well lead us to models of computation that are too weak to be relevant. In other words, some of these models are obviously not rich enough to involve ideas of the kind that we are looking for. We should therefore be careful to look for evidence that indeed such ideas are present. We believe that the 2D vs. 2N question passes this test, and we explain why.

2.1-I. *Robustness.* One of the main reasons why problems like P vs. NP and L vs. NL are so attractive is their robustness. The essence of each question remains the same under many different variations of the mathematical definitions of the model and/or the resource. It is only on such stable grounds that the theoretical framework around each question could have been erected, with the definition of the *classes of problems* that can be solved in each case (P, NP, L, NL) and the identification of *complete problems* for the nondeterministic classes, that allowed for a more tangible reformulation of the original question (is satisfiability in P? is connectivity in L?). The coherence and richness of these theories further enhance our confidence that they indeed describe important high-level questions about the nature of computation, as opposed to technical low-level inquiries about the peculiarities of particular models.

The 2D vs. 2N problem is also robust. Its resolution is independent of all reasonable variations of the definition of the two-way finite automaton and/or the size measure, including changes in the conventions for accepting and rejecting, for end-marking the input, for moving the head; changes in the size of the alphabet, which may be fixed to binary; changes in the size measure, which may include the number of transitions or equal the length of some fixed binary encoding. It is on this stable ground that the classes 2D and 2N have been defined. In addition, 2N-complete (families of) problems have been identified [**48**], allowing more concrete forms of the question. Overall, there is no doubt that in this case, too, our investigations are beyond technical peculiarities and into the high-level properties of computation.

2.1-II. *Hardness.* A second important characteristic of problems like P vs. NP and L vs. NL that contributes to their popularity is their hardness. Until today, a long list of people that cared about these problems have tried to attack them from several different perspectives and with several different techniques. Their limited success constitutes significant evidence that, indeed, answering these questions will require a *deeper understanding* of the combinatorial structure of computation. In this sense, the answer is likely to be highly rewarding.

In contrast, the 2D vs. 2N problem can boast no similar attention on the part of the community, as it has never attracted the efforts of a large group of researchers. However, it does belong to the culture of this same community and several of its members have tried to attack it or problems around it, with the same limited success. In this sense, it is again fair to predict that the answer to this question will indeed involve ideas which are deep enough to be significantly rewarding.

2.1-III. *Surprising conjecture.* If we open a computational complexity textbook, we will probably find P defined as the class of problems that can be solved by polynomial-time Turing machines, and NP as the class of problems that can be solved by the same machines when they are enhanced with the ability to make nondeterministic choices. Then, we will probably also find a discussion of the standard conjecture that P $\neq$ NP, justified by the well-known compelling list of pragmatic and philosophical reasons.

In this context, the conjecture does not sound surprising at all. There is certainly nothing strange with the enhanced machines being strictly more powerful than the original, non-enhanced ones. They start already as powerful, and then the magical extra feature of nondeterminism comes along: how could this result in nothing new? But the conjecture does describe a fairly intriguing situation.

First, if we interpret the conjecture in the context of the struggle of fast deterministic algorithms to subdue fast nondeterministic ones, it says that a particular nondeterministic Turing machine using just one tape and only linear time [**37**] can actually solve a problem that defies all polynomial-time multi-tape deterministic Turing machines, irrespective of the degree of the associated polynomial and the number of tapes. In other words, the claim is that *nondeterministic algorithms can beat deterministic ones even with minimal use of the rest of their abilities.* This is an aspect of the P $\neq$ NP conjecture that our definitions do not highlight.

Second, if we interpret the conjecture in the context of the struggle of fast deterministic algorithms to solve a specific NP-complete problem, it says that the fastest way to check whether a propositional formula is satisfiable is essentially to try all possible assignments. Although this matches with our experience in one sense (in the simple sense that we have no idea how to do anything faster in general), it also seriously clashes with our experience in another sense, equally important: it claims that *the obvious, highly inefficient solution is also the optimal one.* This is in contrast with what one would first imagine, given that optimality is almost always associated with high sophistication.

Similar comments are valid for L vs. NL. The standard conjecture that L $\neq$ NL asserts that a particular nondeterministic finite automaton with a small bunch of states and just two heads that only move one-way [**57**] can actually solve a problem that defies all deterministic multi-head two-way finite automata, irrespective of their number of states and heads. At the same time, it claims that the most space-economical method of checking connectivity is essentially the one by Savitch, a fairly non-trivial algorithm which nevertheless still involves several natural choices (e.g., recursion) and lacks the high sophistication that we usually expect in optimality.

Similarly to P vs. NP and L vs. NL, the conjecture for the 2D vs. 2N problem is again that 2D $\neq$ 2N. Moreover, two stronger variants of it have also been proposed.

First, it is conjectured that 2NFAs can be exponentially smaller than 2DFAs even when they never move their head to the left. In other words, it is suggested that even *one-way nondeterministic finite automata* (1NFAs) can be exponentially smaller than 2DFAs. (Note the similarity with the version of L $\neq$ NL mentioned above, where the nondeterministic automaton need only move its heads one-way to beat all two-way multi-head deterministic ones.) So, once more we have an instance of the claim that nondeterministic algorithms can beat deterministic ones with minimal use of the rest of their abilities.

Second, it is even conjectured that this alleged exponential difference in size between 1NFAs and 2DFAs covers the entire gap from n to $2^n - 1$ which is known to exist between 1NFAs and 1DFAs [**35**].[2] In other words, according to this conjecture, a 2DFA trying to simulate a 1NFA may as well drop its bidirectionality, since it is going to be totally useless: its optimal strategy is going to be the well-known brute-force one-way deterministic simulation [**47**]. So, once more we have an instance of the claim that the obvious, highly inefficient solution is also the optimal one.[3]

For a concrete example of what all this means, recall the problem that we described early in this introduction (page 8). Remember that we presented it as a problem that is solvable by small 2NFAs but is conjectured to require large 2DFAs. Notice that the best 2NFA algorithm presented there is actually one-way, namely a 1NFA. So, if the conjecture about this problem is true, then 2NFAs can indeed beat 2DFAs, and they can do so without using their bidirectionality. Also notice that the best known 2DFA for that problem is one-way, too. In fact, it is simply the brute-force 1DFA simulation of the 1NFA solver. So, if this is really the smallest 2DFA for the problem, then indeed the optimal way of simulating the hardest 2NFA is the obvious, highly inefficient one.

In total, interpreting the questions on the power of nondeterminism (P vs. NP, L vs. NL) as a contest between deterministic and nondeterministic algorithms, our conjectures claim that nondeterministic algorithms can win with one hand behind their back; and then, the best that deterministic algorithms can do in their defeat to minimize their losses is essentially not to think. This *is* a counter-intuitive claim, and our conjectures for 2D vs. 2N make this same claim, too.

2.1-IV. *A mathematical connection.* Of the similarities described above between 2D vs. 2N and the more important questions on the power of nondeterminism, none is mathematical. However, a mathematical connection is known, too. As explained in [**3**], if we can establish that 2D $\neq$ 2N *using only "short" strings*, then we

[2]As with 2DFAs (cf. Footnote 1 on page 8), a 1DFA is allowed to reject by just hanging anywhere along its input. Without this freedom, the gap would actually be from n to 2^n.

[3]Note that, contrary to the strong versions of P $\neq$ NP and L $\neq$ NL mentioned above [**37, 57**], the two conjectures mentioned in these two last paragraphs may be strictly stronger than 2D $\neq$ 2N. It may very well be that 1NFAs cannot be exponentially smaller than 2DFAs, but 2NFAs can (it is known that 2NFAs can be exponentially smaller than 1NFAs). Moreover, even if 1NFAs can be exponentially smaller than 2DFAs, it may very well be that this exponential gap is smaller than the gap from n to $2^n - 1$.

would also have a proof that L $\neq$ NL. To describe this implication more carefully, we need to first discuss how a proof of 2D $\neq$ 2N may actually proceed.

To prove the conjecture, we need to start with a 2N-complete family of regular problems $\Pi = (\Pi_n)_{n\geq 0}$, and prove that it is not in 2D. That is, we must prove that for any polynomial-size family of 2DFAs $D = (D_n)_{n\geq 0}$ there exists an n which is *bad*, in the sense that D_n does not solve Π_n. Now, two observations are due:

- To prove that some n is bad, we need to find a string w_n that "fools" D_n, in the sense that $w_n \in \Pi_n$ but D_n rejects w_n, or $w_n \notin \Pi_n$ but D_n accepts w_n.
- Every D has a bad n iff every D has infinitely many bad n. This is true because, if a polynomial-size family D has only finitely many bad n, then replacing the corresponding D_n with correct automata of any size would result in a new family which is still polynomial-size and has no bad n.

Hence, proving the conjecture amounts to proving that, for any polynomial-size family D of 2DFAs for Π, there is a family of strings $w = (w_n)_{n\geq 0}$ such that, for infinitely many n, the input w_n fools D_n.

Now, the connection with L vs. NL says the following: if we can indeed find such a proof and *in addition* manage to guarantee that the lengths of the strings in w are bounded by some polynomial of n, then L $\neq$ NL.

There is no doubt that this connection increases our confidence in the relevance of the 2D vs. 2N problem to the more important questions on the power of nondeterminism. However, its significance should not be over-estimated. First, *the two problems may very well be resolved independently.* On one hand, if 2D = 2N then the connection is irrelevant, obviously. On the other hand, if 2D $\neq$ 2N then our tools for short strings are so much weaker than our tools for long strings, that it is hard to imagine us arriving at a proof that uses only short strings before actually having a proof that uses long ones. Second, and perhaps most importantly, *ideas do not need mathematical connections to transcend domains.* In other words, an idea that works for one type of machines may very well be applicable to other types of machines, too, even if no high-level theorem encodes this transfer. Examples of this situation include the Immerman-Szelepcsényi idea [**24, 58**], Savitch's idea [**49**], and Sipser's "rewind" idea [**54**], each of which has been applied to machines of significantly different power [**14, 54**].

In conclusion, from the computational complexity perspective, the 2D vs. 2N problem is a question on the power of nondeterminism which seems both *simple* enough to be tractable and, at the same time, *robust*, *hard*, and *intriguing* enough to be relevant to our efforts against other, more important questions of its kind.

2.2. Descriptional complexity. From the perspective of descriptional complexity, the 2D vs. 2N question falls within the general major goal of understanding the relative succinctness of language descriptors. Here, by "language descriptor" we mean any formal model for recognizing or generating strings: finite automata, regular expressions, pushdown automata, grammars, Turing machines, etc.

Perhaps the most famous question in this domain is the one about the relative succinctness of 1DFAs and 1NFAs. Since both types of automata recognize exactly the regular languages [**47**], every such language can be described both by the deterministic and by the nondeterministic version. *Which type of description is shorter?* Or, measuring the size of these descriptions by the number of states in the corresponding automata, which type of automaton needs the fewest states? Clearly, since determinism is a special case of nondeterminism, a smallest 1DFA cannot be smaller than a smallest 1NFA. So, the question really is: *How much larger than a smallest* 1NFA *need a smallest* 1DFA *be?* By the well-known simulation of [**47**], we know that every n-state 1NFA has an equivalent 1DFA with at most $2^n - 1$ states.[4] Moreover, this simulation is optimal [**35**], in the sense that certain n-state 1NFAs have no equivalent 1DFA with fewer than $2^n - 1$ states. Hence, this question of descriptional complexity is fully resolved: if the minimal 1NFA description of a regular language is of size n, then the corresponding minimal 1DFA description is of size at most $2^n - 1$, and sometimes is exactly that big.

There is really no end to the list of questions of this kind that can be asked. For the example of finite automata alone, we can change how the size of the descriptions is measured (e.g., use the number of transitions) and/or the resource that differentiates the machines (e.g., use any combination of nondeterminism, bidirectionality, ambiguity, alternation, randomness, pebbles, heads, etc.). Moreover, the models being compared can even be of completely different kind (e.g., 1NFAs versus regular expressions) and/or have different power (e.g., 1NFAs versus deterministic pushdown automata, or context-free grammars), in which case each model may have its own measure for the size of descriptions.

Typically, every question of this kind is viewed in the context of a *conversion.* For example, the question about 1DFAs and 1NFAs is viewed as follows:

Given a 1NFA, we must convert it into a smallest equivalent 1DFA.
What is the increase in the number of states in the worst case?

In other words, starting with a 1NFA, we want to trade size for determinism and we would like to know in advance the worst possible loss in size. We encode this information into a function f, called the *trade-off* of the conversion: for every n, $f(n)$ is the least upper bound for the new number of states when an arbitrary n-state 1NFA is converted into a smallest equivalent 1DFA. In this terminology, our previous discussion can be summarized into the following concise statement:

the trade-off from 1NFAs to 1DFAs is $f(n) = 2^n - 1$.

Note that this encodes both the simulation of [**47**], by saying that $f(n) \leq 2^n - 1$, and the "hard" 1NFAs of [**35**], by saying that $f(n) \geq 2^n - 1$.

The 2D vs. 2N problem can also be concisely expressed in these terms. It concerns the conversion from 2NFAs to 2DFAs, where again we trade size for determinism, and precisely asks whether the associated trade-off can be upper-bounded by some polynomial:

2D = 2N $\iff$ the trade-off from 2NFAs to 2DFAs is polynomially bounded.

Indeed, if the trade-off is polynomially bounded, then every family of regular problems that is solvable by a polynomial-size family of 2NFAs $N = (N_n)_{n\geq 0}$ is also solvable by a polynomial-size family of 2DFAs: just convert N_n into a smallest equivalent 2DFA D_n, and form the resulting family $D := (D_n)_{n\geq 0}$. Since the size s_n of N_n is bounded by a polynomial in n and the size of D_n is bounded by a polynomial in s_n (the trade-off bound), the size of D_n is also bounded by a polynomial in n. Overall, 2D = 2N. Conversely, suppose the trade-off is not polynomially bounded. For every n, let N_n be any of the n-state 2NFAs that cause the value of the trade-off for n, and let D_n be a smallest equivalent 2DFA. Then the sizes of the

[4]Recall that a 1DFA may reject by hanging anywhere along its input (cf. Footnote 2 on page 12).

$$\begin{tikzcd}
 & \text{2NFA} \arrow[dl, dashed, bend right] \arrow[dd, bend left, "c"] \arrow[dr, bend left, "d"] & \\
\text{2DFA} \arrow[ur, "e"] \arrow[rr, "d"] \arrow[dr, bend right, "b"'] & & \text{1NFA} \arrow[ul, "e"'] \arrow[ll, dashed, bend left] \arrow[dl, bend left, "a"] \\
 & \text{1DFA} \arrow[uu, "e"] \arrow[ul, "e"] \arrow[ur, "e"'] &
\end{tikzcd}$$

$$a = 2^n - 1$$
$$b = n\big(n^n - (n-1)^n\big)$$
$$c = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \tbinom{n}{i}\tbinom{n}{j}\left(2^i - 1\right)^j$$
$$d = \tbinom{2n}{n+1}$$
$$e = n$$

FIGURE 1. The 12 conversions defined by nondeterminism and bidirectionality in finite automata, and the known exact trade-offs.

automata in the family $D := (D_n)_{n\geq 0}$ are exactly the values of the trade-off, and therefore D is not of polynomial size. Moreover, for Π_n the language recognized by N_n and D_n, the family $\Pi := (\Pi_n)_{n\geq 0}$ is clearly in 2N (because of the linear-size family $N := (N_n)_{n\geq 0}$) but not in 2D (since D is not of polynomial size). Overall, 2D $\neq$ 2N.

Note the sharp difference in our understanding of the two conversions mentioned so far. On the one hand, our understanding of the conversion from 1NFAs to 1DFAs is perfect: we know the *exact* value of the associated trade-off. On the other hand, our understanding of the conversion from 2NFAs to 2DFAs is minimal: not only do we not know the exact value of the associated trade-off, but we cannot even tell whether it is polynomial or not. The best known upper bound for it is exponential, while the best known lower bound is quadratic. In fact, the details of this gap reveal a much more embarrassing ignorance. The exponential upper bound is the trade-off from 2NFAs to 1DFAs, while the quadratic lower bound is the trade-off from unary 1NFAs to 2DFAs. In other words, put in the shoes of a 2DFA that tries to simulate a 2NFA, we have no idea how to benefit from our bidirectionality; at the same time, put in the shoes of a 2NFA that tries to resist being simulated by a 2DFA, we have no idea how to use our bidirectionality or our ability to distinguish between different tape symbols.

A bigger picture is even more peculiar. The 12 arrows in Figure 1 show all possible conversions that can be performed between the four most fundamental types of finite automata: 1DFAs, 1NFAs, 2DFAs, and 2NFAs. For 10 of these conversions, the problem of finding the exact value of the associated trade-off has been completely resolved (as shown in the figure), and therefore our understanding of them is perfect. The only two that remain unresolved are the ones from 2NFAs and 1NFAs to 2DFAs (as shown by the dashed arrows), that is, exactly the conversions associated with 2D vs. 2N.

In conclusion, from the descriptional complexity perspective, the 2D vs. 2N problem represents the *last two open questions* about the relative succinctness of the basic types of automata defined by nondeterminism and bidirectionality. Moreover, the contrast in our understanding between these two questions and the remaining ten is the sharp contrast between *minimal* and *perfect* understanding.

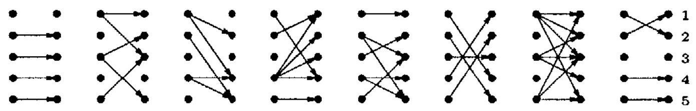
3. Progress

Our progress against the 2D vs. 2N question has been in two distinct directions: we have proved lower bounds for automata of *restricted information* and for automata of *restricted bidirectionality*. In both cases, our theorems involve a particular computational problem called *liveness*. We start by describing this problem.

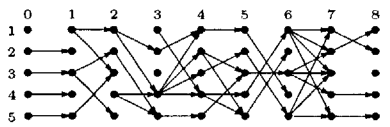
3.1. Liveness. As already mentioned in Section 2.1-III, we currently believe that 2NFAs can be exponentially smaller than 2DFAs even without using their bidirectionality. That is, we believe that even 1NFAs can be exponentially smaller than 2DFAs. In computational complexity terms, this is the same as saying that the reason why 2D $\not\supseteq$ 2N is because already 2D $\not\supseteq$ 1N, where 1N is the class of families of regular problems that can be solved by polynomial-size families of 1NFAs. In descriptional complexity terms, this is the same as saying that the reason why the trade-off from 2NFAs to 2DFAs is not polynomially bounded is because already the trade-off from 1NFAs to 2DFAs is not. In this thesis, we focus on this stronger conjecture.

As in any attempt to show non-containment of one complexity class into another (P $\not\supseteq$ NP, L $\not\supseteq$ NL), it is important to know specific complete problems—namely, problems which witness the non-containment iff the non-containment indeed holds. In our case, we need a family of regular problems that can be solved by a polynomial-size family of 1NFAs and, in addition, they are such that no polynomial-size family of 2DFAs can solve them iff 2D $\not\supseteq$ 1N. Such families are known. In fact, we have already presented one: the family of problems defined on page 8 over the alphabets Δ_n. So, it is safe to invest all our efforts in trying to understand that particular family, and prove or disprove the 2D $\not\supseteq$ 1N conjecture by showing that the family does not or does belong to 2D. However, it is easier (and as safe) to work with another complete family, which is defined over an even larger alphabet and thus brings us closer to the combinatorial core of the conjecture. This family is called *liveness*, denoted by $B = (B_n)_{n\geq 0}$, and defined as follows [48].

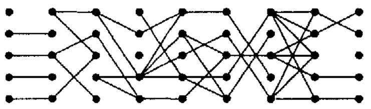
For each n, we consider the alphabet $\Sigma_n := \mathcal{P}(\{1,2,\dots,n\}^2)$ of all directed 2-column graphs with n nodes per column and only rightward arrows. For example, for $n = 5$ this alphabet includes the symbols:



where, e.g., indexing the vertices from top to bottom, the rightmost symbol is $\{(1,2),(2,1),(4,4),(5,5)\}$. Given an m-long string over Σ_n, we naturally interpret it as the representation of a directed $(m+1)$-column graph, the one that we get by identifying the adjacent columns of neighboring symbols. For example, for $m = 8$ the string of the above symbols represents the graph:



where columns are indexed from left to right starting from 0. In this graph, a *live path* is any path that connects the leftmost column to the rightmost one (i.e., the 0th to the mth column), and a *live vertex* is any vertex that has a path from the leftmost column to it. The string is *live* if live paths exist; equivalently, if the rightmost column contains live vertices. Otherwise, the string is *dead*. For example, in the above string, the 5th node of the 2nd column is live because of the path $3 \rightarrow 3 \rightarrow 5$, and the string is live because of two live paths, one of which is $3 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 5 \rightarrow 3 \rightarrow 3 \rightarrow 2 \rightarrow 1$. Note that no information is lost if we drop the direction of the arrows, and we do. So, the above string is simply:



The problem B_n consists in determining whether a given string over Σ_n^* is live or not. In formal dialect, we define $B_n := \{w \in \Sigma_n^* \mid w \text{ is live }\}$, for all n.

As already claimed, $B \in$ 1N. That is, there exist small 1NFA algorithms for B_n. The smallest possible one is rather obvious:

> We scan the list of graphs from left to right, trying to nondeterministically follow one live path. Initially, we guess the starting vertex among those of the leftmost column. Then, on reading each graph, we find which vertices in the next column are accessible from the most recent vertex. If none is, we hang (in this branch of the nondeterminism). Otherwise, we guess one of them and move on remembering only it. If we ever arrive at the end of the input, we accept.

It is easy to verify that this algorithm can be implemented on a 1NFA with exactly one state per possible vertex in a column. Hence, B_n is solvable by an n-state 1NFA. In contrast, nobody knows how to solve B_n on a 2DFA with fewer than $2^n - 1$ states. The best known 2DFA algorithm is the following:

> We scan the list of graphs from left to right, remembering only the set of live vertices in the most recent column. Initially, all vertices of the leftmost column are live. Then, on reading each graph, we use its arrows to compute the set of live vertices in the next column. If it is empty, we simply hang. Otherwise, we move on, remembering only this set. If we ever arrive at the end of the input, we accept.

Easily, this algorithm needs exactly one state per possible non-empty set of live vertices in a column, for a total of $2^n - 1$ states, as promised.

By the completeness of B, our questions about the relation between 2D and 1N can be encoded into questions about the size of a 2DFA solving B_n. In other words, the following three statements are equivalent:

- 2D $\supseteq$ 1N,
- the trade-off from 1NFAs to 2DFAs is polynomially bounded,
- B_n can be solved by a 2DFA of size polynomial in n.

Hence, to prove the conjecture that 2D $\not\supseteq$ 1N, we just need to prove that the number of states in every 2DFA solving B_n is super-polynomial in n. In fact, as explained in Section 2.1-III, a stronger conjecture says that the above 2DFA algorithm is optimal! That is, in every 2DFA solving B_n—the conjecture goes—the number of states is not only super-polynomial but already $2^n - 1$ or bigger. To better understand what this means, observe that the above algorithm is one-way: it is, in fact, the smallest 1DFA for liveness (as we can easily prove). Therefore, the claim is that *in solving liveness, a* 2DFA *has no way of using its bidirectionality to save even* 1 *of the* $2^n - 1$ *states that are necessary without it.*

3.2. Restricted information: Moles. The first direction that we explore in our investigation of the efficiency of 2DFAs against liveness is motivated by the particular way of operation of the 1NFA algorithm that we described above.

Specifically, consider any branch of the nondeterministic computation of that 1NFA. Along that branch, the automaton moves through the input from left to right, reading one graph after the other. However, although at every step the entire next graph is read, only part of its information is used. In particular, the automaton 'focuses' only on one of the vertices in the left column of the graph and 'sees' only the arrows which depart from that vertex. The rest of the graph is ignored. In this sense, the automaton operates in a mode of 'restricted information'.

A more intuitive way to describe this mode of operation is to view the input string as a 'network of tunnels' and the 1NFA as an n-state one-way nondeterministic robot that explores this network. Then, at each step, the robot reads only the index of the vertex that it is currently on and the tunnels that depart from that vertex, and has the option to either follow one of these tunnels or abort, if none exists. In yet more intuitive terms, the automaton behaves like an n-state one-way nondeterministic *mole*.

Given this observation, a natural question to ask is the following: Suppose we apply to this mole the same conversion that defines the question whether $2D \supseteq 1N$. Namely, suppose that this mole loses its nondeterminism in exchange for bidirectionality. How much larger does it need to get to still be solving B_n? That is, *can liveness be solved by a small two-way deterministic mole?* Equivalently, is there a 2DFA algorithm that can tell whether a string is live or not by simply exploring the graph defined by it? Note that, at first glance, there is nothing to exclude the possibility of some clever graph exploration technique that correctly detects the existence of live paths and can indeed be implemented on a small 2DFA.

In Chapter 2 we prove that the answer to this question is strongly negative:

no two-way deterministic mole can solve liveness.

To understand the value of this answer, it is necessary to understand both the "good news" and the "bad news" that it contains.

The good news is that we have crossed an entire, very natural class of 2DFA algorithms off the list of candidates against liveness. We have thus come to know that every correct 2DFA must be using the information of every symbol in a more complex way than moles.

However, note that our answer talks of *all* two-way deterministic moles, as opposed to only *small* ones. This might sound like "even better news", but it is actually bad. Remember that our primary interest is not moles themselves, but rather the behavior of small 2DFAs against liveness. So, our hope was that we would get an answer that involves small moles, and this hope did not materialize. Put another way, we asked a complexity-theoretic question and we received a computability-theoretic answer.

Overall, our understanding has indeed advanced, but not for the class of machines that we were mostly interested in. Nevertheless, some of the tools developed for the proof of this theorem may still be useful for the more general goal. Specifically, if indeed small 2DFAs cannot solve liveness, then it is hard to imagine a proof that will not involve very long inputs. Such a proof will probably need tools similar to the *dilemmas* and *generic strings for* 2DFAs that were used in our argument.

3.3. Restricted bidirectionality: Sweeping automata. The second direction that we explore is motivated by the known fact that 2D is closed under complement [**54, 14**], whereas the corresponding question for 2N is open. So, one way to prove that 2D $\neq$ 2N is to show that 2N is *not* closed under complement. In terms of classes, we can write this goal as 2N $\neq$ co2N, where co2N is the class of families of regular problems whose complements can be solved by polynomial-size families of 2NFAs. Of course, it is conceivable that 2N = co2N, in which case a proof of this would be evidence that 2D = 2N.

As a matter of fact, 2N = co2N is already known to hold in some special cases. First, the analogue of this question for logarithmic-space Turing machines is known to have been resolved this way: NL = coNL [**24, 58**]. By the argument of [**3**], this implies that every small 2NFA can be converted into a small 2NFA that makes exactly the opposite decisions on all "short" inputs (in the sense of Section 2.1-IV). In addition, the proof idea of NL = coNL has been used to prove that indeed 2N = co2N for the case of unary regular problems [**14**]. So, 2N and co2N are already known to coincide on *short* and on *unary* inputs.

However, there is little doubt that the above special cases avoid the core of the hardness of the 2N vs. co2N question. In this sense, our confidence in the conjecture that 2N $\neq$ co2N is not seriously harmed. As a matter of fact, in Chapter 2 we prove a theorem that constitutes evidence for it. We consider a restriction on the bidirectionality of the 2NFAs and prove that, under this restriction, 2N $\neq$ co2N. The restricted automata that we consider are the "sweeping" 2NFAs.

A two-way automaton is called *sweeping* if its input head can change the direction of its motion only on the two ends of the input. In other words, each computation of a sweeping automaton is simply a sequence of one-way passes over the input, with alternating direction. We use the notation SNFA for sweeping 2NFAs, and SN for the class of families of regular problems that can be solved by polynomial-size families of SNFAs. With these names, our theorem says that:

$$\text{SN} \neq \text{coSN}.$$

More specifically, our proof uses liveness, which is obviously in SN: $B \in$ SN. We prove that, in contrast, *every* SNFA *solving the complement of* B_n *needs* $2^{\Omega(n)}$ *states*, so that $B \notin$ coSN. Overall, $B \in$ SN $\setminus$ coSN and the two classes are different.

Another way to interpret this theorem is to view it as a generalization of two other, previously known facts about the complement of liveness: that it is not solvable by small 1NFAs [**48**] and that it is not solvable by small sweeping 2DFAs [**55, 14**], either. So, proving the same for small SNFAs amounts to generalizing both these facts to sweeping bidirectionality and to nondeterminism, respectively. For another interesting interpretation, note that the smallest known SNFA solving the complement of B_n is still the obvious 2^n-state 1DFA from page 17. Hence, our theorem says that, even after allowing sweeping bidirectionality and nondeterminism *together*, a 1DFA can still not achieve significant savings in size against the complement of liveness—whether it can save even 1 state is still open.

Finally, our proof can be modified so that all strings used in it are drawn from a special subclass of Σ_n^* on which the complement of liveness can actually be determined by a small 2DFA. This immediately implies that:

the trade-off from 2DFAs to SNFAs is exponential,

which generalizes a known similar relation between 2DFAs and SDFAs [**55, 2, 36**].

4. Other Problems in This Thesis

Apart from the progress against the 2D vs. 2N question explained above, this thesis also contains a few other, related theorems in descriptional complexity.

4.1. Exact trade-offs for regular conversions. As explained in Section 2.2 (Figure 1), the 2D vs. 2N question concerns only 2 of the 12 possible conversions between the four most basic types of finite automata (1DFAs, 1NFAs, 2DFAs, and 2NFAs). For each of the remaining conversions our understanding is perfect, in the sense that we know the exact value of the associated trade-off.

For the conversion from 1NFAs to 1DFAs (Figure 1a), the upper bound is due to [**47**] and the lower bound due to [**35**]. For any of the conversions from weaker to stronger automata (Figure 1e), the upper bound is obvious by the definitions and the lower bound is due to [**6**]. For the remaining four conversions (from 2NFAs or 2DFAs to 1NFAs or 1DFAs), both the upper and lower bounds are due to this thesis—although the fact that the trade-offs were exponential was known before. We establish these exact values in Chapter 1. For a quick look, see Figure 1b–d.

We stress, however, that the exact values alone do to reveal the depth of the understanding behind the associated proofs. In order to explain what we mean by this, let us revisit the conversion from 1NFAs to 1DFAs. As already mentioned, we can encode our understanding of this conversion into the concise statement that:

the trade-off from 1NFAs to 1DFAs is exactly $2^n - 1$.

A less succinct but more informative description is that, for all n:

- every n-state 1NFA has an equivalent 1DFA with at most $2^n - 1$ states, and
- some n-state 1NFA has no equivalent 1DFA with fewer than $2^n - 1$ states.

But even these more verbose statements fail to describe the kind of understanding that led to them. What we really know is that *every* 1NFA *N can be simulated by a* 1DFA *that has* 1 *distinct state for each non-empty subset of states of N which* (as an instantaneous description of *N*) *is both realizable and non-redundant.* This is exactly the idea where everything else comes from: *the value* $2^n - 1$ (by a standard counting argument), *the simulation* for the upper bound (just construct a 1DFA with these states and with the then obvious transitions), and *the hard instances* for the lower bound (just find 1NFAs that manage to keep all of their instantaneous descriptions realizable and non-redundant). In this sense, we know more than just the value of the trade-off; we know the precise, single reason behind it:

the non-empty subsets of states of the 1NFA

that is being converted. To be able to pin down the exact source of the difficulty of a conversion in terms of such a simple and well-understood class of set-theoretic objects is a rather elegant achievement.

Our analyses in Chapter 1 are supported by this same kind of understanding: in each one of the four trade-offs that we discuss, we first identify the correct set-theoretic object at the core of the conversion and then move on to extract from it the exact value, the simulation, and the hard instances that we need. As a foretaste, here are the objects at the core of the conversion from 2NFAs to 1NFAs:

the pairs of subsets of states of the 2NFA being converted, where
the second subset has exactly 1 more state than the first subset.

So, every 2NFA can be simulated by a 1NFA that has 1 distinct state for every such pair, and for some 2NFAs all these states are necessary. Moreover, the value of the trade-off is exactly the number of such pairs that we can construct out of an n-state 2NFA; a standard counting argument shows that this number is $\binom{2n}{n+1}$.

4.2. Non-recursive trade-offs for non-regular conversions. In contrast to Chapters 1 and 2, the last chapter studies conversions between machines other than the automata of Figure 1, including machines that can also recognize non-regular problems. As we shall see, the trade-offs for such conversions may, in general, behave in a quite different manner.

To understand the difference, note that already since [**53**] we knew how to effectively convert any 2NFA (the strongest type of automata in Figure 1) into a 1DFA (the weakest type). This immediately guaranteed a recursive upper bound for each one of the 12 trade-offs of Figure 1. In contrast, for other conversions, such a recursive upper bound cannot be taken for granted. As first shown in [**35**], there are cases where the trade-off of a conversion grows faster than any recursive function: e.g., the conversion from one-way nondeterministic pushdown automata that recognize regular languages to 1DFAs. Moreover, this phenomenon cannot be attributed simply to the difference in power between the types of the machines involved. As shown in [**56**], if the pushdown automata in the previous conversion are deterministic, then the trade-off does admit a recursive upper bound. Such trade-offs, that cannot be recursively bounded, are simply called *non-recursive.* Note that this name is slightly misleading, as it allows the possibility of a non-recursive trade-off that still admits recursive upper bounds. However, no such cases will appear in this thesis.

In Chapter 3 we refine a well-known technique [**16**] to prove a general theorem that implies the non-recursiveness of the trade-off for a list of conversions involving two-way machines. Roughly speaking, our theorem concerns any two types of machines, A and B, that satisfy the following two conditions:

- the A machines can solve problems that no B machine can solve, and
- the A machines can simulate any two-way deterministic finite automaton that works on a unary alphabet and has access to a linearly-bounded counter.

For any such pair of types, our theorem says that the trade-off from A machines to B machines is non-recursive. For example, we can have A be the multi-head finite automata with $k+1$ heads and B be the multi-head finite automata with k heads. No matter what k is, the conditions are known to be true, and therefore replacing a multi-head finite automaton with an equivalent one that has 1 fewer head results in an non-recursive increase in the size of the automaton's description, in general.

At the core of the argument of this theorem lies a lemma of independent interest: we prove that the emptiness problem remains unrecognizable (non-semidecidable) even for a unary two-way deterministic finite automaton that has access to a linearly-bounded counter and *obeys a threshold*—in the sense that it either rejects all its inputs or accepts exactly those that are longer than some fixed length.

CHAPTER 1

Exact Trade-Offs

In this chapter we prove the exact values of the trade-offs for the conversions from two-way to one-way finite automata, as pictured in Figure 1 (page 15). In Section 3 we cover the conversion from 2DFAs to 1DFAs (Figure 1b), whereas the conversion from 2NFAs to 1DFAs (Figure 1c) is the subject of Section 4. The conversions from 2NFAs and 2DFAs to 1NFAs (Figure 1d) are covered together in Section 5. We begin with a short note on the history of the subject and a summary of our conclusions.

1. History of the Conversions

The conversion from 1NFAs to 1DFAs is the archetypal problem of descriptional complexity. As already mentioned (Figure 1*a*), the problem is fully resolved, in the sense that we know the exact value of the associated trade-off:[1]

the trade-off from 1NFAs to 1DFAs is $2^n - 1$.

The history of this problem began in the late 50's, when Rabin and Scott [**46, 47**] introduced 1NFAs as a generalization of 1DFAs and showed how 1DFAs can simulate them. This proved the upper bound for the trade-off. The matching lower bound was established much later, via several examples of "hard" 1NFAs [**44, 35, 43, 51, 48, 33**].[2] Both bounds are based on the crucial idea that

the non-empty subsets of states of the 1NFA

capture everything that a simulating 1DFA needs to describe with its states.

As part of the same seminal work [**45, 47**], Rabin and Scott also introduced *two-way automata* and proved "to their surprise" that 1DFAs were again able to simulate their generalization. This time, though, the proof was complicated enough to be superseded by a simpler proof by Shepherdson [**53**] at around the same time. All authors were actually talking about what we would now call *single-pass two-way deterministic finite automata* (ZDFAs), as their definitions did not involve end-markers. However, the automata quickly grew into full-fledged *two-way deterministic finite automata* (2DFAs) and also into nondeterministic counterparts (ZNFAs and 2NFAs), while all theorems remained valid or easily adjustable.

Naturally, the descriptive complexity questions arose again. Shepherdson mentioned that, according to his proof, every n-state 2DFA had an equivalent 1DFA with at most $(n+1)^{(n+1)}$ states. Had he cared for his bound to be tight, he would surely

[1]Recall our conventions, as explained in Footnote 2 on page 12.

[2]The earliest ones, both over a binary alphabet, appeared in [**35**] (an example that was described as a simplification of one contained in an even earlier unpublished report [**44**]) and in [**43**] (where [**44**] is again mentioned as containing a different example with similar properties). Other examples have also appeared, over both large [**51, 48**] and binary alphabets [**33**]. A more natural but not optimal example was also mentioned in [**35**] and attributed to Paterson.

have noted that his proof had actually established an upper bound of only $n(n+1)^n$ —e.g., see [**20**, Section 3.7]. Many years later, Birget [**6**, Theorem A3.4] claimed that this upper bound is really just n^n. On the other hand, towards a lower bound, several authors showed that the trade-off is at least exponential [**1, 35, 55**] and indeed very close to the upper bound given by Shepherdson [**35, 43**].[3]

Here we will prove that both the upper and lower bounds meet at the value $n\big(n^n - (n-1)^n\big)$. We will thus have arrived at the conclusion that

the trade-off from 2DFAs to 1DFAs is $n\big(n^n - (n-1)^n\big)$.

Note that this value is larger than the upper bound n^n claimed by Birget [**6**]. Indeed, his argument contained an oversight. But it did contain the correct idea, and it is exactly that idea which we apply here. We also note that our lower bound is valid even when the 2DFA being converted is single-pass. More importantly, both the upper and the lower bound are derived in a straightforward manner after we carefully identify the correct set-theoretic objects that 'live' in the relationship between the computations of 2DFAs and 1DFAs. These are

the *tables* of the 2DFA

as defined in Section 2.1-II. No big surprise is to be anticipated: we simply follow the idea of [**6**] in properly restricting the functions used in [**53**, proof of Theorem 2].

The relation between the most and least powerful of all automata mentioned so far, namely between 2NFAs and 1DFAs, has also been examined. Via a straightforward adjustment, Shepherdson's argument could show very early that every n-state 2NFA can be converted into a 1DFA with at most $2^{n^2}(2^n - 1)$ states. Much later, Birget [**6**, Theorem A3.4] claimed it to be no more than $n\binom{n}{n/2}2^{(n-1)^2}$. Towards a lower bound, we just mention the one provided by the systematic framework of [**48**], which was $2^{(n/2-2)^2}$. Here we will show that

the trade-off from 2NFAs to 1DFAs is $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\binom{n}{i}\binom{n}{j}\left(2^i - 1\right)^j$

where the lower bound is valid even when the 2NFA being converted is single-pass. As before, we will first identify the correct set-theoretic objects that relate the computations of 2NFAs to those of 1DFAs. These are

the *tables* of the 2NFA

as defined in Section 2.2. Again, we arrive at them by appropriately restricting the functions in the Shepherdson argument.

The most interesting of the descriptive complexity questions that we consider emerge as we examine the conversions from 2NFAs to 1NFAs and 2DFAs:

(Q1) from 2NFAs to 1NFAs: *is bidirectionality essential to* 2NFAs*?* Or, is there a problem that a 1NFA would be able to solve with exponentially fewer states if it were allowed to move its head to the left?

(Q2) from 2NFAs to 2DFAs: *is nondeterminism essential to* 2NFAs*?* Or, is there a problem that a 2DFA would be able to solve with exponentially fewer states it it were allowed to make nondeterministic choices?

[3]That the lower bound is close to the upper bound given by Shepherdson was shown by (a slight modification of) the language of [**35**, Proposition 2], which requires $\geq n^n$ states on every 1DFA, but only $\leq 5n+5$ states on a 2DFA (a ZDFA, even). Similarly, [**43**] gave a language that requires $\geq n^n + o(n^n)$ states on a 1DFA, but only $\leq 2n+5$ on a 2DFA. For just an exponential separation, one can look at [1] for $\geq 2^n + 2$ and $\leq 2n+2$ states (even on a ZDFA); at the Paterson example of [**35**] for $\geq 2^n$ and $\leq n+2$ states (even on a SDFA); or at [**55**] for $\geq 2^n$ and $\leq O(n)$ (even on a SDFA).

The second question is of course the 2D vs. 2N problem, as explained in the Introduction and covered in detail in Chapter 2. For the first question, the answer is known to be positive, but here we will find the exact value of the trade-off.

For the upper bound, it is straightforward to use Shepherdson's idea to show that every n-state 2NFA has an equivalent 1NFA with at most $n2^{n^2}$ states. A more economical simulation, with fewer than $(n!)^2$ states, can be achieved by crossing sequences [**21**, Section 2.6]. However, it is not hard to observe that the order in which the pairs of successive states (after the first state) appear inside a crossing sequence is not important; equivalently, in applying Shepherdson's idea we can use the nondeterminism of the simulating 1NFA not only for 'guessing forwards' but also for 'guessing backwards'. Based on this observation, we can actually construct a 1NFA with at most $n(n+1)^n$ states. But this would still be wasting exponentially many states, as Birget [**6**] showed 8^n+2 states are always enough. On the other hand, towards a lower bound, exponential separations between 2DFAs and 1NFAs have long been known [**48, 6**], even when the 2DFAs are single-pass [**51, 8**], the best being $2^{(n-1)/2}-1$.[4]

Here, we will again show that the upper and lower bounds meet exactly at the value $\binom{2n}{n+1}$. We will thus have that

the trade-off from 2NFAs to 1NFAs is $\binom{2n}{n+1}$.

This will again be possible after we identify the correct set-theoretic objects relating the computations of 2NFAs to those of 1NFAs. These are

the *frontiers* of the 2NFA

as defined in Section 5.1. Essentially, these objects are what remains of the crossing sequences of [**21**] after we ignore not only the order of the pairs of successive states (as we did for the $n(n+1)^n$ bound above) but the correspondence between first and second components in this set of pairs.

As a matter of fact, the lower bound is valid even when the 2NFA being converted is deterministic (and single-pass, actually). This immediately implies that

the trade-off from 2DFAs to 1NFAs is $\binom{2n}{n+1}$

as well. Hence, the ability of a 2NFA to move its head in both directions strictly inside the input can *alone* cause all the hardness that a simulating 1NFA must overcome. In other words, the answer to (Q1) above is positive, exactly because even the answer to the following question is positive (Figure 1*d*):

(Q3) from 2DFAs to 1NFAs: *can bidirectionality beat nondeterminism?* Is there a problem that a 1NFA would be able to solve with exponentially fewer states if it were allowed to replace nondeterminism with bidirectionality?

Note the similarity with the conjectured resolution of (Q2) above. As explained in the Introduction, we believe that the answer to (Q2) is also positive, exactly because the answer to the following question is positive:

[4]In [**48**] a language was given that requires $\leq 2n+1$ states on a 2DFA, but $\geq 2^n-1$ states on a 1NFA. Through a different method, [**6**] found the same $2^{(n-1)/2}-1$ lower bound. Seiferas [**51**] gave a language that needs $\leq 4n+2$ states on a ZDFA, but $\geq 2^n$ states on any 1NFA, while Damanik [**8**] independently arrived at the same argument. Copying that idea, one can easily see that the restriction of the language B_n of [**48**] to strings of length 2 has similar properties ($\leq 2n$ and $\geq 2^n$ states).

(Q4) from 1NFAs to 2DFAs: *can nondeterminism beat bidirectionality?* Is there a problem that a 2DFA would be able to solve with exponentially fewer states if it were allowed to replace bidirectionality with nondeterminism?

So, it appears that in both cases, *the hardness of a simulation can stem entirely from the feature of the simulated machine that is absent in the simulating machine.*

Finally, let us also briefly discuss the conversions from weaker to stronger automata (Figure 1e). By the definitions, the trade-off for each one of them is trivially upper-bounded by n. Moreover, it is also lower bounded by n. To see why, notice that the n-th singleton unary language $\{0^{n-1}\}$ can be solved by an n-state 1DFA but no 2NFA with fewer than n states [4]. This proves that

the trade-off from 1DFAs to 2NFAs is n

and implies the same for all other conversions of this kind.

Before proving our claims, Section 2 will define the notions that we work with.

2. Preliminaries

We write $[n]$ for the set $\{1, 2, \ldots, n\}$. The special objects $\mathtt{l}$, $\mathtt{r}$, $\perp$ are used for building the *disjoint union* of two sets and the *augmentation* of a set

$$A \uplus B = (A \times \{\mathtt{l}\}) \cup (B \times \{\mathtt{r}\}) \qquad \text{and} \qquad A_\perp = A \cup \{\perp\}.$$

When A, B are *disjoint*, their *union* $A \cup B$ is also written as $A + B$ (so that $+$ can replace $\cup$ in both equations above). The size of A, the set of subsets of A, and the set of non-empty subsets of A are denoted respectively by $|A|$, $\mathcal{P}(A)$, and $\mathcal{P}'(A)$.

For Σ an alphabet, we use Σ^* for the set of all finite strings over Σ and Σ_e for $\Sigma + \{\vdash, \dashv\}$, where $\vdash$ and $\dashv$ are two special end-marking symbols. If $u \in \Sigma_e^*$ is a string, $|u|$ is its length and u_i is its i-th symbol, for all $i = 1, 2, \ldots, |u|$. By 'the i-th *boundary* of u' we mean the boundary between u_i and u_{i+1}, if $0 < i < |u|$; or the leftmost boundary of u, if $i = 0$; or the rightmost boundary of u, if $i = |u|$. (Figure 2a.) We also write u_e for the end-marked extension $\vdash u \dashv$ of u and $u_{e,i}$ for the i-th symbol $(u_e)_i$ of this extension. The empty string is denoted by ϵ.

Of the automata that we consider, the two-way deterministic ones constitute the most natural variety and are described in the next section. Section 2.2 introduces the one-way and nondeterministic cases, while Section 2.3 discusses some of the problems that we will be solving with all these machines.

2.1. Two-way deterministic finite automata.

A *two-way deterministic finite automaton* (2DFA) *over the states of a set Q and the symbols of an alphabet Σ* consists of a finite control that can represent all states in Q, a tape that can represent all symbols in Σ_e, and a read-only head. An input $w \in \Sigma^*$ is presented on the tape surrounded by the end-markers, as $\vdash w \dashv$. The automaton starts at a designated *start state*, its head reading the left end-marker $\vdash$. At every step, the symbol under the head is read; based on this symbol and the current state, the automaton selects a next state and whether to move its head left or right; it then simultaneously changes its state and moves its head accordingly. The input is *accepted* if the machine ever moves past the right end-marker $\dashv$ into a designated *final state*—this being the only case in which violating an end-marker is allowed.[5]

[5]Note the unusual conventions about end-marker violations and the position of the head at acceptance, borrowed from [6]. They make our definitions and theorems significantly nicer.

Formally, a 2DFA over Q and Σ is defined as a triple $M = (s, \delta, f)$, where $s, f \in Q$ are the *start* and the *final* states, respectively, and δ is the *transition function*, partially mapping $Q \times \Sigma_e$ to $Q \times \{\mathrm{l}, \mathrm{r}\}$. In addition, δ obeys the aforementioned restrictions about end-marker violation: on $\vdash$, it either moves the head to the right or hangs; on $\dashv$, it moves the head to the left, or hangs, or moves the head to the right and enters f.

2.1-I. *Computations.* Although M is typically started at s and on the tape cell containing the left end-marker $\vdash$, many other possibilities exist: for any string u, position i, and state q, *the computation of M when started at q on the i-th symbol of u* is the unique sequence

$$\text{COMP}_{M,q,i}(u) = \big((q_t, i_t)\big)_{0 \le t \le m}$$

with $(q_0, i_0) = (q, i)$ and $0 \le m \le \infty$, that meets the following restrictions:

- the head is always inside u, except possibly at the very end:
  $0 \le t < m \implies 1 \le i_t \le |u| \quad \& \quad m \ne \infty \implies 0 \le i_m \le |u| + 1.$
- every two successive pairs respect the transition function:
  $$0 \le t < m \implies \delta(q_t, u_{i_t}) = (q_{t+1}, d),$$
  where either $d = \mathrm{l}$ & $i_{t+1} = i_t - 1$ or $d = \mathrm{r}$ & $i_{t+1} = i_t + 1$.
- a last pair inside u exists only if the transition function allows it:
  $m \ne \infty$ & $1 \le i_m \le |u| \implies \delta(q_m, u_{i_m})$ is undefined.

We say (q_t, i_t) is the *t-th point* and 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 *hangs*, if $1 \le i_m \le |u|$; or it *hits right into q_m*, if $i_m = |u| + 1$. (Figure 2.) When $i = 1$ or $i = |u|$ we get the *left computation of M from q on u* or the *right computation of M from q on u*, respectively:

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

Finally, for $w \in \Sigma^*$, *the computation of M on w* refers to the typical usage

$$\text{COMP}_M(w) := \text{LCOMP}_{M,s}(w_e),$$

so that M accepts w iff the computation $\text{COMP}_M(w)$ hits right into f.

1. REMARK. Note that, when u is the empty string, the left computation of M from q on u is just $\text{LCOMP}_{M,q}(\epsilon) = \big((q, 1)\big)$ and therefore hits *right* into q, whereas the corresponding right computation is just $\text{RCOMP}_{M,q}(\epsilon) = \big((q, 0)\big)$ and therefore hits *left* into q.

2. REMARK. Also note that, since M can violate an end-marker only when it moves past $\dashv$ into f, a computation of M on any *end-marked* u (e.g., $\text{COMP}_M(w)$ is such a computation) can only loop, or hang, or hit right into f.

2.1-II. *Tables.* Pick any string u and suppose $\text{LCOMP}_{M,s}(u)$ hits right into some state p_u. Motivated by [**53**], we define *the table of M on u* to be the function

$$\text{TABLE}_M(u) := \ \tau : Q_\perp \to Q$$

that satisfies $\tau(\perp) := p_u$ and, for all $q \in Q$,

$$\tau(q) := \begin{cases} p & \text{if } \text{RCOMP}_{M,q}(u) \text{ hits right into } p, \\ p_u & \text{if } \text{RCOMP}_{M,q}(u) \text{ hits left, loops, or hangs.} \end{cases}$$

We stress that *the table is defined only when* $\text{LCOMP}_{M,s}(u)$ *hits right*; in all other cases, no meaning is associated with the notation $\text{TABLE}_M(u)$.

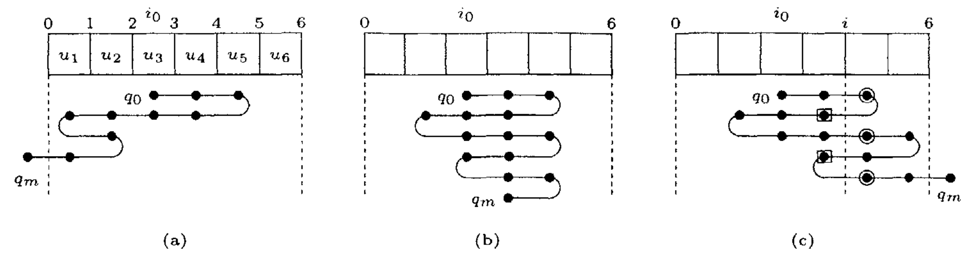


FIGURE 2. (a) Symbols and boundaries on a 6-long string u, and a computation that hits left. (b) A computation that hangs. (c) A computation c that hits right, and its i-th frontier: R_i^c in circles and L_i^c in boxes.

Note that, whenever the table of M on u is defined, it almost fully describes the behavior of the $1 + |Q|$ computations

$$\text{LCOMP}_{M,s}(u) \qquad \text{and} \qquad \text{RCOMP}_{M,q}(u), \text{ for all } q \in Q,$$

on the rightmost boundary of u, in the sense that, whenever the boundary is indeed hit, τ returns the resulting last state. The only ambiguity arises when $\tau(q) = p_u$, for some $q \in Q$: then we do not know if this is because the corresponding computation $\text{RCOMP}_{M,q}(u)$ misses the rightmost boundary, or because it hits it but it does so into p_u. If we allowed τ to take values in $Q_\perp$ (as opposed to just Q), we could easily remove this ambiguity—at the same time making our representation identical to that of [**53**]. But we will not do so. Our ultimate goal is the construction of a 1DFA that simulates M and, as we shall prove, this slightly ambiguous representation contains exactly the amount of information required for this purpose.

3. REMARK. Note that, according to our conventions (Remark 1), the table on the *empty* string $\text{TABLE}_M(\epsilon)$ is defined, and it equals the constant function s (i.e., the function that maps every element of $Q_\perp$ to the start state of M). Similarly, according to our conventions for end-marker violation (Remark 2), whenever the table $\text{TABLE}_M(u)$ on an *end-marked* u is defined, it necessarily equals the constant function f (i.e., the function that maps every element of $Q_\perp$ to the final state of M).

2.1-III. *Frontiers.* Fix some computation $c = ((q_t, i_t))_{0 \le t \le m}$ of M and consider the i-th boundary of the string being read (Figure 2c). The computation crosses this boundary 0 or more times, each crossing being either in the left-to-right or in the right-to-left direction. Collect into set R_i^c all states that result from a rightward crossing and do the same for the leftward crossings to get set L_i^c:

$$R_i^c := \{q_{t+1} \mid 0 \le t < m \ \ \& \ \ i_t = i \ \ \& \ \ i_{t+1} = i+1\},$$
$$L_i^c := \{q_{t+1} \mid 0 \le t < m \ \ \& \ \ i_t = i+1 \ \ \& \ \ i_{t+1} = i\},$$

also making the special provision that $R_{i_0-1}^c$ necessarily contains q_0.[6] The pair (L_i^c, R_i^c) partially describes the behavior of c over the i-th boundary and we call it the *i-th frontier of c*.

[6]This reflects the convention that the starting state of any computation is considered to be the result of an 'invisible' left-to-right step.

Note that the description is indeed partial, as the pair contains no information about the order in which c exhibits the states around the i-th boundary, and says nothing about the number of times each individual state is exhibited. For a full description we would need instead the *i-th crossing sequence of c* (e.g., as defined in [**21**]). However, in certain interesting cases, the extra information provided by the complete description is redundant. In particular, if we only care to decide reachability between two points via cycle-free computations, then the computations' frontiers contain exactly the amount of information that we need. We will prove and use this in Section 5.

2.2. Nondeterministic, one-way, and single-pass variations. If in the definition of a 2DFA $M = (s, \delta, f)$ more than one next moves are allowed at each step, we say the automaton is *nondeterministic* (2NFA). This formally means that δ *totally* maps $Q \times \Sigma_e$ to the *powerset* of $Q \times \{\mathrm{l}, \mathrm{r}\}$ and implies that $C := \text{COMP}_{M,q,i}(u)$ is now a *set* of computations. If then

$$P := \{p \mid \text{some } c \in C \text{ hits right into } p\},$$

we say that C *hits right into P*. Note that, if u is end-marked, then P is either $\emptyset$ or $\{f\}$ (cf. Remark 2). An input string $w \in \Sigma^*$ is considered to be accepted iff the set of computations $\text{COMP}_M(w) = \text{LCOMP}_{M,s}(w_e)$ hits right into $\{f\}$.

If the head of M never moves to the left, we say M is *one-way* (a 1NFA; or a 1DFA, if M is deterministic).[7] If no computation of M 'continues after reaching an end-marker', we say M is *single-pass* (a ZNFA; or a ZDFA).

2.2-I. *Tables of a* 2NFA. If M is a 2NFA and $u \in \Sigma_e^*$ is any string, we can define *the table of M on u* similarly to what we did for 2DFAs in Section 2.1-II. In particular, the table is defined only if the set of computations $\text{LCOMP}_{M,s}(u)$ hits right into some $P_u \neq \emptyset$, and is then the function

$$\text{TABLE}_M(u) := T : Q_\perp \to \mathcal{P}'(Q)$$

that satisfies $T(\perp) := P_u$ and, for all $q \in Q$,

$$T(q) := \begin{cases} P \setminus P_u & \text{if } \text{RCOMP}_{M,q}(u) \text{ hits right into some } P \not\subseteq P_u, \\ P_u & \text{if } \text{RCOMP}_{M,q}(u) \text{ hits right into some } P \subseteq P_u. \end{cases}$$

Note that the definition is consistent with the one for the deterministic case.[8] Moreover, it suffers the same ambiguities: whenever $T(q) = P_u$ we do not know if this is because all computations in $\text{RCOMP}_{M,q}(u)$ miss the right boundary or because some of them hit it but do so only into states that are already in P_u.

4. REMARK. Also note that an analogue of Remark 3 is true. If u is the empty string, then T is defined, and it equals the constant function $\{s\}$. If u is *end-marked*, then T is either undefined or equal to the constant function $\{f\}$.

[7]Note that, under this definition, a one-way finite automaton works on an *end-marked* input, a deviation from the standard definition. However, it is easy to verify that an automaton that follows either definition can be converted into an equivalent automaton that follows the other definition and has the same set of states. So, all our conclusions concerning the numbers of states in different automata will be valid irrespective of which definition we have in mind.

[8]If the 2NFA M is actually deterministic and T, τ are its tables on u as described by the definitions for 2NFAs and 2DFAs respectively, then T is defined iff τ is. Moreover, when both tables are defined, we have $T(r) = \{\tau(r)\}$ for all $r \in Q_\perp$.

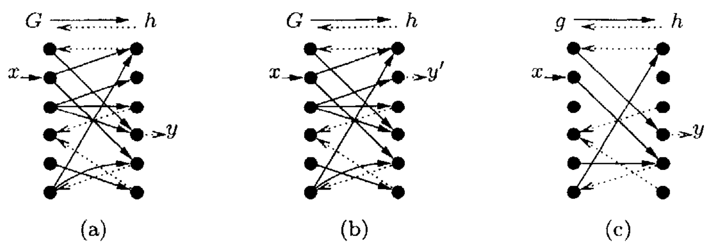


FIGURE 3. (a) A nice input, that has a path. (b) An nice input with no path. (c) A deterministic nice input, that has a path.

2.3. Problems. Given any alphabet Σ, a *(promise) problem over* Σ is any pair $\Pi = (\Pi_{\text{yes}}, \Pi_{\text{no}})$ of disjoint subsets of Σ^*. An automaton *solves* Π iff it accepts *every* $w \in \Pi_{\text{yes}}$ but *no* $w \in \Pi_{\text{no}}$. Note that the behavior of an automaton on strings outside $\Pi_{\text{yes}} + \Pi_{\text{no}}$ does not affect whether it solves Π or not. When there are no such strings, namely when $\Pi_{\text{no}} + \Pi_{\text{no}} = \Sigma^*$, the problem is also called *language* and is adequately described by Π_{yes} alone (since then $\Pi_{\text{no}} = \overline{\Pi_{\text{yes}}}$).

We will be interested in problems over the alphabet that contains pairs of the form (x, d) or (G, d), where x is a number in $[n]$, G is a binary relation on $[n]$, and d is a direction tag from $\{\mathtt{l}, \mathtt{r}\}$. In other words, our alphabet is

$$\Gamma := (\ [n] + \mathcal{P}([n] \times [n])\ ) \times \{\mathtt{l}, \mathtt{r}\}.$$

However, among all strings over Γ, we will only care about those that have length 4 and happen to follow the specific pattern

$$(x, \mathtt{l})(G, \mathtt{l})(h, \mathtt{r})(y, \mathtt{r}) \tag{1}$$

where x and y are two numbers in $[n]$, G is a binary relation on $[n]$, and h is a partial function from $[n]$ to $[n]$ which is not defined on y. (Note that a partial function is a special case of a binary relation. So, these symbols do exist in Γ.) We call these strings *nice inputs.*

Intuitively, given a nice input as above, we think of the two-column graph of Figure 3a, where the columns are two copies of $[n]$, the arrows between the columns are determined by G (left-to-right) and h (right-to-left), and the two special nodes are determined by x (entry point) and y (exit point). On this graph, a path from the entry point to the exit point may or may not exist; if it does, we just say that 'the (graph of the) input has a path'. For example, the nice input of Figure 3a has a path, but the nice input of Figure 3b does not have a path.

What makes nice inputs interesting is that a 2NFA can decide whether such an input has a path or not using only n states and a single pass over the input. More precisely, consider the promise problem $\Phi = (\Phi_{\text{yes}}, \Phi_{\text{no}})$ with

$$\Phi_{\text{yes}} := \{w \in \Gamma^* \mid w \text{ is a nice input that has a path}\},$$
$$\Phi_{\text{no}} := \{w \in \Gamma^* \mid w \text{ is a nice input that has no path}\}.$$

Then Φ can be solved by a ZNFA N_0 that has $[n]$ as its set of states and implements the following natural algorithm:

> On a nice input like (1) surrounded by end-markers, we use the first 2 steps to reach $(G, \mathtt{l})$ at state x. Then we repeatedly and alternately read the two middle symbols, each time selecting nondeterministically

and following one of the arrows defined by G (if any) or following the (at most one) arrow defined by h. If we ever reach (h, r) at a state z from which no h-arrow departs, we stay at z and move right to check whether $z = y$. If so, we move 2 more steps to the right and accept.

Formally, $N_0 := (1, \delta, 1)$, where δ is any total function from $[n] \times \Gamma_\mathrm{e}$ to the powerset of $[n] \times \{1, \mathrm{r}\}$ that satisfies the following equations:

$$\delta(1, \vdash) = \{(1, \mathrm{r})\}, \qquad \delta\big(1, (x, 1)\big) = \{(x, \mathrm{r})\}, \qquad \delta(1, \dashv) = \{(1, \mathrm{r})\},$$
$$\delta\big(z, (G, 1)\big) = \{(z', \mathrm{r}) \mid (z, z') \in G\},$$
$$\delta\big(z, (h, \mathrm{r})\big) = \text{ if } h(z) \text{ is defined then } \big\{\big(h(z), 1\big)\big\} \text{ else } \{(z, \mathrm{r})\},$$
$$\delta\big(z, (y, \mathrm{r})\big) = \text{ if } (z = y) \text{ then } \{(1, \mathrm{r})\} \text{ else } \emptyset.$$

Recall that the behavior of N_0 on inputs that are not nice is irrelevant.

We will also be interested in the special case of inputs of the form (1) where, like h, the relation G is also a partial function (Figure 3c). It is easy to verify that on such inputs N_0 does not use its nondeterminism, so we refer to strings of this form as *deterministic nice inputs* and we use g in place of G to represent them:

(2) $$(x, 1)(g, 1)(h, \mathrm{r})(y, \mathrm{r}).$$

Not surprisingly, the promise problem $\Psi = (\Psi_\text{yes}, \Psi_\text{no})$ with

$$\Psi_\text{yes} := \{w \in \Gamma^* \mid w \text{ is a deterministic nice input that has a path}\},$$
$$\Psi_\text{no} := \{w \in \Gamma^* \mid w \text{ is a deterministic nice input that has no path}\},$$

can be solved by a ZDFA M_0 with state set $[n]$, executing the following straightforward modification of the previous algorithm:

On a deterministic nice input like (2) surrounded by end-markers, we use the first 2 steps to reach $(g, 1)$ at state x. We then repeatedly and alternately read the two middle symbols, each time following the arrow (if any) defined by g or h. If we ever reach (h, r) at a state z from which no h-arrow departs, we stay at z and move right to check whether $z = y$. If so, we move 2 more steps to the right and accept.

Formally, $M_0 := (1, \delta, 1)$, where δ is any partial function from $[n] \times \Gamma_\mathrm{e}$ to $[n] \times \{1, \mathrm{r}\}$ that satisfies the following equations:

$$\delta(1, \vdash) = (1, \mathrm{r}), \qquad \delta\big(1, (x, 1)\big) = (x, \mathrm{r}), \qquad \delta(1, \dashv) = (1, \mathrm{r}),$$
$$\delta\big(z, (g, 1)\big) = \text{ if } g(z) \text{ is defined then } \big(g(z), \mathrm{r}\big) \text{ else `undefined'},$$
$$\delta\big(z, (h, \mathrm{r})\big) = \text{ if } h(z) \text{ is defined then } \big(h(z), 1\big) \text{ else } (z, \mathrm{r}),$$
$$\delta\big(z, (y, \mathrm{r})\big) = \text{ if } (z = y) \text{ then } (1, \mathrm{r}) \text{ else `undefined'}.$$

Again, the behavior of M_0 on inputs that are not deterministic nice is irrelevant.

The lower bounds that we will be proving in the following sections will be based on variants of problems Φ and Ψ. Perhaps the reader has already recognized in them two restrictions of C_n, the 2N-complete language of [**48**]. At the same time, Φ is a large-alphabet variant of a problem used in [**3**].

3. From 2DFAs to 1DFAs

Fix an n-state 2DFA $M = (s, \delta, f)$ over some set of states Q and an alphabet Σ. In this section we will build a 1DFA that is equivalent to M. First some facts.

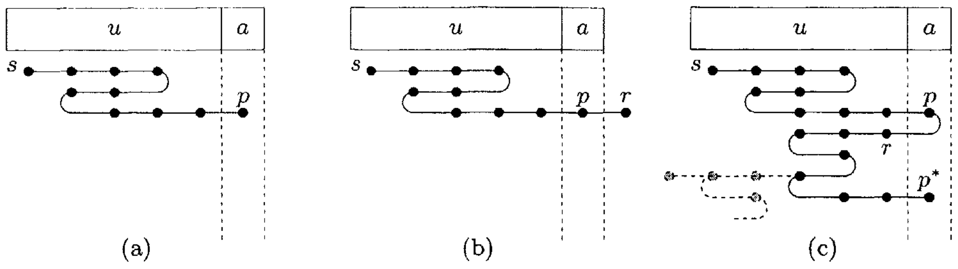


FIGURE 4. Trying to compute $\tau'(\perp)$: (a) c hangs right away, (b) c hits right in the first step, and (c) c moves left in the first step.

3.1. Tables. Consider some *non-empty* string u and suppose that the table of M on it $\tau := \text{TABLE}_M(u)$ is defined. We then know that the computation $c := \text{LCOMP}_{M,s}(u)$ hits right into $\tau(\perp)$. This implies that c visits the rightmost symbol of u at least once. If q is the state of M during the latest such visit, we easily see that the computation $c' := \text{RCOMP}_{M,q}(u)$ is a suffix of c. Hence, it also hits right, like c. Moreover, it certainly hits right into the same state as c, meaning $\tau(q) = \tau(\perp)$. We thus conclude that τ assigns to $\perp$ one of the values that it uses for the states in Q. That is, $\tau(\perp) \in \tau[Q]$. This motivates the following.

5. DEFINITION. A *table of* M is any $\tau : Q_\perp \to Q$ such that $\tau(\perp) \in \tau[Q]$.

Note that this defines what a "table of M" is, whereas Section 2.1-II defined what the "table of M on u" is, for any string u. The next lemma shows the relation between these two notions. The lemma after it, carries out an easy counting argument.

6. LEMMA. *If the* table of M on a string u *is defined, then it is a* table of M.

PROOF. If $u \neq \epsilon$, the proof is the argument before Definition 5. If $u = \epsilon$, the table of M on u is the constant function s (cf. Remark 3), and therefore it obviously qualifies as a table of M. □

7. LEMMA. *The number of distinct tables of M is exactly* $n(n^n - (n-1)^n)$.

PROOF. Easily, the number of distinct tables of M is exactly equal to the number of $(n+1)$-tuples of elements of $[n]$ where the first component equals some other component. Since there is a total of n^{n+1} unrestricted tuples and exactly $n(n-1)^n$ of them violate the restriction about the first component, the number that we want is the difference $n^{n+1} - n(n-1)^n$, exactly as claimed. □

3.2. Compatibilities among tables. Consider any string u, any symbol a, and suppose that the table $\tau := \text{TABLE}_M(u)$ is defined. We would like to know whether the table $\tau' := \text{TABLE}_M(ua)$ is also defined and, if so, to compute it. In this section we will show how this can be done using only τ and a, but not u. Our algorithm will be based on the algorithm implied in [**53**], but it will also need some modifications to account for the ambiguity of our representation.

Recall that τ' is defined iff the computation $\text{LCOMP}_{M,s}(ua)$ hits right. (Figure 4.) Clearly, this computation ends in $c := \text{RCOMP}_{M,\tau(\perp)}(ua)$. So, in order to figure out whether τ' is defined, we can just check whether c hits right. If it does, our check will also reveal the last state of c, which we know is the value of τ' on $\perp$.

```
$p \longleftarrow \tau(\perp)$
repeat:
  if $\delta(p,a)$ undefined: fail
  $(r,d) \longleftarrow \delta(p,a)$
  if $d = \mathtt{r}$: return $r$
  if $\tau(r) = \tau(\perp)$: fail
  if $\tau(r)$ seen before: fail
  $p \longleftarrow \tau(r)$
```

```
$p \longleftarrow q$
repeat:
  if $\delta(p,a)$ undefined: return $\tau'(\perp)$
  $(r,d) \longleftarrow \delta(p,a)$
  if $d = \mathtt{r}$: return $r$
  if $\tau(r) = \tau(\perp)$: return $\tau'(\perp)$
  if $\tau(r)$ seen before: return $\tau'(\perp)$
  $p \longleftarrow \tau(r)$
```

FIGURE 5. Computing $e_{\tau,a}(\perp)$ (left) and $e_{\tau,a}(q)$ (right).

So, let us set $p := \tau(\perp)$ and consider the first step of c. If $\delta(p,a)$ is undefined (Figure 4a), then c hangs inside ua and τ' is undefined. If $\delta(p,a)$ is defined and equal to some $(r,\mathtt{r})$ (Figure 4b), then c immediately hits right into r, so τ' is defined and $\tau'(\perp) = r$. The last case (Figure 4c) is that $\delta(p,a)$ equals some $(r,\mathtt{l})$, so then c starts behaving as $d := \text{RCOMP}_{M,r}(u)$ and we know this behavior is already encoded in τ as the value $p^* := \tau(r)$. We distinguish two cases.

- If $p^* = \tau(\perp)$, then we can conclude that τ' is undefined, as we are in one of the following two cases: *either* d hits left, loops, or hangs inside u, therefore c does the same inside ua, and hence τ' is not defined; *or* d hits right into $\tau(\perp)$, therefore c is back on a and at state p again, so c loops inside ua and hence τ' is again undefined.
- If $p^* \neq \tau(\perp)$, we know d hits right into p^*, so that c is back on a. To find out what happens next, we simply ask δ exactly as before, and continue. However, we should be careful not to ask δ a question we have already asked. If this is about to happen, then c repeats p^* under a, so c actually loops inside ua, in which case τ' is undefined.

This concludes our description of how to check whether τ' is defined and, if so, also compute $\tau'(\perp)$. Equivalently, we have shown that the algorithm of Figure 5(left) always terminates and *either* fails, if τ' is undefined, *or* correctly returns the value $\tau'(\perp)$, if τ' is defined.

In the case that τ' is defined, we also want to compute the rest of its values, namely $\tau'(q)$ for all $q \in Q$. Given the discussion so far, this is easy: we simply run the algorithm of Figure 5(left) again, but starting with $p := q$ (as opposed to $p := \tau(\perp)$). It is easy to verify that, if the algorithm does not fail, then $\text{RCOMP}_{M,q}(ua)$ hits right exactly into the state that the algorithm returns. So, if the algorithm does return a value, this value is the correct $\tau'(q)$. On the other hand, if the algorithm fails, this is due to one of its `fail` statements. We distinguish cases.

- If this is due to the 1st or the 3rd `fail` statement: Then we know that $\text{RCOMP}_{M,q}(ua)$ hangs on a or loops inside ua. Therefore, by definition, $\tau'(q)$ equals $\tau'(\perp)$. So, instead of failing, the algorithm should have returned $\tau'(\perp)$.
- If this is due to the 2nd `fail` statement: Then we know that one of the following is true about the computation $\text{RCOMP}_{M,q}(ua)$:
  - at some point, the computation locks itself inside u and eventually hits left, hangs, or loops: Then we again know that, by definition, $\tau'(q) = \tau'(\perp)$. So, instead of failing, the algorithm should have returned $\tau'(\perp)$.
  - at some point, the computation really enters state $\tau(\perp)$ while on a: Then we know that, from that point on, the computation will behave

identically to $\text{RCOMP}_{M,\tau(\perp)}(ua)$ and hence it will eventually hit right into $\tau'(\perp)$. So, once again, the algorithm should have returned $\tau'(\perp)$. In conclusion, we see that in all cases of failure the algorithm should have returned $\tau'(\perp)$. Hence, if in Figure 5(left) we just replace every `fail` statement with the statement "`return` $\tau'(\perp)$", we have a correct algorithm for computing $\tau'(q)$—provided, of course, that $\tau'(\perp)$ is defined and available. Figure 5(right) shows the algorithm after these modifications.

Overall, we have described an algorithm $e_{\tau,a}$ that can be used for checking if τ' is defined and, if so, for computing its values. The algorithm can be run on any element of $Q_\perp$. Figure 5(left) shows the computation $e_{\tau,a}(\perp)$. Figure 5(right) shows the computation $e_{\tau,a}(q)$, for $q \in Q$, where every reference to $\tau'(\perp)$ can be understood as a call to $e_{\tau,a}(\perp)$. With $e_{\tau,a}$ in our vocabulary, we can summarize the discussion of this section into the next definition and lemma.

8. Definition. If τ and τ' are two tables of M and a some symbol in Σ_e, we say that τ *is a-compatible to* τ' if and only if

$$\tau'(\perp) = e_{\tau,a}(\perp) \qquad \text{and} \qquad \text{for all } q \in Q:\ \tau'(q) = e_{\tau,a}(q),$$

where $e_{\tau,a}$ is the algorithm from Figure 5.

9. Lemma. *Suppose $u \in \Sigma_e^*$ and the table $\tau := \text{TABLE}_M(u)$ is defined. Then, for any $a \in \Sigma_e$ and any table τ' of M, the following holds:*

$$\tau \textit{ is a-compatible to } \tau' \iff \text{TABLE}_M(ua) \textit{ is defined and equals } \tau'.$$

Proof. Suppose τ is a-compatible to τ'. Then $\tau'(\perp) = e_{\tau,a}(\perp)$. Hence, on input $\perp$, the algorithm $e_{\tau,a}$ does not fail. This implies that the table $\text{TABLE}_M(ua)$ is defined. Moreover, its values are exactly those returned by $e_{\tau,a}$. But the values of τ' are also the same as those returned by $e_{\tau,a}$ (because τ is a-compatible to it). Overall, $\text{TABLE}_M(ua) = \tau'$.

Conversely, assume that the table $\text{TABLE}_M(ua)$ is defined and equals τ'. The argument before Definition 8 proves that $e_{\tau,a}$ returns the same values as $\text{TABLE}_M(ua)$. Hence, $e_{\tau,a}$ returns the same values as τ'. So, τ is a-compatible to τ'. □

3.3. The upper bound.

We are now ready to build a 1DFA M' that simulates M with exactly $n(n^n - (n-1)^n)$ states. First, we characterize the acceptance of an input by M in terms of the tables of M and the compatibilities between them.

10. Definition. Suppose $w \in \Sigma^*$ is of length l and $\tau_0, \tau_1, \ldots, \tau_{l+2}$ is a sequence of tables of M. We say the sequence *fits* w iff:

1. τ_0 is the constant function s,
2. for all $i = 0, 1, \ldots, l+1$: τ_i is $w_{e,i+1}$-compatible to τ_{i+1},[9]
3. τ_{l+2} is the constant function f.

11. Theorem. *M accepts $w \in \Sigma^*$ iff some sequence of tables of M fits w.*

Proof. Fix w and consider the sequence

$$\text{(3)} \qquad \text{TABLE}_M(\epsilon),\ \text{TABLE}_M(\vdash),\ \text{TABLE}_M(\vdash w_1),\ \ldots,\ \text{TABLE}_M(\vdash w \dashv).$$

Clearly, there is no guarantee that all members in this sequence are defined. But *if they all are*, then the sequence trivially satisfies Conditions 1 and 3 of Definition 10 (cf. Remark 3) and also Condition 2 (by Lemma 9), so the sequence fits w.

[9]Recall that by $w_{e,i+1}$ we mean the $i+1$st symbol of the end-marked string $\vdash w \dashv$. This is either $\vdash$ (when $i = 0$), or w_i (when $i \neq 0, l+1$), or $\dashv$ (when $i = l+1$).

Now assume M accepts w. Then the computation $\text{LCOMP}_{M,s}(w_e)$ hits right into f. This immediately implies that each one of the tables in (3) is defined. So, their sequence fits w. Conversely, suppose some sequence of tables of M fits w. By Lemma 9 and an easy induction, this sequence must be identical to the one in (3). This implies that the table $\text{TABLE}_M(w_e)$ is defined and equal to the constant function f. Hence, the computation $\text{LCOMP}_{M,s}(w_e)$ hits right, into f. This means that M accepts w. □

Based on this lemma, the construction of M' is straightforward. To test if its input is accepted by M, the automaton checks if there is a sequence of tables of M that fits it. At every step, it 'remembers' the last table of the subsequence found so far. More carefully, the algorithm is as follows:

> We start with the constant table s in our memory. On reading a symbol a, we check if the table in our memory is a-compatible to any table of M. If so, there is exactly one such table, so we change our memory to it and move right. If not, there is no sequence that fits w and we hang. We accept if we ever reach the constant table f.

Formally, $M' := (s', \delta', f')$ where $Q' := \{\tau \mid \tau \text{ is a table of } M\}$, $s' :=$ the table that always returns s, and $f' :=$ the table that always returns f. For each τ and a, the value $\delta'(\tau, a)$ is *either* the unique τ' to which τ is a-compatible, if such a τ' exists; *or* undefined, if τ is a-compatible to none of the tables of M. It should be clear that M' is correct and as large as claimed.

3.4. The lower bound. We will now prove that the construction of the previous section is optimal. In other words, we will prove that some n-state 2DFAs have no equivalent 1DFAs with fewer than $n(n^n - (n-1)^n)$ states. To this end, we will actually exhibit such a 2DFA. Our witness will be the automaton M_0 from Section 2.3, solving problem Ψ.

So, for the remainder of this section, we assume that the n-state 2DFA M that we kept fixed from the beginning of Section 3 is actually M_0. We will prove that the automaton M' constructed in the previous section is smallest among the 1DFAs that are equivalent to M. In fact, we will show that M' needs all its states not only for staying equivalent to M, but even for solving Ψ—on a stricter promise, even.

More precisely, consider any table $\tau : [n]_\perp \to [n]$ of M and any 'query' $i \in [n]$. The pair (τ, i) gives rise to the deterministic nice input

$$w_\tau^i := (x_\tau, \text{l})(g_\tau, \text{l})(h_\tau^i, \text{r})(y_\tau^i, \text{r}),$$

where x_τ is the smallest x for which $\tau(x) = \tau(\perp)$; g_τ is the restriction of τ to $[n]$; h_τ^i is the single arrow from $\tau(\perp)$ to i, if $\tau(i) \neq \tau(\perp)$, or else the empty function; and y_τ^i is just $\tau(i)$ (see Figure 6 for examples):

$$\text{(4)} \qquad \begin{array}{ll} x_\tau := \min\{x \mid \tau(x) = \tau(\perp)\}, & y_\tau^i := \tau(i) \\ g_\tau := \big\{(x, \tau(x)) \mid x \in [n]\big\} & h_\tau^i := \begin{cases} \emptyset & \text{if } \tau(i) = \tau(\perp), \\ \big\{(\tau(\perp), i)\big\} & \text{otherwise.} \end{cases} \end{array}$$

It is easy to verify the following.

12. LEMMA. *For any table τ and any state i of M: The table of M on the prefix $\vdash(x_\tau, \text{l})(g_\tau, \text{l})$ of w_τ^i is exactly τ and the computation c of M on w_τ^i is accepting.*

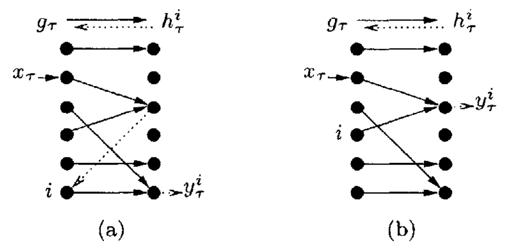


FIGURE 6. The input w^i_τ when $n = 6$, table τ maps $\bot, 1, 2, 3, 4, 5, 6$ to the values $3, 1, 3, 6, 3, 5, 6$ respectively, and (a) $i = 6$, or (b) $i = 4$.

Moreover, if $\tau(i) \neq \tau(\bot)$, then c contains a left-to-right crossing of the middle boundary from i to $\tau(i)$.

Hence, the inputs in the set $\{w^i_\tau \mid \tau \text{ is a table and } i \text{ a state of } M\}$ push M' to its limits, in the sense that they collectively force *every* one of its states to be used in *every* interesting way in *some* accepting computation. Hence, no state of M' is redundant in some straightforward manner, which intuitively suggests M' is minimal. It is this intuition that we turn into a proof.

We start with the observation that, for every two distinct tables of M, there is a 'query' that can distinguish them.

13. LEMMA. *Two tables τ and τ' of M are distinct iff there exist a partial function $h : [n] \to [n]$ and a $y \in [n]$ such that exactly one of the following inputs has a path:*

$$(x_\tau, \mathrm{l})(g_\tau, \mathrm{l})(h, \mathrm{r})(y, \mathrm{r}) \quad \textit{and} \quad (x_{\tau'}, \mathrm{l})(g_{\tau'}, \mathrm{l})(h, \mathrm{r})(y, \mathrm{r}).$$

PROOF. If the two tables are identical, then the two inputs are also identical and either none or both of them have a path. For the interesting direction, suppose τ, τ' are distinct. We examine two cases.

If $\tau(\bot) \neq \tau'(\bot)$, then choose h to be the empty function and y the smallest between $\tau(\bot)$ and $\tau'(\bot)$. Then the input corresponding to the table from which y takes its value has a path but the other input does not.

If $\tau(\bot) = \tau'(\bot) =: y^*$, then there exists an $x \in [n]$ such that $\tau(x) \neq \tau'(x)$ (or else τ, τ' would be identical, a contradiction). Pick the smallest such x, choose h to contain only the arrow from y^* to x, and set y to the smallest of $\tau(x)$, $\tau'(x)$ that is different from y^*. Clearly, there is a path in the input corresponding to the table from which y takes its value, while the other input has no path. □

Now, for every pair of tables τ and τ' of M, let us define the input

$$w_{\tau,\tau'} := (x_\tau, \mathrm{l})(g_\tau, \mathrm{l})(h_{\tau,\tau'}, \mathrm{r})(y_{\tau,\tau'}, \mathrm{r}),$$

where x_τ and g_τ are defined as in (4), while $h_{\tau,\tau'}$ and $y_{\tau,\tau'}$ are *either* the values given by the proof of Lemma 13, if $\tau \neq \tau'$; *or* the values h^i_τ, y^i_τ as defined in (4) for $i = x_\tau$, if $\tau = \tau'$.[10] Strengthening the promise of Ψ to allow only deterministic

[10]In the first case, note that the order of the pair τ, τ' is not important: $h_{\tau,\tau'} = h_{\tau',\tau}$ and $y_{\tau,\tau'} = y_{\tau',\tau}$. In the second case, note that we just have $h_{\tau,\tau} = \emptyset$ and $y_{\tau,\tau} = \tau(\bot)$.

nice inputs of this form, we get the problem Ψ' with

$$\Psi'_{\text{yes}} := \{w_{\tau,\tau'} \mid \tau,\tau' \text{ are tables of } M \text{ and } w_{\tau,\tau'} \text{ has a path}\},$$
$$\Psi'_{\text{no}} := \{w_{\tau,\tau'} \mid \tau,\tau' \text{ are tables of } M \text{ and } w_{\tau,\tau'} \text{ has no path}\}.$$

Clearly, M solves Ψ', so that a single-pass 2DFA can solve this problem with only n states. However, for 1DFAs the problem is maximally hard.

14. LEMMA. *Every* 1DFA *that solves Ψ' has at least $n(n^n - (n-1)^n)$ states.*

PROOF. Towards a contradiction, suppose that A is a 1DFA that solves Ψ' with fewer than $n(n^n - (n-1)^n)$ states. For every table τ of M, the automaton accepts $w_{\tau,\tau}$ (Lemma 12), so the computation $c_\tau := \text{COMP}_A(w_{\tau,\tau})$ hits right. In particular, it crosses the middle boundary from left to right. Let q_τ be the state that results from this crossing. Since there are fewer states in A than tables of M, two tables $\tau \neq \tau'$ must map to the same state $q := q_\tau = q_{\tau'}$. As a consequence, the computations of A on $w_{\tau,\tau'}$ and $w_{\tau',\tau}$ both cross the middle boundary into q (since the two strings are identical to $w_{\tau,\tau}$ and $w_{\tau',\tau'}$ before this boundary, respectively) and therefore have the same suffix (since the two strings are identical to each other after that boundary). In particular, they are either both accepting or both rejecting, a contradiction to Lemma 13 and the definition of $w_{\tau,\tau'}$ and $w_{\tau',\tau}$. ☐

4. From 2NFAs to 1DFAs

Fix an n-state 2NFA $N = (s, \delta, f)$ over some set of states Q and alphabet Σ. We will generalize the discussion of Section 3, to build a 1DFA equivalent to N.

4.1. Tables. Consider any *non-empty* string u and assume that the table $T := \text{TABLE}_N(u)$ is defined. This means that the set of computations $C := \text{LCOMP}_{N,s}(u)$ hits right into the set of states $T(\bot) \neq \emptyset$. Hence, C contains right-hitting computations. Let c be one of them. Clearly, c visits the rightmost symbol of u at least once. If p is the state of N at one of these visits, then combining the prefix of c up to that visit with any of the (possibly 0) right-hitting computations in $\text{RCOMP}_{N,p}(u)$, produces a right-hitting computation which is also in C. Therefore, the computations of $\text{RCOMP}_{N,p}(u)$ can hit right only into states that are already in $T(\bot)$. By the definition of T, this implies that $T(p) = T(\bot)$. We thus conclude that T *assigns the value $T(\bot)$ to at least one state.* Furthermore, a straightforward inspection of the definition of T reveals that *every state that is not assigned the set $T(\bot)$ is assigned a set disjoint from $T(\bot)$.* This motivates the following definition.

15. DEFINITION. A *table of N* is any $T : Q_\bot \to \mathcal{P}'(Q)$ such that
1. for every $p \in Q$: $T(p) = T(\bot)$ or $T(p) \cap T(\bot) = \emptyset$,
2. for some $p \in Q$: $T(p) = T(\bot)$.

As in the deterministic case, note that this definition explains what a "table of N" is, whereas Section 2.2-I defines what the "table of N on u" is, for any string u. The relation between the two notions is shown in the next lemma. The lemma after it carries out some counting.

16. LEMMA. *If the* table of N on a string u *is defined, then it is a* table of N.

PROOF. If $u \neq \epsilon$, then the argument before Definition 15 proves the claim. If $u = \epsilon$, then the table of N on u is the constant function $\{s\}$ (cf. Remark 4), and thus it is obviously a table of N. ☐

```
$P \longleftarrow T(\bot)$,  $S' \longleftarrow \emptyset$
repeat:
  $R \longleftarrow \bigcup\{\delta(p,a) \mid p \in P\}$
  $S' \longleftarrow S' \cup \{r \mid (r,\mathrm{r}) \in R\}$
  $P^* \longleftarrow \bigcup\{T(r) \mid (r,\mathrm{l}) \in R\}$
  $P \longleftarrow \{p \in P^* \mid p$ not seen before$\}$
  if $P = \emptyset$ then:
    if $S' = \emptyset$: fail
    if $S' \neq \emptyset$: return $S'$
```

```
$P \longleftarrow \{q\}$,  $S' \longleftarrow \emptyset$
repeat:
  $R \longleftarrow \bigcup\{\delta(p,a) \mid p \in P\}$
  $S' \longleftarrow S' \cup \{r \mid (r,\mathrm{r}) \in R\}$
  $P^* \longleftarrow \bigcup\{T(r) \mid (r,\mathrm{l}) \in R\} \setminus T(\bot)$
  $P \longleftarrow \{p \in P^* \mid p$ not seen before$\}$
  if $P = \emptyset$ then:
    if $S' \subseteq T'(\bot)$: return $T'(\bot)$
    if $S' \not\subseteq T'(\bot)$: return $S' \setminus T'(\bot)$
```

FIGURE 7. Computing $E_{T,a}(\bot)$ (left) and $E_{T,a}(q)$ (right).

17. LEMMA. *The number of distinct tables of N is exactly*[11]

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\binom{n}{i}\binom{n}{j}(2^i-1)^j.$$

PROOF. The number of distinct tables for N is equal to the number of distinct $(n+1)$-tuples of non-empty subsets of $[n]$ where the set of the first component appears in other components, too, but intersects no *other* set in the tuple. For each $i, j = 1, 2, \dots, n$, there are $\binom{n}{i}$ choices for the set S in the first component and $\binom{n}{j}$ choices for the set of the components after it that host the same set S. Given i and j, each one of the remaining $(n+1)-(j+1)$ components can admit any of the $2^{n-i}-1$ non-empty sets that avoid intersection with S. Overall, we have

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\binom{n}{i}\binom{n}{j}(2^{n-i}-1)^{n-j} = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\binom{n}{i}\binom{n}{j}(2^i-1)^j$$

choices for completing this $(n+1)$-tuple, where the right-hand-side of the equation is obtained via a straightforward variable substitution. □

4.2. Compatibilities among tables. Consider any string u, any symbol a, and suppose the table $T := \text{TABLE}_N(u)$ is defined. We will describe an algorithm for deciding whether the table $T' := \text{TABLE}_N(ua)$ is defined and, if so, for computing it based on T and a but not u. As in Section 3.2, our algorithm works on any element of $Q_\bot$. On input $\bot$, it either returns $T'(\bot)$ or fails, depending on whether T' is defined or not. On input $q \in Q$ and with $T'(\bot)$ available, it returns $T'(q)$.

We call this algorithm $E_{T,a}$ and we derive it from $e_{T,a}$ of Section 3.2 by a straightforward generalization that takes into account nondeterminism—see Figure 7 for the two distinct computations, $E_{T,a}(\bot)$ and $E_{T,a}(q)$. Based on $E_{T,a}$, we can again define a-compatibility between tables—the proof of the next lemma is similar to that of Lemma 9 and is omitted.

18. DEFINITION. If T and T' are two tables of N and a some symbol in Σ_e, we say that T *is a-compatible to* T' if and only if

$$T'(\bot) = E_{T,a}(\bot) \qquad \text{and} \qquad \text{for all } q \in Q\text{:}\ \ T'(q) = E_{T,a}(q),$$

where $E_{T,a}$ is the algorithm from Figure 7.

[11]Note that for $i = j = 0$, this expression uses the quantity 0^0. In this context, $0^0 = 1$.

19. LEMMA. *Suppose $u \in \Sigma_e^*$ and the table $T := \text{TABLE}_N(u)$ is defined. Then, for any $a \in \Sigma_e$ and any table T' of N, the following holds:*

$$T \text{ is } a\text{-compatible to } T' \iff \text{TABLE}_N(ua) \text{ is defined and equals } T'.$$

4.3. The upper bound. We now construct a 1DFA M that is equivalent to N. As in Section 3.3, we base our construction on a characterization of acceptance by N in terms of tables and compatibilities.

20. DEFINITION. Suppose $w \in \Sigma^*$ is of length l and $T_0, T_1, \dots, T_{l+2}$ is a sequence of tables of N. We say the sequence *fits* w iff:

1. T_0 is the constant function $\{s\}$,
2. for all $i = 0, 1, \dots, l+1$: T_i is $w_{e,i+1}$-compatible to T_{i+1},
3. T_{l+2} is the constant function $\{f\}$.

21. THEOREM. *N accepts $w \in \Sigma^*$ iff some sequence of tables of N fits w.*

The theorem is proved similarly to Theorem 11 and suggests that M should simply try to find a sequence of tables of N that fits its input. So, M implements the following algorithm:

> We start with the constant table $\{s\}$. On reading a symbol a, we check if the current table is a-compatible to any table of M. If so, there is exactly one such table, so we move right with this in our memory. If not, there is no sequence that fits w and we hang. We accept if we ever reach the constant table $\{f\}$.

Formally, $M := (s', \delta', f')$ where $Q' := \{T \mid T \text{ is a table of } N\}$, $s' :=$ the table that always returns $\{s\}$, and $f' :=$ the table that always returns $\{f\}$. For any T and a, the value $\delta'(T, a)$ is *either* the unique table to which T is a-compatible, if such table exists; *or* undefined, otherwise. It should be clear that M is correct and as large as claimed.

4.4. The lower bound. We will now exhibit an n-state 2NFA for which every equivalent 1DFA need at least one state per table. Our witness will be the automaton N_0 from Section 2.3, solving problem Φ. So, for the rest of this section, we assume that the n-state 2NFA N that we fixed at the beginning of Section 4 is the automaton N_0, and we will show that the 2DFA M constructed in the previous section is minimal. We start with some intuition why this must be the case.

For each table $T : [n]_\perp \to \mathcal{P}'([n])$ of N, each $i \in [n]$, and each $j \in T(i)$, we consider the nice input

$$w_T^{i,j} := (x_T, \text{l})(G_T, \text{l})(h_T^i, \text{r})(j, \text{r}),$$

where x_T is the smallest x for which $T(x) = T(\perp)$; G_T is the binary relation induced by T on $[n]$; and h_T^i contains either exactly the arrows from $T(\perp)$ to i, if $T(i) \neq T(\perp)$, or else no arrow at all (see Figure 8 for examples):

$$\begin{array}{ll} x_T := \min\{x \mid T(x) = T(\perp)\} & \\ G_T := \{(x,y) \mid y \in T(x)\} & \end{array} \qquad h_T^i := \begin{cases} \emptyset & \text{if } T(i) = T(\perp), \\ \{(y,i) \mid y \in T(\perp)\} & \text{otherwise.} \end{cases} \tag{5}$$

It is easy to verify the following fact.

22. LEMMA. *For any table T and any two states i, j of N such that $j \in T(i)$: The table of N on the prefix $\vdash(x_T, \text{l})(G_T, \text{l})$ of $w_T^{i,j}$ is exactly T and some computations in $\text{COMP}_N(w_T^{i,j})$ are accepting. Moreover, if $T(i) \neq T(\perp)$, then each accepting computation contains a step that crosses the middle boundary from i to j.*

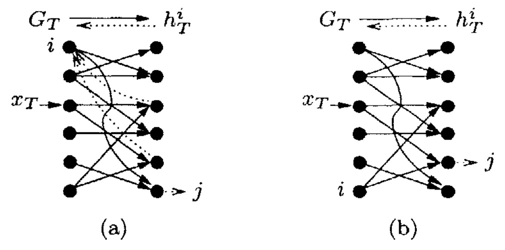


FIGURE 8. The input $w_T^{i,j}$ when $n = 6$; table T maps $\perp, 1, 2, 3, 4, 5$, and 6 to the values $\{3,5\}, \{2,6\}, \{1,2,4\}, \{3,5\}, \{4\}, \{6\}, \{3,5\}$ respectively; and (a) $i = 1$, $j = 6$, or (b) $i = 6$, $j = 5$.

Hence, as T, i, and j vary as above, the inputs $w_T^{i,j}$ collectively force *every* interesting use of *every* state of M in *some* accepting computation, so that intuitively no state of M is dispensable. To turn this intuition into a proof, we first establish the fact that distinct tables can be distinguished by 'query'.

23. LEMMA. *Two tables T and T' of N are distinct iff there exist a partial function $h : [n] \rightharpoonup [n]$ and a $y \in [n]$ such that exactly one of the following two inputs has a path:*

$$(x_T, \mathrm{l})(G_T, \mathrm{l})(h, \mathrm{r})(y, \mathrm{r}) \qquad \textit{and} \qquad (x_{T'}, \mathrm{l})(G_{T'}, \mathrm{l})(h, \mathrm{r})(y, \mathrm{r}).$$

PROOF. If $T = T'$ then, for all h and y, the two inputs are identical and therefore Φ does not distinguish between them. For the interesting direction, we suppose $T \neq T'$ and examine two cases.

If $T(\perp) \neq T'(\perp)$, then we can pick h to be the empty function and y the smallest state in the symmetric difference of the two $\perp$-values. Then the input that corresponds to the $\perp$-value that contains y is the only one with a path.

If $T(\perp) = T'(\perp) =: Y^*$, consider the smallest $x \in [n]$ with $T(x) \neq T'(x)$ (such an x exists, since $T \neq T'$). It is not hard to see that *either* both of the two x-values avoid intersection with Y^*, *or* exactly one of them does while the other one equals Y^*. (Indeed: If an x-value intersects Y^*, then it is actually equal to Y^*, since T and T' are tables. Hence, if both x-values intersected Y^*, we would have $T(x) = T'(x)$, a contradiction. So, at most one of them intersects Y^*. And, if one does, this one is equal to Y^*.) In both cases, the symmetric difference of the two x-values contains an element which does not belong to Y^*. If y is the smallest such element and h contains exactly the arrows from Y^* to x, namely

$$h := \{(y^*, x) \mid y^* \in Y^*\},$$

then the input that corresponds to the x-value containing y clearly has a path, while the other one does not. ☐

Now, for every pair of tables T and T' of N we define the nice input

$$w_{T,T'} := (x_T, \mathrm{l})(G_T, \mathrm{l})(h_{T,T'}, \mathrm{r})(y_{T,T'}, \mathrm{r})$$

where x_T, G_T are as in (5), while $h_{T,T'}$ and $y_{T,T'}$ are *either* the ones given by the proof of Lemma 23, if $T \neq T'$; *or* the values h^i_T and $\min T(i)$ as defined in (5) for

$i = x_T$, if $T = T'$.[12] Strengthening the promise for Φ to allow only inputs of this particular form, we get a new problem Φ' with

$$\Phi'_{\text{yes}} := \{w_{T,T'} \mid T, T' \text{ are tables for } N \text{ and } w_{T,T'} \text{ has a path}\},$$
$$\Phi'_{\text{no}} := \{w_{T,T'} \mid T, T' \text{ are tables for } N \text{ and } w_{T,T'} \text{ has no path}\}.$$

This is clearly still solvable by N, so that n states are enough on a single-pass 2NFA against Φ'. However, 1DFAs need many more states.

24. LEMMA. *The size of every* 1DFA *solving* Φ' *is at least*

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\binom{n}{i}\binom{n}{j}(2^i-1)^j.$$

PROOF. Assume A is a 1DFA solving Φ'. For every table T of N, the automaton accepts $w_{T,T}$ (Lemma 22). Hence, the computation $c_T := \text{COMP}_A(w_{T,T})$ hits right, and therefore crosses the middle boundary into some state, call it q_T. If the states of A were fewer than the tables of N, two tables $T \neq T'$ would map to the same state $q_T = q_{T'}$ and (by the standard cut-and-paste argument, as in Lemma 14) the automaton would be deciding identically on $w_{T,T'}$ and $w_{T',T}$, contradicting Lemma 23 and the definition of the two strings. Hence, A must have as many states as there are tables of N. The rest of the proof is by Lemma 17. ☐

5. From 2NFAs to 1NFAs

Fix an n-state 2NFA $N = (s, \delta, f)$ over state set Q and alphabet Σ. In this section we will build an equivalent $\binom{2n}{n+1}$-state 1NFA via an optimal construction.

5.1. Frontiers. Let us momentarily assume that N is actually deterministic and that $c := \text{COMP}_N(w)$ is accepting, for some l-long input w. Consider the i-th frontier (L_i^c, R_i^c) of c, for some $i \neq 0, l+2$. The number of states in R_i^c equals the number of times that c left-to-right crosses the i-th boundary: each crossing contributes a state into R_i^c and no two crossings contribute the same state, or else c would be looping. Similarly, L_i^c contains as many states as many times c right-to-left crosses the i-th boundary. Now, since c accepts, it goes from the leftmost symbol $\vdash$ all the way past the rightmost one $\dashv$, forcing the left-to-right crossings on every boundary to be exactly 1 more than the right-to-left crossings. Hence,

$$|L_i^c| + 1 = |R_i^c|,$$

which remains true even on the leftmost boundary ($i = 0$, under our convention from Footnote 6 on page 28) and also on the rightmost one ($i = l + 2$, obviously). So, the equality holds over every boundary, and motivates the following definition.

25. DEFINITION. A *frontier of* N is any (L, R) with $L, R \subseteq Q$ and $|L|+1 = |R|$.

Note that this defines what a "frontier of N" is, whereas Section 2.1-III defined what a "frontier of a computation" is. The relation between the two notions is partially explained by the motivating argument above, which shows that *if the computation* c *of a* 2DFA *on an input is accepting, then all frontiers of* c *are frontiers of the* 2DFA.

However, this argument is not valid for our nondeterministic N, as a state repetition under a cell does not necessarily imply looping. Still, it implies a cycle.

[12]Again, when $T \neq T'$, the order of the two tables in the definition of these values is not important: $h_{T,T'} = h_{T',T}$ and $y_{T,T'} = y_{T',T}$. Also, $h_{T,T} = \emptyset$.

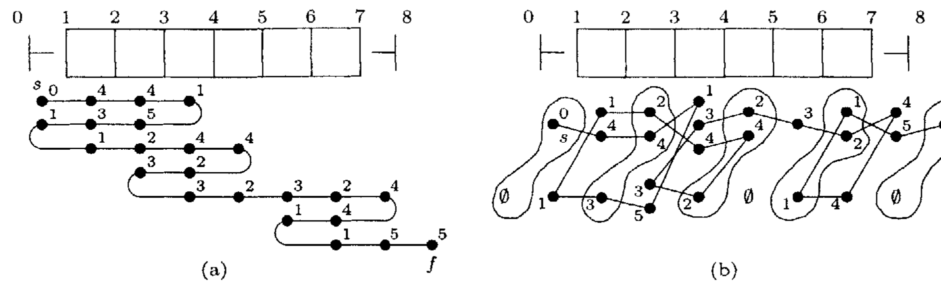


FIGURE 9. (a) An accepting minimal $c \in \text{COMP}_N(w)$, for a 6-long w; we assume $0, 1, \ldots, 5$ are states of N, and $s = 0$, $f = 5$. (b) The same c arranged in frontiers; only the even-indexed frontiers are drawn.

26. DEFINITION. A computation is *minimal* if every two of its points are distinct.

In other words, a computation is minimal iff it is cycle-free. Obviously, a minimal computation is not looping, and for 2DFAs the converse is also true. However, for 2NFAs the converse is not always true: accepting computations may not be minimal. So, in order to extend our previous observation to 2NFAs, we need to take this detail into account. The following lemma makes the appropriate corrections. The lemma after it carries out an easy counting.

27. LEMMA. *If a computation of N on a string w is* accepting *and* minimal, *then all frontiers of that computation are frontiers of N.*

PROOF. We just need to modify the argument before Definition 25, as follows: No two left-to-right crossings of the i-th boundary contribute the same state into R_i^c, or else c would not be *minimal.* Similarly for L_i^c. ☐

28. LEMMA. *The number of distinct frontiers of N is exactly $\binom{2n}{n+1}$.*

PROOF. Easily, the number of different frontiers of N is equal to the number of distinct pairs of subsets of $[n]$ where the second subset is 1-larger than the first one. In turn, this is equal to the number of different ways of choosing $n + 1$ elements of the set $[2n]$: the *unselected* elements of $[n]$ determine the first subset, while the *selected* elements of $[2n] - [n]$ determine the second subset. Therefore, our number is exactly $\binom{2n}{n+1}$, as claimed.[13] ☐

5.2. Compatibilities among frontiers. Suppose c is an accepting minimal computation of N on an l-long w and let $F_i^c = (L_i^c, R_i^c)$ be its *i-th frontier*, for each $i = 0, 1, \ldots, l + 2$ (Figure 9). Note that the first and last frontiers in the sequence $F_0^c, F_1^c, \ldots, F_{l+2}^c$ are always

$$F_0^c = (\emptyset, \{s\}) \qquad \text{and} \qquad F_{l+2}^c = (\emptyset, \{f\}),$$

as c starts at s, ends in f, and never right-to-left crosses an outer boundary.

Also note that, for $(L, R) := (L_i^c, R_i^c)$ and $(L', R') := (L_{i+1}^c, R_{i+1}^c)$ two successive frontiers in the sequence (Figure 10a), it should always be that $R \cap L' = \emptyset$:

[13]Another way to write this number is nC_n, where C_n is the n-th Catalan number. So, Catalan strikes again! [12, 10]

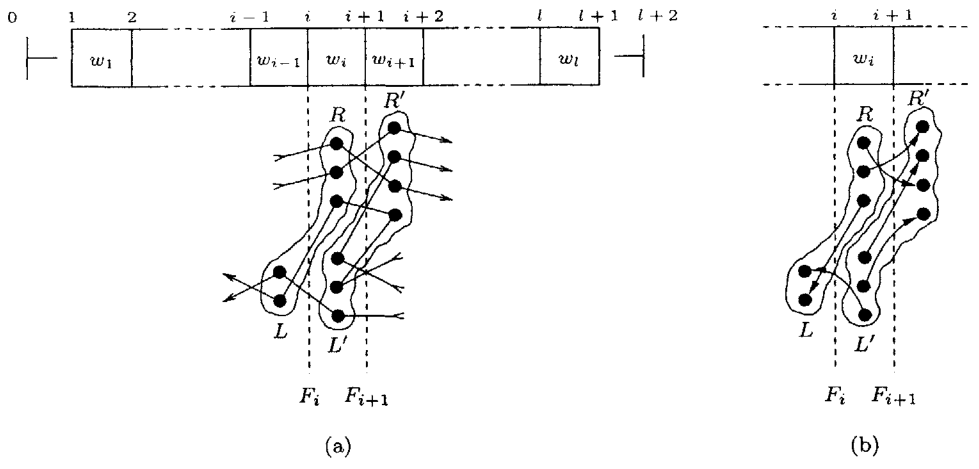


FIGURE 10. (a) Two successive frontiers. (b) The associated bijection.

otherwise, c would be using the same state under the $(i+1)$st cell of the tape and would not be minimal. Hence, $R+L'$ contains as many states as many (occurrences of) states there are in L and R' together:

$$|R+L'| = |R| + |L'| = (|L|+1) + (|R'|-1) = |L| + |R'| = |L \uplus R'|.$$

Hence, bijections can be found from $R+L'$ to $L \uplus R'$. Among them, a very natural one (Figure 10b): for each $q \in R+L'$ find the unique step in c that produces q under the $(i+1)$st cell (this is either a left-to-right crossing of boundary i or a right-to-left crossing of boundary $i+1$; the minimality of c guarantees uniqueness); the next step left-to-right crosses boundary $i+1$ into some state $p \in R'$ or right-to-left crosses boundary i into some $p \in L$; depending on the case, map q to (p, r) or (p, l) respectively. If $\rho : R+L' \to L \uplus R'$ is this mapping, it is easy to verify that it is injective (because c is minimal) and therefore bijective, as promised. In addition, ρ clearly respects the transition function: $\rho(q) \in \delta(q, w_{\mathrm{e},i+1})$, for all $q \in R+L'$.

Overall, this discussion shows that every accepting minimal computation in $\text{COMP}_N(w)$ exhibits a sequence of frontiers which necessarily obeys certain restrictions. The following definitions and lemma summarize these findings.

29. DEFINITION. If (L, R) and (L', R') are two frontiers of N and a some symbol in Σ_e, we say that that (L, R) *is a-compatible to* (L', R') if and only if

1. $R \cap L' = \emptyset$, and
2. some bijection $\rho : R+L' \to L \uplus R'$ respects the transition function on a:

$$\text{for all } q \in R+L'\text{: } \rho(q) \in \delta(q, a).$$

30. DEFINITION. Suppose $w \in \Sigma^*$ is of length l and $F_0, F_1, \ldots, F_{l+2}$ is a sequence of frontiers of N. We say the sequence *fits* w iff

1. $F_0 = (\emptyset, \{s\})$,
2. for all $i = 0, 1, \ldots, l+1$: F_i is $w_{\mathrm{e},i+1}$-compatible to F_{i+1},
3. $F_{l+2} = (\emptyset, \{f\})$.

31. LEMMA. *For all $w \in \Sigma^*$: if* $\text{COMP}_N(w)$ *contains an accepting computation, then some sequence of frontiers of N fits w.*

PROOF. Suppose $\text{COMP}_N(w)$ contains an accepting computation d. Removing from d all cycles, we get a computation c which is also in $\text{COMP}_N(w)$ and is *accepting and minimal.* Then, the argument before Definition 29 proves that the sequence of the frontiers of c fits w. □

The crucial observation—proved in the next section—is that the converse of this lemma is also true, and therefore an analogue of Theorems 11 and 21 holds.

32. LEMMA. *For all $w \in \Sigma^*$: if some sequence of frontiers of N fits w, then $\text{COMP}_N(w)$ contains an accepting computation.*

33. THEOREM. *N accepts $w \in \Sigma^*$ iff some sequence of frontiers of N fits w.*

5.3. Proof of the main observation. In this section we prove Lemma 32. So, assume that the sequence of frontiers

$$F_0 = (L_0, R_0), \quad F_1 = (L_1, R_1), \quad \dots, \quad F_{l+2} = (L_{l+2}, R_{l+2})$$

fits w. We will prove a stronger statement: for every $i = 0, 1, \dots, l+2$, the states of R_i can be produced by $|R_i|$ right-hitting computations on $\vdash w_1 \cdots w_{i-1}$, one of them starting at s and on $\vdash$ and each one of the remaining $|L_i|$ starting at a distinct $q \in L_i$ and on w_{i-1}. More formally, we will prove the following claim.

CLAIM. *For all $i = 0, 1, \dots, l+2$, a bijection $\pi_i : (L_i)_\perp \to R_i$ is such that*
1. *some $c \in \text{LCOMP}_{N,s}(\vdash w_1 \cdots w_{i-1})$ hits right into $\pi_i(\perp)$, and*
2. *for all $q \in L_i$, some $c \in \text{RCOMP}_{N,q}(\vdash w_1 \cdots w_{i-1})$ hits right into $\pi_i(q)$.*

Note that Lemma 32 indeed follows from this claim, when $i = l+2$: The only bijection from $(L_{l+2})_\perp = \emptyset_\perp = \{\perp\}$ to $R_{l+2} = \{f\}$ is $\pi_{l+2} := \{(\perp, f)\}$. So, condition 1 says that some computation in $\text{LCOMP}_{N,s}(\vdash w_1 \cdots w_l \dashv)$ hits right into $\pi_{l+2}(\perp) = f$, which implies that $\text{COMP}_N(w)$ contains an accepting computation.

To prove the claim, we use induction on i.

BASE CASE. The base case $i = 0$ is satisfied by the definitions. The only bijection from $(L_0)_\perp = \emptyset_\perp = \{\perp\}$ to $R_0 = \{s\}$ is $\pi_0 = \{(\perp, s)\}$. Condition 1 is true because $\text{LCOMP}_{N,s}(\epsilon)$ contains exactly the 0-length computation $\big((s,1)\big)$ which does hit right into s (cf. Remark 1). Condition 2 is true vacuously, as $L_0 = \emptyset$.

INDUCTIVE STEP. For the inductive step (Figure 11), assume $i < l+2$, let $(L, R) := (L_i, R_i)$, $(L', R') := (L_{i+1}, R_{i+1})$, $a := w_{e,i+1}$, and consider the bijections

$$\pi := \pi_i : L_\perp \to R \qquad \text{and} \qquad \rho : R + L' \to L \uplus R'$$

guaranteed respectively by the inductive hypothesis and by the assumption that (L, R) is a-compatible to (L', R'). We need to build a bijection

$$\pi' := \pi_{i+1} : (L')_\perp \to R'$$

that satisfies Conditions 1, 2 of the claim, and we will do so based on π, ρ, and one more function σ that will emerge from the following discussion.

DEFINITION OF σ. Consider any state $q \in R$ and let us take a trip around under $\vdash w_1 w_2 \cdots w_{i-1} a$ by alternately following bijections ρ and π

$$(6) \qquad \underset{r_0}{q}, \quad \underset{r_1}{\rho(q)}, \quad \underset{r_2}{\pi\rho(q)}, \quad \underset{r_3}{\rho\pi\rho(q)}, \quad \underset{r_4}{\pi\rho\pi\rho(q)}, \quad \dots$$

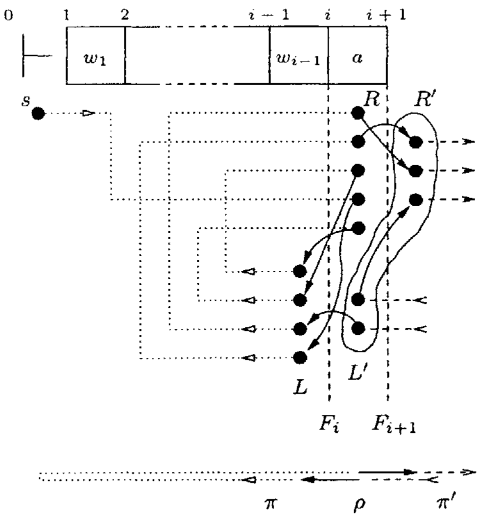


FIGURE 11. An example for the inductive step in the proof of Lemma 32. Note, for instance, that σ maps the 3rd and 5th (from the top) states of R to $\bot$, while the 4th state is mapped to the 1st state of R'.

until the first time that 'ρ fails to return a state in L'',[14] and let $r_0, r_1, r_2, \dots$ be the states that we visit. There are two cases about what might happen.

Case 1 is that ρ does eventually fail to return a state in L and the trip pays only a finite number of visits

$$r_0, r_1, \dots, r_k,$$

for some even $k \geq 0$ and with $r_k \in R$. Then r_k is ρ-mapped to some $q' \in R'$.

Case 2 is that ρ always returns a state in L and the trip is infinite. Since all even-indexed and all odd-indexed visits in the trip are inside the finite sets R and L respectively, there have to be repetitions of states both on the even and on the odd indices. Let k be the first index for which some earlier index $j < k$ of the *same parity* points to the same state: $r_j = r_k$. If k is odd, then j is also odd and hence $j \geq 1$; then $r_j = r_k \implies \rho^{-1}(r_j) = \rho^{-1}(r_k) \implies r_{j-1} = r_{k-1}$ and $k-1$ also has the property that k is the earliest one to have, a contradiction. So, k must be even, and so must j. In fact, j must be 0—otherwise we can again reach a contradiction (as before, with π^{-1} instead of ρ^{-1}). Hence, the first state to be revisited is the state q we started from and our trip consists of infinitely many copies of a cycle

$$r_0, r_1, \dots, r_k,$$

for some even $k \geq 2$, for $r_k = r_0 = q \in R$, and with no two states in the list $r_0, r_1, \dots, r_{k-1}$ being both equal and at positions of the same parity.

Overall, in the trip that we take, we either reach a state $r_k \in R$ (possibly $k = 0$) that is ρ-mapped to a state $q' \in R'$ (Case 1), or we return to the starting state

[14]Note that we abuse notation here. Bijection ρ can only return a *pair* of the form (p, l) or (p, r). So, in the description (6) above, $\rho(\cdot)$ really means 'the first component of $\rho(\cdot)$, if the second component is l'. Similarly, 'ρ fails to return a state in L' means 'ρ returns a pair of the form (p, r)'. Hopefully, the abuse does not confuse. We will stick to it throughout the definition of σ.

$q \in R$ having previously repeated no state in L and no state in R (Case 2). We define the function

$$\sigma : R \to (R')_{\perp}$$

to encode exactly this information. Specifically: in Case 1, we set $\sigma(q) := q'$; in Case 2, we set $\sigma(q) := \perp$. In either case, our trip respects π and ρ, which in turn respect the behavior of N on $\vdash w_1 w_2 \dots w_{i-1} a$. It should therefore be clear that

- $\sigma(q) = q'$ implies there is some $c \in \text{RCOMP}_{N,q}(\vdash w_1 \cdots w_{i-1} a)$ that respects π, ρ and *hits right into* q', while
- $\sigma(q) = \perp$ implies there is some *looping* $c \in \text{RCOMP}_{N,q}(\vdash w_1 \cdots w_{i-1} a)$ that respects π, ρ and repeats a cycle that visits only states from R when under a.

This concludes the definition of σ and our discussion of its properties. □

We are now ready to return to the construction of bijection π'. Recall that this must inject everything in $L'_{\perp}$ to R' so that Conditions 1 and 2 of the claim above are satisfied. We examine three separate cases about the argument of π'.

Case (a). The easiest argument is a state $p \in L'$ that is ρ-mapped rightward to a state $r \in R'$. Then we just let π' also return that state: $\pi'(p) := r$. Since ρ respects the transition function on a, the corresponding 1-step computation from p rightward to r is indeed in $\text{RCOMP}_{N,p}(\vdash w_1 \cdots w_{i-1} a)$ that hits right into $\pi'(p)$.

Case (b). If the argument is some $p \in L'$ that is ρ-mapped leftward to a state $r \in L$, then we consider where in R bijection π takes us from there: $q := \pi(r)$. We know some computation of N can start at p under a and eventually reach q under a, so the question is what can happen after that if we keep following ρ and π. To answer this question, we examine $\sigma(q)$.

If $\sigma(q) = \perp$, then we will eventually return to q after a cycle of length at least 2 and having visited only states of R when under a. But can this happen? If it does, then the next-to-last and last steps in this cycle will follow ρ and π respectively, ending up in q. Since ρ and π are bijections, the last two states (before q) in this cycle must respectively be p and r. In particular, p must be in the cycle. But, since the cycle visits only states from R whenever under a, we should have $p \in R$. This means that R and L' must intersect, and hence (L, R) is not a-compatible to (L', R'), a contradiction.

It is therefore necessary that $\sigma(q) = q' \in R'$, which implies that some computation $c \in \text{RCOMP}_{N,q}(\vdash w_1 \cdots w_{i-1} a)$ hits right into q'. Prefixed by the computation that takes p to q, this c becomes a computation in $\text{RCOMP}_{N,p}(\vdash w_1 \cdots w_{i-1} a)$ that hits right into q'. So, we can safely set $\pi'(p) := q'$.

Case (c). It remains to define $\pi'(\perp)$. The reasoning is similar to Case (b). We consider the state $q \in R$ where $\pi(\perp)$ takes us, and examine $\sigma(q)$. Again, $\sigma(q) = \perp$ is impossible, as this would imply that $\perp \in L$, a contradiction. Hence, $\sigma(q) = q'$ for some $q' \in R'$. Combining the computation guaranteed by $\pi(\perp) = q$ with the one guaranteed by $\sigma(q) = q'$, we get a computation in $\text{LCOMP}_{N,s}(\vdash w_1 \cdots w_{i-1} a)$ that hits right into q'. So, we can safely set $\pi'(\perp) := q'$.

This concludes the definition of π'. The construction must have made clear that π' satisfies the two conditions of the claim; that it is also a bijection should be an easy consequence of the way bijections π and ρ are used. Hence, the inductive step is complete, and with it the proof of the claim.

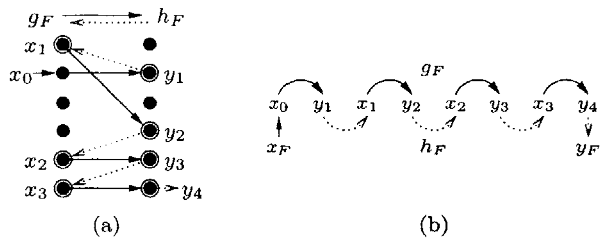


FIGURE 12. (a) The deterministic nice input w_F when $n = 6$ and $F = (\{1,5,6\},\{2,4,5,6\})$. (b) How to derive it from the corresponding list $2,2,1,4,5,5,6,6$.

5.4. The upper bound. We are now ready to build a $\binom{2n}{n+1}$-state 1NFA N' that simulates N. Our construction is based on Theorem 33. In other words, the strategy of N' is to go through the input 'guessing' the members of a sequence of frontiers, one after the other, and verifying that this sequence fits the input. More precisely, N' implements the following algorithm:

> We start with the frontier $(\emptyset,\{s\})$ in our memory. On reading a symbol a, we check if the frontier in our memory is a-compatible to any other frontiers. If not, we just hang. If it is, we find all such frontiers, select one of them nondeterministically, and move right with it as our new memory. If we ever reach the frontier $(\emptyset,\{f\})$, we accept.

Formally, $N' := (s',\delta',f')$ where $Q' := \{F \mid F \text{ is a frontier for } N\}$, $s' := (\emptyset,\{s\})$, $f' := (\emptyset,\{f\})$, and the transition function is such that $\delta'(F,a) := \{F' \mid F \text{ is } a\text{-compatible to } F'\}$, for all $F \in Q'$ and $a \in \Sigma_e$. It should be clear that N' is correct and as large as promised.

5.5. The lower bound. To prove that the construction of the previous section is optimal, we will exhibit an n-state 2NFA that has no equivalent 1NFA with fewer than $\binom{2n}{n+1}$ states. In fact, our witness will simply be the automaton M_0 from Section 2.3, which is *deterministic* and *single-pass*. Moreover, 1NFAs will be shown to need $\binom{2n}{n+1}$ states not only for staying equivalent to M_0, but even for solving Ψ—on a stricter promise, actually.

So, assume that the n-state 2NFA N that we kept fixed since the beginning of Section 5 is actually M_0. We will show that the 1NFA N' constructed in the previous section is minimal. We start with some intuition why this should be true.

Consider an arbitrary frontier $F = (L,R)$ of N and let us list the elements of the sets $L, R \subseteq [n]$ in increasing order,

$$L = \{x_1, x_2, \dots, x_m\} \qquad \text{and} \qquad R = \{y_1, y_2, \dots, y_{m+1}\},$$

for the appropriate $0 \le m < n$. Since $m < n$, we know L is a strict subset of $[n]$. So, we can name an element outside L, say $x_0 := \min \overline{L}$. Then the combined list

$$x_0\, y_1\, x_1\, y_2\, x_2\, \cdots\, y_m\, x_m\, y_{m+1} \tag{7}$$

gives rise to the following deterministic nice input (see Figure 12):

$$w_F := (x_F, \mathrm{l})(g_F, \mathrm{l})(h_F, \mathrm{r})(y_F, \mathrm{r}),$$

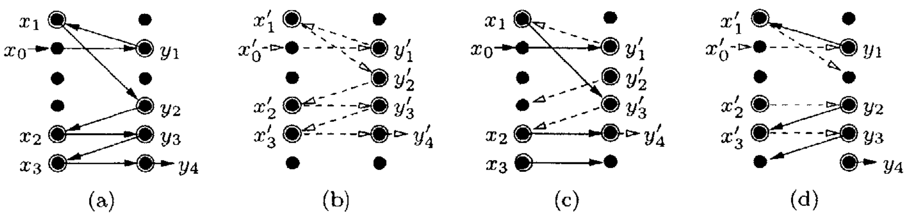


FIGURE 13. (a) Input w_F from Figure 12. (b) A new input $w_{F'}$, for $F' = (\{1,4,5\},\{2,3,4,5\})$. (c,d) Inputs $w_{F,F'}$ and $w_{F',F}$. Note that only $w_{F,F'}$ has a path.

where $x_F := x_0$, the function g_F maps every x in the list (7) to the following y, the function h_F maps every $y \neq y_{m+1}$ to the following x, and $y_F := y_{m+1}$. Namely:

$$(8) \qquad \begin{array}{ll} x_F := \min \overline{L} & y_F := \max R \\ g_F := \{(x_i, y_{i+1}) \mid 0 \le i \le m\} & h_F := \{(y_i, x_i) \mid 1 \le i \le m\}. \end{array}$$

It is easy to verify that this input has a path and that the following holds.

34. LEMMA. *For any frontier F of N, the computation of N on w_F is accepting and its frontier under the middle boundary is exactly F.*

This implies that *every* state of N' is used in *some* accepting computation and is therefore not redundant in any obvious way. Hence, N' indeed seems minimal.

To prove this intuition, we start by noting that every two frontiers F and F' of N give rise to the deterministic nice input (see Figure 13 for an example)

$$w_{F,F'} := (x_F, \mathtt{l})(g_F, \mathtt{l})(h_{F'}, \mathtt{r})(y_{F'}, \mathtt{r}),$$

where x_F, g_F, $h_{F'}$, $y_{F'}$ are defined as in (8). Strengthening the promise for problem Ψ to allow only inputs of this form, we get problem $\Psi'' = (\Psi''_{\text{yes}}, \Psi''_{\text{no}})$, with

$$\Psi''_{\text{yes}} := \{w_{F,F'} \mid F, F' \text{ are frontiers for } N \text{ and } w_{F,F'} \text{ has a path}\},$$
$$\Psi''_{\text{no}} := \{w_{F,F'} \mid F, F' \text{ are frontiers for } N \text{ and } w_{F,F'} \text{ has no path}\}.$$

Clearly, N solves this problem, so that n states on a (single-pass deterministic) 2NFA are enough to solve Ψ''. For 1NFAs the problem is harder.

35. LEMMA. *Every* 1NFA *that solves Ψ'' has at least $\binom{2n}{n+1}$ states.*

At the heart of the argument for this lemma lies the fact that, in the $\binom{2n}{n+1} \times \binom{2n}{n+1}$ matrix $W = [w_{F,F'}]_{F,F'}$ containing all inputs of the form $w_{F,F'}$, two distinct inputs sitting in cells that are symmetric with respect to the main diagonal cannot both have a path. We prove this claim before proving Lemma 35 itself.

CLAIM. *For any two frontiers F, F' of N: $w_{F,F'}, w_{F',F} \in \Psi''_{\text{yes}} \iff F = F'$.*

PROOF. Let $F = (L, R)$ and $F' = (L', R')$ be any two frontiers of N. If $F = F'$ then $w_{F,F'} = w_{F',F} = w_F$ and we have already observed that this input has a path. For the interesting direction, we assume that $F \neq F'$ and we will prove that at least one of $w_{F,F'}$, $w_{F',F}$ lacks a path.

We start by letting $m = |L|$, $m' = |L'|$ and considering the combined lists defined by the two frontiers, as in (7):

$$x_0\, y_1\, x_1\, y_2\, x_2 \cdots y_m\, x_m\, y_{m+1} \quad \text{and} \quad x'_0\, y'_1\, x'_1\, y'_2\, x'_2 \cdots y'_m\, x'_m\, y'_{m'+1}.$$

If the two lists were identical *after* their first elements, they would agree in their lengths, in their x's (except possibly at x_0, x_0'), and in their y's, forcing $F = F'$, a contradiction. Hence, there have to be positions of disagreement after 0. Consider the earliest one among them.

If this position is occupied by y's, say y_i and y_i', then we have *either* that $y_i < y_i'$ (Case 1) *or* that $y_i > y_i'$ (Case 2). If it is occupied by x's, say x_i and x_i', then we have *either* that $x_i < x_i'$ or x_i' is not present at all[15] (Case 3) *or* that $x_i > x_i'$ or x_i is not present at all (Case 4). The four cases are treated with similar arguments. We will present only the argument for the first one in detail, and sketch the rest.

So, suppose that the first disagreement is between y_i and y_i', and that in fact $y_i < y_i'$. This implies that all previous positions after 0 contain identical elements:

$$\begin{tikzcd}
x_F \arrow[r] & x_0 \arrow[r, bend left, "g_F"] & y_1 & x_1 \arrow[r, bend left] & y_2 & x_2 \arrow[r, bend left] & \cdots \arrow[r, bend left] & y_{i-1} & x_{i-1} \arrow[r, bend left] & y_i \\
 & x_0' & y_1' \arrow[r, bend right, "h_{F'}"'] & x_1' & y_2' \arrow[r, bend right] & x_2' & \cdots & y_{i-1}' \arrow[r, bend right] & x_{i-1}' & y_i'
\end{tikzcd}$$

It also implies that y_i is not in R'. Indeed, if it were, it would be in the sublist $y_1', y_2', \dots, y_{i-1}'$ (since $y_i < y_i'$), and hence in the sublist $y_1, y_2, \dots, y_{i-1}$ (since the two sublists coincide), contradicting the fact that y_i is greater than all these elements of R. So $y_i \notin R'$, and therefore y_i is not $y_{F'}$ (which is in R') and has no value under $h_{F'}$ (since the domain of $h_{F'}$ is also in R'). But then searching for a path in $w_{F,F'}$ we travel deterministically

$$x_0 \xrightarrow{g_F} (y_1 = y_1') \xrightarrow{h_{F'}} (x_1' = x_1) \xrightarrow{g_F} (y_2 = y_2') \xrightarrow{h_{F'}} \cdots \xrightarrow{h_{F'}} (x_{i-1}' = x_{i-1}) \xrightarrow{g_F} y_i$$

reaching a node which is neither the exit $y_{F'}$ nor the start of an $h_{F'}$-arrow. This means that $w_{F,F'}$ has no path.

In Case 2, we similarly conclude that $y_i' \notin R$ and that $g_{F'}$, h_F combine to reach y_i'; but this is neither the exit y_F nor the start of an h_F-arrow, implying $w_{F',F}$ has no path. In Case 3, we deduce that $x_i \notin L'$ and yet $g_{F'}$, h_F combine to reach it, so $w_{F',F}$ has no path. Finally, in Case 4, x_i' can be shown outside L while g_F and $h_{F'}$ reach it, so that $w_{F,F'}$ has no path. □

PROOF OF LEMMA 35. Towards a contradiction, assume that A is a 1NFA that solves Ψ'' with fewer than $\binom{2n}{n+1}$ states. For each frontier F for N, we know the input $w_F = w_{F,F}$ is in Ψ''_{yes} and therefore A accepts it. Choose any accepting computation $c_F \in \text{COMP}_A(w_F)$ and let q_F be the state immediately after the middle boundary is crossed. Since the states of A are fewer than the frontiers for N, we know $q_F = q_{F'}$ for two frontiers $F \neq F'$. But then, the usual cut-and-paste argument on the computations c_F and $c_{F'}$ shows that A must also accept the inputs $w_{F,F'}$ and $w_{F',F}$. Since A solves Ψ'', we conclude that $w_{F,F'}, w_{F',F} \in \Psi''_{\text{yes}}$ despite $F \neq F'$, a contradiction to the last claim. □

6. Conclusion

In this first chapter we showed the exact trade-offs in the conversions from two-way (deterministic or nondeterministic) to one-way (deterministic or nondeterministic) finite automata. Our arguments recast those of Birget [**6**] into a more

[15]This happens if the list for F' stops at y_i'.

standard set-theoretic vocabulary and then complement them by carefully removing the redundancies in the associated constructions.[16]

Introducing frontiers, we provided a set-theoretic *characterization of* 2NFA *acceptance* (already present in [**6**], essentially) that complements the also set-theoretic *characterization of* 2NFA *rejection* given in [**60**]. Moreover, by applying the concept of promise problems even to the domain of regular languages, we nicely confirmed its reputation for always leading us straight to the combinatorial core of the hardness of a computational task.

Crucially, the tight simulations performed by one-way automata in our proofs are as 'meaningful' as the tight simulation given in [**47**] for the removal of nondeterminism from 1NFAs: each state in these automata corresponds to a realizable and non-redundant set-theoretic object (a table, a frontier) that naturally emerges from the computational behavior of the simulated machine.

It would be nice to identify similar objects and derive exact trade-offs for the conversions from and towards other types of automata (e.g., alternating, probabilistic, or pebble automata, or even Hennie machines [**4**]) and more powerful machines (e.g., pushdown automata). In addition, it would be interesting to know if the large size of the alphabet over which problems Φ and Ψ are defined is necessary for the exactness of the associated trade-offs.

A preliminary version of the contents of Section 5 can be found in [**29**].

[16]First, the reasoning for the improvement on Shepherdson's idea in the proof of [**6**, Theorem A3.4] was refined. Second, the universal 1NFA constructed in the proof of [**6**, Theorem 4.2(1)] was observed to not be minimal: it could be implemented with only $4n + 4$ states, as opposed to $8n+3$. Then, a careful application of the reachable-set construction in the proof of [**6**, Theorem 4.5] (on the minimal universal 1NFA obtained previously) revealed the frontier structure.

CHAPTER 2

2D versus 2N

After Chapter 1, our understanding for almost all conversions shown in Figure 1 (page 15) is perfect. The only exceptions are the two conversions associated with the 2D vs. 2N problem, and our understanding of them is so limited that we cannot even tell whether the associated trade-offs are polynomially bounded. In this chapter we will advance our knowledge about these conversions in two quite different directions.

In Section 2, we will focus on the conversion from 1NFAs to 2DFAs and the associated complete problem of liveness. We will prove that a certain class of 2DFAs of *restricted information* fail to solve this problem, no matter how large they are. In Section 3 we will focus on the conversion from 2NFAs to 2DFAs and a certain class of 2NFAs of *restricted bidirectionality*, the sweeping 2NFAs. We will prove that small automata of this kind are not closed under complement. See Section 3 of the Introduction for the motivation behind these two different approaches.

We begin with a brief note on the history of the 2D vs. 2N question.

1. History of the Problem

The 2D vs. 2N question was first studied in the manuscript [**51**]. In it, Seiferas worked on the conversion from 1NFAs to 2DFAs. He suggested the strong conjecture (cf. page 12) that the trade-off is at least $2^n - 1$, and presented several examples of problems that could serve as witnesses. Soon after that, Sakoda and Sipser [**48**] invested the question with a robust theoretical framework (cf. Introduction, Section 3.1). Among other things, they defined the classes 1N, 2D, and 2N, along with the appropriate reduction relation that allowed the identification of complete problems. A 2N-complete and a 1N-complete problem were also defined, the latter being liveness. At the same time, the problems from [**51**] proved to be 1N-complete, too.

In one class of attempts towards 2D $\neq$ 2N, people have focused on proving exponential lower bounds for the trade-off from 1NFAs to 2DFAs of *limited bidirectionality.* Already in [**51**], Seiferas showed that the trade-off is at least $2^n - 1$ if the 2DFAs are *single-pass.* Later, Sipser [**55**] did the same for the case of 2DFAs that are *sweeping*—much later, Leung [**34**] showed the lower bound remains as large even on a binary alphabet, as opposed to the exponentially large one of [**55**]. Recently, Hromkovic and Schnitger [**22**] did the same for the case when the 2DFAs are *oblivious*, in the sense that they move identically on all inputs of the same length—they also showed the lower bound remains exponential if we relax the restriction to allow a sub-linear (in the input length) number of distinct trajectories. Unfortunately, we know that none of these theorems resolves the conjecture in its generality, since full 2DFAs can be exponentially more succinct than each of these restricted variants [**51, 55, 2**].

A second class of attempts has focused on unary automata. Under this restriction, Chrobak [**7**] proved that the trade-off from 1NFAs to 2DFAs is at most $O(n^2)$

and at least $\Omega(n^2)$. Note that, on one hand, this upper bound shows that 2D $\supseteq$ 1N for unary automata, so that the situation on unary inputs is sharply different from what is conjectured to be in general. On the other hand, the lower bound is the best known one even for the trade-off from general 2NFAs to 2DFAs. In two more recent developments, Geffert, Mereghetti and Pighizzini have established the sub-exponential upper bound $2^{\Theta(\lg^2 n)}$ for the trade-off from unary 2NFAs to 2DFAs [**13**], as well as a polynomial upper bound for the trade-off in the complementation of unary 2NFAs [**14**].

Finally, there have also been some variations of the general problem of converting a 2NFA to 2DFA. If we demand that the 2DFA *can* decide identically to the simulated 2NFA *no matter what state and input position the latter is started at* (a condition conceptually stronger than ordinary simulation, but always satisfiable [**5**]), then the trade-off is at least $2^{\lg^k n}$, for any k [**25**]. If we demand that the 2DFA decides identically to the simulated 2NFA *only on all polynomially long inputs* (a requirement conceptually weaker than ordinary simulation), then an exponential lower bound would confirm the old belief that nondeterminism is essential in logarithmic-space Turing machines (L $\neq$ NL) [**3**]. Last, if we allow the starting 2NFA to be a *Hennie machine* (a more powerful device, but still not powerful enough to solve non-regular problems), then converting to a 2DFA indeed costs exponentially, but only because converting to a 2NFA already does [**4**].

2. Restricted Information: Moles

In this section we explore the approach that we described in Section 3.2 of the Introduction. After we formally define what it means for a 2NFA to be a mole, we will move on to prove that two-way deterministic moles cannot solve liveness, irrespective of how large they are.

2.1. Preliminaries. Our notation and definitions are as explained in the previous chapter, in Section 2, plus the following few additional concepts.

If A and B are two sets, then $A \ominus B$ denotes their symmetric difference. If f and g are two functions, then $f \circ g$ and fg denote their composition, returning $g(f(x))$ for every x, while f^k denotes the k-fold composition of f with itself. In contrast, for u a string of symbols, u^k denotes the concatenation of k copies of u.

2.1-I. *Behavior of a* 2DFA. Given any 2DFA M over set of states Q and alphabet Σ and any string $u \in \Sigma^*$, the *behavior of M on u* is the partial mapping γ_u from $Q \times \{\mathtt{l},\mathtt{r}\}$ to $Q \times \{\mathtt{l},\mathtt{r}\}$ that encodes all possible 'entry-exit pairs' as M computes on u: for every $q \in Q$,

$$\gamma_u(q,\mathtt{l}) := \begin{cases} (p,\mathtt{l}) & \text{if } \textsc{lcomp}_{M,q}(u) \text{ hits left into } p, \\ (p,\mathtt{r}) & \text{if } \textsc{lcomp}_{M,q}(u) \text{ hits right into } p, \\ \text{undefined} & \text{if } \textsc{lcomp}_{M,q}(u) \text{ loops or hangs,} \end{cases}$$

while the value $\gamma_u(q,\mathtt{r})$ is defined analogously, with RCOMP instead of LCOMP.

2.1-II. *Strings over Σ_n.* Recall the alphabet Σ_n over which we defined liveness (cf. Introduction, Section 3.1; see also Figure 14 on page 53). A concise way to refer to a symbol of Σ_n is to simply list its arrows in brackets: e.g., the rightmost symbol in Figure 14a is `[12,14,25,44]`. The symbol `[]` containing no arrows is called *the empty symbol.*

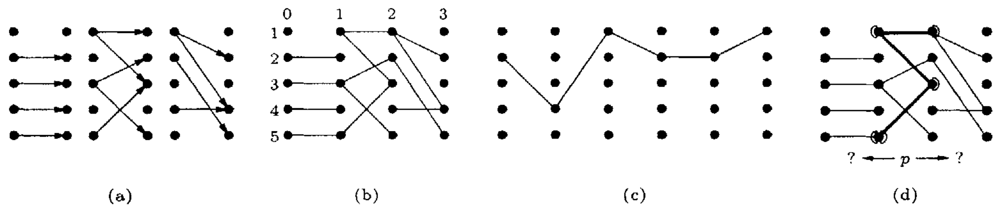


FIGURE 14. (a) Three symbols in Σ_5. (b) The string they define, simplified and indexed; each node has a row and a column index. (c) A 5-long 2-$\{1,2,4\}$-1 path, which is 2-disjoint on itself. (d) State p of focus $(5,\mathtt{l})$ is reading the middle symbol: if it moves right, the next focus will be $(1,\mathtt{l})$ or $(3,\mathtt{l})$; if it moves left, the next focus will be $(1,\mathtt{r})$ or $(5,\mathtt{r})$.

Given any string $x \in \Sigma_n^*$, we define the set of its nodes in the following, quite natural way (Figure 14b):

$$V_x := \{(i,j) \mid i \in [n] \;\; \& \;\; 0 \leq j \leq |x|\}.$$

The *left-degree* of a node $(i,j) \in V_x$ is the number of its neighbors on the column to its left (column $j-1$), or 0 if $j=0$. Similarly, the *right-degree* of (i,j) is the number of its neighbors on the column to its right (column $j+1$), or 0 if $j = |x|$.

If x has exactly $|x|$ edges that form 1 live path, we say x is a *path* (Figure 14c). For $i_{\mathrm{L}}, i_{\mathrm{R}} \in I \subseteq [n]$, we say x is a *i_{L}-I-i_{R} path* if this one live path connects the i_{L}th leftmost node to the i_{R}th rightmost node and visits only nodes with indices in I.

If $y \in \Sigma_n^*$, then $x \cup y$ is the unique string of length $\max(|x|,|y|)$ that has all edges of x, all edges of y, and no other edges. For $k \geq 0$, we say y is *k-disjoint* on x if in $x \cup (\square^k y)$ the edges from x and from y meet at *no* node (Figure 14c, Figure 16c).

2.2. Moles. To define when a 2NFA over Σ_n is a mole, we need a way of describing the notion of a state 'focusing on' some a particular node of the current symbol. We define a *focus* to be any pair $(i,s) \in [n] \times \{\mathtt{l},\mathtt{r}\}$ of *index* and *side*. We write $\bar{s}$ for the side opposite s. The *(i,s)th node* of a string x is the ith node of its leftmost (resp., rightmost) column, if $s = \mathtt{l}$ (if $s = \mathtt{r}$). The connected component of that node in the graph implied by x is called the *(i,s)th component* of x. By $x \restriction (i,s)$ we denote the unique string that has the length of x, all edges of the (i,s)th component of x, and no other edges.

A 2NFA is a mole if each state p of it can be assigned a focus (i_p, s_p) so that, whenever at p, the automaton behaves like a mole located on the (i_p,s_p)th node of the current symbol and facing $\bar{s}_p$: (i) it can 'see' only the component of that node, and (ii) it can 'move' only to nodes in that same component. More carefully:

1. DEFINITION. Let $M = (\cdot,\delta,\cdot)$ be a 2NFA over a set of states Q and the alphabet Σ_n. An *assignment of foci for M* is any mapping $\phi : Q \to [n] \times \{\mathtt{l},\mathtt{r}\}$ such that, for any states $p,q \in Q$, symbol $a \in \Sigma_n$, and side $s \in \{\mathtt{l},\mathtt{r}\}$: whenever M is at p reading a,

(i) its next move depends only on the component containing the node which p is focused on: $\delta(p,a) = \delta(p,\, a \restriction \phi(p)\,)$,

(ii) its next state and position can only be such that the new focused node belongs to the same connected component as the node which p is focused on:

$$\delta(p,a) \ni (q,s) \implies (\exists i \in [n])\big[\, \phi(q) = (i,\bar{s}) \quad \& \quad a \restriction (i,s) = a \restriction \phi(p)\,\big].$$

We say $\phi(p)$ is *the focus of* p. If an assignment of foci for M exists, M is a *mole.*

To understand Condition (ii), consider as an example the case $s = \mathrm{r}$ (see also Figure 14d): If p on a moves *right* into q, then in the new position q must focus on the *left* column ($\phi(q) = (\cdot,\bar{s}) = (\cdot,\mathrm{l})$), the one shared with the previous position. Moreover, if in this column q focuses on the ith node ($\phi(q) = (i,\mathrm{l})$), then in the previous position this node (now the ith node of the *right* column) must belong to the same connected component as the node which p focused on ($a \restriction (i,\mathrm{r}) = a \restriction \phi(p)$).

Note that the 1NFA from page 17 satisfies Definition 1. So, small one-way nondeterministic moles can solve liveness. In contrast, we will prove the following.

2. THEOREM. *Two-way deterministic moles cannot solve liveness.*

Remark that the theorem applies to all two-way deterministic moles, as opposed to only small ones. We also stress that the main purpose of Definition 1 is to disambiguate the intuitive notion of a mole—in contrast, the arguments in our proofs will heavily rely on intuition.

2.3. Mazes. What makes moles so weak is of course the fact that, as they move through the input, they can only observe the part of the graph directly connected to their current location. The rest of the graph is not observable, even if it occupies the same symbols as the observable part, and therefore does not affect the computation. Lemma 3 below turns this intuition into a clean fact that can be used in proofs. Before stating it, we need to talk about mazes and how moles compute on them and their compositions.

Intuitively, a maze is any string on which some nodes have been designated as 'entry-exit gates' for moles (Figure 16d). More carefully, for $x \in \Sigma^*$, let $V_x^0 \subseteq V_x$ consist of every node that has exactly one of its two degrees equal to 0 (and can thus serve as a gate). A *maze* on x is any pair (x, X) where $X \subseteq V_x^0$.

The computation of a mole on a maze is the same object as the computation of any 2NFA on any string, with the extra condition that it 'starts by entering a gate' and 'if it exits a gate, it ends immediately'. Formally, let $\chi = (x, X)$ be a maze, $u = (i,j) \in X$ a gate with 0-degree side s, and p a state of a mole M with focus $\phi(p) = (i,s)$. Then, *the computation* $\textsc{comp}_{M,p,u}(\chi)$ *of* M *on* χ *from* p *and* u (note the overloading of operator COMP) is *a prefix of* either $\textsc{comp}_{M,p,j+1}(x)$ (if $s = \mathrm{l}$) or $\textsc{comp}_{M,p,j}(x)$ (if $s = \mathrm{r}$). The prefix ends the first time (if ever) it reaches a point (q_t, j_t) where the focus $\phi(q_t) = (i', s')$ is on a gate with 0-degree side $\overline{s'}$. Note that x may contain nodes that have degree 0 on one of their two sides but are not gates; the computation may very well visit the 0-degree side of these nodes without having to terminate.

To compose two mazes means to draw their strings on top of each other and then discard all coinciding gates (Figure 16e). More carefully, mazes $\chi = (x, X)$ and $\psi = (y, Y)$ are *composable* iff $|x| = |y|$ (so that $V_x = V_y = V$) and their graphs intersect only at gates and only appropriately: every $v \in V$, *either* has both its degrees equal to 0 in at least one of x, y; *or* is a gate in both mazes, with a different 0-degree side in each of them. If χ, ψ are composable, then their *composition* is the pair $\chi \circ \psi := (x \cup y, X \ominus Y)$. Clearly, the composition is a maze, too.

Note that, by the conditions of composability, in each one of the symbols of $x \cup y$ every non-empty connected component comes entirely from exactly one of x or y. Hence, when a mole reads a symbol, its next step depends on exactly one of x or y. Generalizing, we can prove the following.

3. LEMMA. *Let χ and ψ be as above, and $\omega := \chi \circ \psi$ be their composition. Consider a computation $c := \text{COMP}_{M,p,u}(\chi \circ \psi)$ of a mole M from a gate $u \in X \ominus Y$ that comes from X. A unique list of computations $c_1, c_2, \dots$ exists, such that:*

- *each c_t is a computation of M on χ (resp., on ψ) iff t is* odd (even)*;*
- *c_1 starts from p and u; each c_{t+1} starts from the state and gate where c_t ends;*
- *if we remove the first point of each c_t after c_1 and then concatenate, we get c.*

Put another way, if we can decompose a maze ω into two mazes χ and ψ, then any computation c of a mole on ω can be uniquely decomposed into 'subcomputations' $c_1, c_2, \dots$ that alternate between χ and ψ. We say these computations are the *fragments* of c with respect to the decomposition $\omega = \chi \circ \psi$. Clearly, *either* all fragments are finite, and then their list is infinite iff c is; *or* not all fragments are finite, in which case their list is finite and the only infinite fragment is the last one. Note that a different decomposition of ω leads to a different decomposition of c.

2.4. Hard inputs. In Section 2.5 we will fix an arbitrary deterministic mole and prove that it fails against liveness. To this end, we will construct inputs on which the automaton decides incorrectly. Those fatally hard strings will be extremely long. However, we will build them out of other, much shorter (but still very long) strings, which already strain the ability of the automaton to process the information on its tape. In this section we describe those shorter strings. We start with inputs which can be built for any 2DFA and later (Section 2.4-V) focus on inputs that can be built particularly for deterministic moles. So, fix M to be an arbitrary 2DFA over state set Q and alphabet Σ.

2.4-I. *Dilemmas.* Consider any property $P \subseteq \Sigma^*$ of the strings over Σ, and assume that it is *infinitely extensible to the right*, in the sense that every string that has the property can be right-extend into a strictly longer one that also has it:

$$(\forall y \in P)(\exists z \neq \epsilon)(yz \in P).$$

For example, the property of being of even length is of this kind.

Given any $y \in P$, we can perform the following experiment. For each $p \in Q$, we examine the computation $\text{LCOMP}_{M,p}(y)$ and check if it *hits right*: if it does, we set a bit $a_{y,p}$ to 1; otherwise, the computation *hangs*, *loops*, or *hits left*, and $a_{y,p}$ is set to 0. In the end, we build the bit-vector $a_y := (a_{y,p})_{p \in Q}$. This is our outcome.

How does the outcome change if we right-extend y into some $yz \in P$? How do a_y and a_{yz} compare? For every p, clearly $\text{LCOMP}_{M,p}(y)$ is a prefix of $\text{LCOMP}_{M,p}(yz)$. So, if the first computation hits left, loops, or hangs, so does the second one; but if the first one hits right, there is no guarantee what the second computation does. Hence, all bits in a_y that are 0 keep the same value in a_{yz}; but a bit which is 1 may turn into a 0. Overall, if "$\geq$" is the natural component-wise order, we have the following.

4. LEMMA. *For all $y, yz \in P$: $a_y \geq a_{yz}$.*

What happens to the outcome of the experiment if we further right-extend y into $yzz' \in P$? And then into $yzz'z'' \in P$? While y is infinitely right-extensible

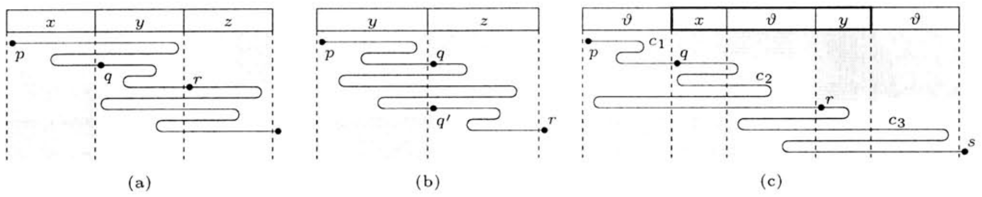


FIGURE 15. 2DFA computations on dilemmas, generic strings, and traps.

inside P, the outcome may decrease only finitely many times. Obviously then, from some point on it must stop changing. When this happens, the extension of y that we have arrived at is very useful tool. The following definition and lemma talk about it formally.

5. DEFINITION. Let $P \subseteq \Sigma^*$. An L-*dilemma* over P is any $y \in P$ such that[1]

$$(\forall yz \in P)(\forall p \in Q)\big[\text{LCOMP}_{M,p}(y) \text{ hits right } \iff \text{LCOMP}_{M,p}(yz) \text{ hits right}\big].$$

An R-*dilemma* over P can be defined symmetrically, on left-extensions and RCOMP.

6. LEMMA. *Suppose $P \subseteq \Sigma^*$. If P is non-empty and infinitely extensible to the right (resp., left), then there exist* L-*dilemmas over P* (R-*dilemmas over P*).

PROOF. Pick any $y \in P$ and keep extending it in the direction of infinite extensibility until a_y stops changing. When it does, the extension is a dilemma. □

In [**51**], dilemmas are called "blocking strings". We now explain these names.

7. LEMMA. *Assume $x \in \Sigma^*$, y is an* L-*dilemma over P, $yz \in P$, and that some computation $c := \text{LCOMP}_{M,p}(xyz)$ crosses the xy-z boundary. After the first such crossing, c never visits x again and it eventually hits right.*

PROOF. Consider the first time c crosses the xy-z boundary (Figure 15a). Let r be the state resulting from this crossing, and q the state resulting from the last crossing of the x-yz boundary before that. Then, the computation between these two crossings is $\text{LCOMP}_{M,q}(y)$ and hits right (into r). Since y is an L-dilemma over P and z does not spoil the property ($yz \in P$), we know that $\text{LCOMP}_{M,q}(yz)$ also hits right. But this computation is a suffix of c. So, c also hits right. Moreover, after crossing the xy-z boundary, it never visits x again. □

In total, once the computation crosses the xy-z boundary, it is restricted inside yz and forced to eventually hit right. Put another way, when M enters y, it faces a 'dilemma': *either* it will stay forever inside xy, never crossing the xy-z boundary; *or* it will cross it, but then also hit right without visiting x again. In effect, y 'blocks' M from returning to x after having seen z —and 'locks' it into hitting right. In yet other words, y makes sure that every left computation of M on xyz that hits left, hangs, or loops does so inside xy, before making it to z.

[1]Note that the displayed condition is the same as $(\forall yz \in P)(a_y = a_{yz})$; but rather more informative. Also note that the "$\Longleftarrow$" part of the equivalence there is given, by Lemma 4. What is important is the "$\Longrightarrow$" part: on every extension in P, the computation will keep hitting right.

2.4-II. *Generic strings.* Consider again a property $P \subseteq \Sigma^*$ which is infinitely extensible to the right. For each $y \in P$, we can define the set of states that can be produced on the rightmost boundary of y by left computations:

$$\text{LSTATES}(y) := \{q \in Q \mid (\exists p \in Q)(\text{LCOMP}_{M,p}(y) \text{ hits right into } q)\}.$$

How does this set change if we extend y into a string $yz \in P$? How does it compare to the set $\text{LSTATES}(yz)$?

Consider the function $\text{LMAP}(y,z)(\cdot)$, defined as follows (Figure 15b): for each $q \in \text{LSTATES}(y)$, the computation $\text{COMP}_{M,q,|y|+1}(yz)$ is examined; if it hits right into some state r, then $\text{LMAP}(y,z)(q) := r$; otherwise, it hits left, loops, or hangs, and $\text{LMAP}(y,z)(q)$ is left undefined.

Note that the values of $\text{LMAP}(y,z)$ are all in $\text{LSTATES}(yz)$. Indeed, if r is such a value, then $r = \text{LMAP}(y,z)(q)$ for some $q \in \text{LSTATES}(y)$. Hence, the computation $\text{COMP}_{M,q,|y|+1}(yz)$ hits right into r and some computation $\text{LCOMP}_{M,p}(y)$ hits right into q. Combining the two, we get the computation $\text{LCOMP}_{M,p}(yz)$, that hits right into r. We thus conclude that $r \in \text{LSTATES}(yz)$, as claimed.

Moreover, the values of $\text{LMAP}(y,z)$ cover $\text{LSTATES}(yz)$. Indeed, if some state $r \in \text{LSTATES}(yz)$, then some computation $c := \text{LCOMP}_{M,p}(yz)$ hits right into r. We know c crosses the y-z boundary, so let q be the state produced by the first such crossing. The computation before this crossing is $\text{LCOMP}_{M,p}(y)$ and hits right into q, so $q \in \text{LSTATES}(y)$. The computation after the crossing is $\text{COMP}_{M,q,|y|+1}(yz)$ and, as a suffix of c, hits right into r. We thus conclude that $\text{LMAP}(y,z)(q) = r$, namely that $\text{LMAP}(y,z)$ covers r.

Overall, $\text{LMAP}(y,z)$ is a *partial surjection* from $\text{LSTATES}(y)$ to $\text{LSTATES}(yz)$. This clearly implies its domain has enough elements to cover the range, so we know $|\text{LSTATES}(y)| \geq |\text{LSTATES}(yz)|$.

The next fact summarizes our findings. Analogously to $\text{LSTATES}(y)$, the set $\text{RSTATES}(z)$ consists of all states that can be produced on the leftmost boundary of z via right computations. Clearly, the symmetric arguments apply. Note that these involve a partial surjection $\text{RMAP}(y,z)$ from $\text{RSTATES}(z)$ to $\text{RSTATES}(yz)$, defined analogously to $\text{LMAP}(y,z)$.

8. LEMMA. *For $y, yz \in P$, the function $\text{LMAP}(y,z)$ partially surjects $\text{LSTATES}(y)$ to $\text{LSTATES}(yz)$; hence $|\text{LSTATES}(y)| \geq |\text{LSTATES}(yz)|$. Similarly, in the opposite direction, if $yz, z \in P$ then the function $\text{RMAP}(y,z)$ partially surjects $\text{RSTATES}(z)$ to $\text{RSTATES}(yz)$; hence $|\text{RSTATES}(yz)| \leq |\text{RSTATES}(z)|$.*

As in Section 2.4-I, we now ask what happens to the size of the set $\text{LSTATES}(y)$ as we keep right-extending y inside P. Although y is infinitely right-extensible, the size of the set can decrease only finitely many times. Hence, from some point on it must stop changing. When this happens, we have arrived at another useful tool.

9. DEFINITION. Let $P \subseteq \Sigma^*$. A string y is L-*generic* over P if $y \in P$ and[2]

$$(\forall yz \in P)\big[|\text{LSTATES}(y)| = |\text{LSTATES}(yz)|\big].$$

An R-*generic* string over P is defined symmetrically, on left-extensions and RSTATES. A string that is simultaneously L-generic and R-generic over P is called *generic*.

[2]Note that the "$\geq$" part of the displayed equality $|\text{LSTATES}(y)| = |\text{LSTATES}(yz)|$ is given, by Lemma 8. What is important is the "$\leq$" part: on every extension in P, the set will manage to stay as large.

10. LEMMA. *Suppose $P \subseteq \Sigma^*$. If P is non-empty and infinitely extensible to the right (resp., left), then there exist* L*-generic strings over P* (R*-generic strings over P). If y_L is* L*-generic and y_R is* R*-generic, then every $y_\text{L}zy_\text{R} \in P$ is generic.*

PROOF. For the last claim, we simply note that every right-extension of an L-generic string inside P is also L-generic. Similarly in the other direction. □

Generic strings were introduced in [**55**], for SDFAs and over the property of liveness. As we will show in the next section, they strengthen dilemmas. Before that, however, let us prove a last fact about the operators LSTATES and RSTATES.

11. LEMMA. *For all $y, z \in P$:* LSTATES$(yz) \subseteq$ LSTATES(z) *and* RSTATES$(y) \supseteq$ RSTATES(yz).

PROOF. We prove the first containment. Pick any $r \in$ LSTATES(yz) and any computation $d :=$ LCOMP$_{M,p}(yz)$ that hits right into r. (Figure 15b.) We know d crosses the y-z boundary. Let q' be the state produced by the *last* such crossing. Then LCOMP$_{M,q'}(z)$ is a suffix of d, and therefore also hits right into r. So, $r \in$ LSTATES(z). □

2.4-III. *Dilemmas versus generic strings.* To examine the relation between dilemmas and generic strings, it is helpful to have the following alternative characterizations of the two classes of strings, in terms of the functions LMAP(y, z) and RMAP(y, z).

12. LEMMA. *Suppose $y \in P \subseteq \Sigma^*$. Then y is an* L-dilemma *over P iff for all $yz \in P$ the function* LMAP(y, z) *is* total. *Similarly for any $z \in P$ and for* R*-dilemmas and* RMAP(y, z).

PROOF. For the forward direction, assume y is an L-dilemma over P. Consider any $yz \in P$ and any $q \in$ LSTATES(y). (Figure 15b.) Let $c :=$ LCOMP$_{M,p}(y)$ be a computation that hits right into q. We know c is a prefix of $d :=$ LCOMP$_{M,p}(yz)$. So, d crosses the y-z boundary (the first such crossing is into q), and hence hits right (by Lemma 7). Therefore, its suffix COMP$_{M,q,|y|+1}(yz)$ hits right, too. This implies that LMAP$(y, z)(q)$ is defined.

For the reverse direction, fix $y \in P$ and suppose LMAP(y, z) is total for all $yz \in P$. Consider any such yz, any $p \in Q$, and assume $c :=$ LCOMP$_{M,p}(y)$ hits right into some state q. (Figure 15b.) Then $q \in$ LSTATES(y). Therefore, LMAP$(y, z)(q)$ is defined. This implies $c' :=$ COMP$_{M,q,|y|+1}(yz)$ hits right. Combining c and c', we get the computation $d :=$ LCOMP$_{M,p}(yz)$. As c' is a suffix of d, we know d hits right as well. □

13. LEMMA. *Suppose $y \in P \subseteq \Sigma^*$. Then y is* L-generic *over P iff for all $yz \in P$ the function* LMAP(y, z) *is* total *and* bijective. *Similarly for $z \in P$ being* R*-generic and for* RMAP(y, z).

PROOF. For the forward direction, suppose y is L-generic and pick any $yz \in P$. We already know LMAP(y, z) is a partial surjection from LSTATES(y) to LSTATES(yz). Since y is L-generic, we also know the two sets have the same size. So, LMAP(y, z) must be total and injective. Conversely, fix $y \in P$ and suppose $a_{y,z}$ is total and bijective for all $yz \in P$. Then clearly, for every such yz, the sets LSTATES(y) and LSTATES(yz) must have the same size. □

Intuitively, a dilemma guarantees that the computations that manage to survive through it will also survive through every extension that preserves the property. A generic string guarantees that, in addition, the computations will keep exiting each extension into different states.

14. LEMMA. *Let $P \subseteq \Sigma^*$. Over P, every* L*-generic string is an* L*-dilemma and every* L*-dilemma is right-extensible into an* L*-generic string. Similarly for* R*-generic strings,* R*-dilemmas, and left-extensions.*

PROOF. Lemmata 12 and 13 prove the first claim. For the second claim, we simply note that *every* string in P can be right-extended into L-generic strings. □

2.4-IV. *Traps.* Consider a property $P \subseteq \Sigma^*$ which is infinitely extensible in either direction and closed under concatenation. For this section, fix ϑ as a generic string over P, and let

$$L := \text{RSTATES}(\vartheta), \qquad R := \text{LSTATES}(\vartheta),$$

denote the sets of states producible on the leftmost and rightmost boundary of ϑ. Note that, by Lemma 14, we know ϑ is both an L-dilemma and an R-dilemma.

A *trap* (on ϑ) is any string of the form $\vartheta x \vartheta$, where $x \in P$ is the *infix.*

By Lemma 10 and the closure of P under concatenation, traps are still generic strings. However, they further restrict M's freedom: By Lemma 13, the function $\text{LMAP}(\vartheta, x\vartheta)$ is a total bijection from $\text{LSTATES}(\vartheta) = R$ to $\text{LSTATES}(\vartheta x \vartheta)$. Since $\text{LSTATES}(\vartheta x \vartheta) \subseteq R$ (by Lemma 11), $\text{LMAP}(\vartheta, x\vartheta)$ is a total bijection from R to a subset of R. Clearly, this is possible only if this subset is R itself. So, $\text{LMAP}(\vartheta, x\vartheta)$ simply permutes R. To simplify notation, we denote this permutation by α_x. Namely,

$$\alpha_x := \text{LMAP}(\vartheta, x\vartheta).$$

Similarly, $\text{RMAP}(\vartheta x, \vartheta)$ permutes L, and we denote this permutation as β_x. Overall, we have proved the following.

15. LEMMA. *For all $x \in P$: α_x permutes R and β_x permutes L.*

Intuitively, in each direction, the computations that manage to cross the first copy of ϑ eventually cross the entire trap; but, after this first copy, they collectively do nothing more than simply permute the set of states that they have already produced. As we now show, the two permutations fully describe the behavior of M on the trap.

16. LEMMA. *For all infixes $x, y \in P$: $(\alpha_x, \beta_x) = (\alpha_y, \beta_y) \implies \gamma_{\vartheta x \vartheta} = \gamma_{\vartheta y \vartheta}$.*

PROOF. Suppose $(\alpha_x, \beta_x) = (\alpha_y, \beta_y)$ and consider any $p \in Q$. We show $\gamma_{\vartheta x \vartheta}$ and $\gamma_{\vartheta y \vartheta}$ agree on (p, l)—the proof for (p, r) is similar. We examine the computations $c_x := \text{LCOMP}_{M,p}(\vartheta x \vartheta)$ and $c_y := \text{LCOMP}_{M,p}(\vartheta y \vartheta)$. Clearly, these behave identically up to the first crossing of the 'critical' boundary between ϑ and $x\vartheta$ or $y\vartheta$. *If one of them hits left, loops, or hangs,* it does so inside ϑ (since ϑ is an L-dilemma) without crossing the critical boundary; so, the other computation behaves identically, thus $\gamma_{\vartheta x \vartheta}(p, \text{l}) = \gamma_{\vartheta y \vartheta}(p, \text{l})$. *If one of them hits right,* then it crosses the critical boundary into some state q and so does the other one; but then they both hit right, into the same state $r := \alpha_x(q) = \alpha_y(q)$, so $\gamma_{\vartheta x \vartheta}(p, \text{l}) = \gamma_{\vartheta y \vartheta}(p, \text{l}) = (r, \text{r})$. □

We call (α_x, β_x) the *inner-behavior* of M on the trap $\vartheta x \vartheta$, to distinguish from $\gamma_{\vartheta x \vartheta}$.

An interesting case arises when ϑ is an infix of the infix itself. Then the inner-behavior of M on the trap can be deduced from its inner-behavior on the traps that are induced by the other two pieces of the infix.

17. LEMMA. *Suppose $x, y \in P$ and $z := x\vartheta y$. Then $(\alpha_z, \beta_z) = (\alpha_x \circ \alpha_y, \beta_y \circ \beta_x)$.*

PROOF. To show that $\alpha_z = \alpha_x \circ \alpha_y$ (the argument for $\beta_z = \beta_y \circ \beta_x$ is similar), we pick an arbitrary $q \in R$ and show that $\alpha_z(q) = \alpha_y(\alpha_x(q))$. (Figure 15c.) We know q is produced by some right-hitting left computation on ϑ, say $c_1 := \text{LCOMP}_{M,p}(\vartheta)$ for some state p. Since ϑ is an L-dilemma over P and $\vartheta z \vartheta \in P$, we know $c := \text{LCOMP}_{M,p}(\vartheta z \vartheta)$ also hits right, into some state s. Therefore, $\alpha_z(q) = s$. Before hitting right, c surely crosses the $\vartheta x \vartheta$-$y\vartheta$ boundary; let r be the state produced by the first such crossing. Clearly, the computation $c_2 := \text{COMP}_{M,q,|\vartheta|+1}(\vartheta x \vartheta)$ hits right into r, and hence $\alpha_x(q) = r$. Moreover, the suffix of c after the first crossing of the $\vartheta x \vartheta$-$y\vartheta$ boundary is $c_3 := \text{COMP}_{M,r,|\vartheta x \vartheta|+1}(\vartheta x \vartheta y \vartheta)$ and obviously hits right into s. However, since ϑ is an L-dilemma over P and $\vartheta y \vartheta \in P$, we know c_3 never visits the prefix ϑx. Hence, it can also be written as $c_3 = \text{COMP}_{M,r,|\vartheta|+1}(\vartheta y \vartheta)$. Since it hits right into s, we conclude that $\alpha_y(r) = s$. Overall, $\alpha_z(q) = s = \alpha_y(r) = \alpha_y(\alpha_x(q))$. $\square$

An obvious generalization holds when the infix contains multiple copies of ϑ. In a particular case of interest, the infix consists of several ϑ-separated copies of some $x \in P$. Specifically, for any $k \geq 1$, we define $x^{(k)} := x(\vartheta x)^{k-1}$ and prove the following.

18. LEMMA. *For any $x \in P$ and any $k \geq 1$: $(\alpha_{x^{(k)}}, \beta_{x^{(k)}}) = ((\alpha_x)^k, (\beta_x)^k)$.*

2.4-V. *Hard inputs to deterministic moles.* We now assume that the 2DFA M of the previous sections is defined over Σ_n and that it is actually a mole. We will design inputs on which M misses a significant amount of information. All these inputs are going to be paths (cf. Section 2.1-II).

We fix some $I \subseteq [n]$ and $i \in I$, and consider the set $\Pi \subseteq \Sigma_n^*$ of all i-I-i paths. Clearly, Π is non-empty, infinitely extensible in both directions, and closed under concatenation. Hence, by Lemma 10, generic strings over Π exist. We fix ϑ to be one, and let $\kappa := |\vartheta|$. We also set $L := \text{RSTATES}(\vartheta)$, $R := \text{LSTATES}(\vartheta)$, and let $\mu := \text{lcm}(|L|!, |R|!)$ be the least common multiple of the sizes of the corresponding permutation groups.

For every length $l \geq 1$, we consider all traps (on ϑ) with infixes of length l and collect into a set Ω_l all inner-behaviors that M exhibits on these traps:

$$\Omega_l := \{(\alpha_x, \beta_x) \mid x \text{ is an } i\text{-}I\text{-}i \text{ path of length } l\}.$$

As shown in the next fact, every inner-behavior that can be induced by an l-long infix can also be induced by an infix of length $l + 2\mu(l + \kappa)$. The subsequent fact explains that sometimes the converse is also true.

19. LEMMA. *For every $l \geq 1$: $\Omega_l \subseteq \Omega_{l+2\mu(l+\kappa)}$.*

PROOF. Pick any behavior $(\alpha, \beta) \in \Omega_l$. We know that some l-long infix $x \in \Pi$ induces this behavior, namely $(\alpha, \beta) = (\alpha_x, \beta_x)$. Consider the path $x^{(2\mu+1)} = x^{(\mu)}\vartheta x \vartheta x^{(\mu)}$. This is also in Π and of length $(2\mu + 1)l + 2\mu\kappa = l + 2\mu(l + \kappa)$. Moreover, by Lemma 18 and the selection of μ, we know that this path induces the behavior $(\alpha_x^{2\mu+1}, \beta_x^{2\mu+1}) = \big((\alpha_x)^{2\mu}\alpha_x, \beta_x(\beta_x)^{2\mu}\big) = (\alpha_x, \beta_x) = (\alpha, \beta)$. Hence, $(\alpha, \beta) \in \Omega_{l+2\mu(l+\kappa)}$. □

20. LEMMA. *There exist*[3] $l \geq 1$ *such that* $\Omega_l = \Omega_{l+2\mu(l+\kappa)}$.

PROOF. As the constant $(|L|!) \times (|R|!)$ upper bounds the sizes of all sets $\Omega_1, \Omega_2, \ldots$, we know at least one of them is of maximum size. Pick l so that Ω_l is such. Then both $\Omega_l \subseteq \Omega_{l+2\mu(l+\kappa)}$ (by Lemma 19) and $|\Omega_l| \geq |\Omega_{l+2\mu(l+\kappa)}|$ (by the selection of l). Necessarily then, the two sets must be equal. □

Intuitively, for the two lengths l and $l+2\mu(l+\kappa)$, this last fact says that *between two copies of ϑ, every i-I-i path of either length can be replaced by some path of the other length* without M noticing the trick (recall Lemma 16).

2.5. The proof. We now fix an arbitrary deterministic mole $M = (s, \delta, f)$ over Σ_5 and prove that it fails to solve liveness. To this end, in Sections 2.5-II and 2.5-III we construct a maze that 'confuses' M. Our most important building blocks are the paths of the next section.

2.5-I. *Three special paths.* In this section we fix $n := 5$, $i := 2$, $I := \{1, 2\}$. For these n, i, and I, we fix Π, ϑ, κ and μ as in Section 2.4-V, let λ be a length as in Lemma 20, and set $\Lambda := 2\mu(\lambda + \kappa)$.

21. LEMMA. *There exist paths $\pi, \rho, \sigma \in \Pi$ such that*
- *M cannot distinguish among them: $\gamma_\pi = \gamma_\rho = \gamma_\sigma$.*
- *ρ is Λ-disjoint on itself, and π is Λ-disjoint on σ.*
- *π is Λ-shorter than ρ, and ρ is Λ-shorter than σ: $|\rho| - |\pi| = |\sigma| - |\rho| = \Lambda$.*
- *π is non-empty but short: $0 < |\pi| \leq \Lambda$.*

PROOF. Each of π, ρ, σ is a trap on ϑ. We will select their infixes $x, y, z \in \Pi$.

We set $\rho := \vartheta y \vartheta$, where y has length $\lambda + \Lambda$ and guarantees ρ is Λ-disjoint on itself. Constructing y is straightforward (Figure 16a): We pick paths

$\eta :=$ *any* 2-I-1 path of length λ,
$\vartheta' :=$ *the* 1-I-1 path of length κ that is 0-disjoint on ϑ,
$\iota :=$ *any* 1-I-1 path of length $\Lambda - (2\kappa + \lambda)$, and
$\eta' :=$ *the* 1-I-2 path of length λ that is 0-disjoint on η.

Then, setting $y := \eta\vartheta'\iota\vartheta'\eta'$ we see that this is indeed a 2-I-2 path of length $\lambda + \Lambda$; and shifting $\rho = \vartheta y \vartheta = \vartheta\eta\vartheta'\iota\vartheta'\eta'\vartheta$ on a copy of itself by $\Lambda = |\vartheta\eta\vartheta'\iota|$ causes only its prefix $\vartheta\eta\vartheta'$ to overlap with the 'mirroring' suffix $\vartheta'\eta'\vartheta$, so that no vertex is shared (Figure 16b).

We set $\pi := \vartheta x \vartheta$, where x has length λ and guarantees π is indistinguishable to ρ. Selecting x is easy: Since y is of length $\lambda+\Lambda$, the inner-behavior (α_y, β_y) of M on ρ is in $\Omega_{\lambda+\Lambda}$, and therefore in Ω_λ. Hence, there exist λ-long paths that induce this inner-behavior. Picking x to be such a path, we know that $(\alpha_x, \beta_x) = (\alpha_y, \beta_y)$ and hence $\gamma_\pi = \gamma_\rho$.

We set $\sigma := \vartheta z \vartheta$, where z has length $\lambda + 2\Lambda$ and guarantees that π is Λ-disjoint on σ and that σ is indistinguishable to π. Note that, given the lengths of x and z, the disjointness condition amounts to saying that π and σ should not intersect when 'centered' on top of each other. The construction of z is trickier.

We start by selecting a path y' that is as long as y (namely, of length $\lambda + \Lambda$) and does not intersect π when the two paths are 'centered' on top of each other

[3]The argument essentially shows an infinity of such l exists, but this will not be needed here.

(namely, $\vartheta x\vartheta$ is $(\frac{\Lambda}{2}-\kappa)$-disjoint on y'). This selection is trivial: We simply take the unique 1-I-1 path that is as long as π (namely, of length $\lambda+2\kappa$) and 0-disjoint on it, then extend it by $\frac{\Lambda}{2}-\kappa$ in both directions into any 2-I-2 path.

Now, the inner-behavior $(\alpha_{y'},\beta_{y'})$ of M on $\vartheta y'\vartheta$ is in $\Omega_{\lambda+\Lambda}$, and hence in Ω_λ. Therefore, we can find an λ-long $x'\in\Pi$ that induces the same behavior, $(\alpha_{x'},\beta_{x'})=(\alpha_{y'},\beta_{y'})$. We set $z:=(x')^{(\mu)}\vartheta y'\vartheta(x')^{(\mu-1)}\vartheta x$, the path containing $2\mu+1$ ϑ-separated paths, all copies of x' except the middle and rightmost ones, which copy y' and x.

The length of z is indeed $\lambda+2\Lambda$. Moreover, $\sigma=\vartheta z\vartheta$ symmetrically extends y' by $|\vartheta(x')^{(\mu)}\vartheta|=|\vartheta(x')^{(\mu-1)}\vartheta x\vartheta|=\frac{\Lambda}{2}+\kappa$, which in turn symmetrically out-lengths π by $\frac{\Lambda}{2}-\kappa$. Overall, σ symmetrically out-lengths π by Λ without intersecting it. That is, π is Λ-disjoint on σ. Finally, the inner-behavior (α_z,β_z) of M on σ is

$$\big((\alpha_{x'})^\mu\alpha_{y'}(\alpha_{x'})^{\mu-1}\alpha_x,\,\beta_x(\beta_{x'})^{\mu-1}\beta_{y'}(\beta_{x'})^\mu\big)=\big((\alpha_{x'})^{2\mu}\alpha_x,\,\beta_x(\beta_{x'})^{2\mu}\big)=(\alpha_x,\beta_x),$$

where we used Lemmata 17 and 18, and the selection of μ. Hence, $\gamma_\sigma=\gamma_\pi$. $\square$

2.5-II. *A maze of questions.* We start with the two strings $\tau_1:=\texttt{[]}^{3\Lambda}\rho\texttt{[]}$ and

$$\tau_2:=\texttt{[33]}^{\Lambda-1}\texttt{[32]}\,\texttt{[22]}^{\Lambda-1}\texttt{[23]}\,\texttt{[33]}^{\Lambda-1}\texttt{[32,34]}\,\texttt{[45]}\,\texttt{[55]}^{\Lambda-1}\texttt{[54]}\,\texttt{[44]}^{|\pi|-1}\texttt{[23,43]},$$

(Figure 16f) which are equally long and each is Λ-disjoint on itself (recall the selection of ρ). Also, in $\tau:=\tau_1\cup\tau_2$ their graphs intersect only at the endpoints of ρ, so that τ is Λ-disjoint on itself, too. This implies that τ^i is also Λ-disjoint on itself, for all $i\geq 1$ (Figure 16g).

Let $P:=\{\tau^i \mid i\geq 1\}$ be the set of all powers of τ. Select τ_{L} and τ_{R} as L- and R-dilemmas over P. Fix $m:=2^{|Q|}+1$. The live string $z:=\tau_{\mathrm{L}}\tau^m\tau_{\mathrm{R}}$ is also a power of τ and in it we think of the m 'middle' copies of τ as *distinguished.* On this string, we consider the natural maze $\omega=(z,Z):=(z,\{u,v\})$, where $u:=(3,0)$ and $v:=(3,|z|)$.

Consider the $|Q|$ computations of the form $\textsc{comp}_{M,p,\varepsilon}(\omega)$ that we get as we vary $p\in Q$ and pick $\varepsilon:=u$ when p focuses on the left ($\phi(p)=(\cdot,1)$), and $\varepsilon:=v$ otherwise. Some of them are infinite (i.e., they loop) or finite but non-crossing (i.e., they hang; or they start and end on the same gate). We disregard them and keep only those that are *crossing* (i.e., they start and end in different gates). Let k be their number. Clearly, $k\leq|Q|$.

Fix d to be any of these k computations and fix $1\leq i\leq m$. We know d 'visits' the ith distinguished copy of τ, and we want to discuss its behavior there. In particular, we want to consider the parity $b_{i,d}\in\{0,1\}$ of the number of times that d 'fully crosses' the copy of ρ in the ith distinguished copy of τ. A careful definition of $b_{i,d}$ follows.

If we 'rip off' ρ from the ith distinguished copy of τ and then add the two endpoints u_i, v_i of the path as new gates, we construct a new maze,

$$\chi_i:=\big((\tau_{\mathrm{L}}\tau^{i-1})\,\tau_2\,(\tau^{m-i}\tau_{\mathrm{R}}),\;\{u,v,u_i,v_i\}\big).$$

By the 'complementary' operation, where we rip off everything *except* the particular copy of ρ, we can construct the 'complementary' maze,

$$\psi_i:=\big((\texttt{[]}^{|\tau_{\mathrm{L}}\tau^{i-1}|})\,\texttt{[]}^{3\Lambda}\rho\texttt{[]}\,(\texttt{[]}^{|\tau^{m-i}\tau_{\mathrm{R}}|}),\;\{u_i,v_i\}\big).$$

Clearly, $\omega=\chi_i\circ\psi_i$, and d is a finite computation on this composition. By Lemma 3, we can break d into its finitely many, finite fragments $d_1,d_2,\ldots,d_\nu$. We know every even(-indexed) fragment is a computation on ψ_i; we call it *crossing* if its starting

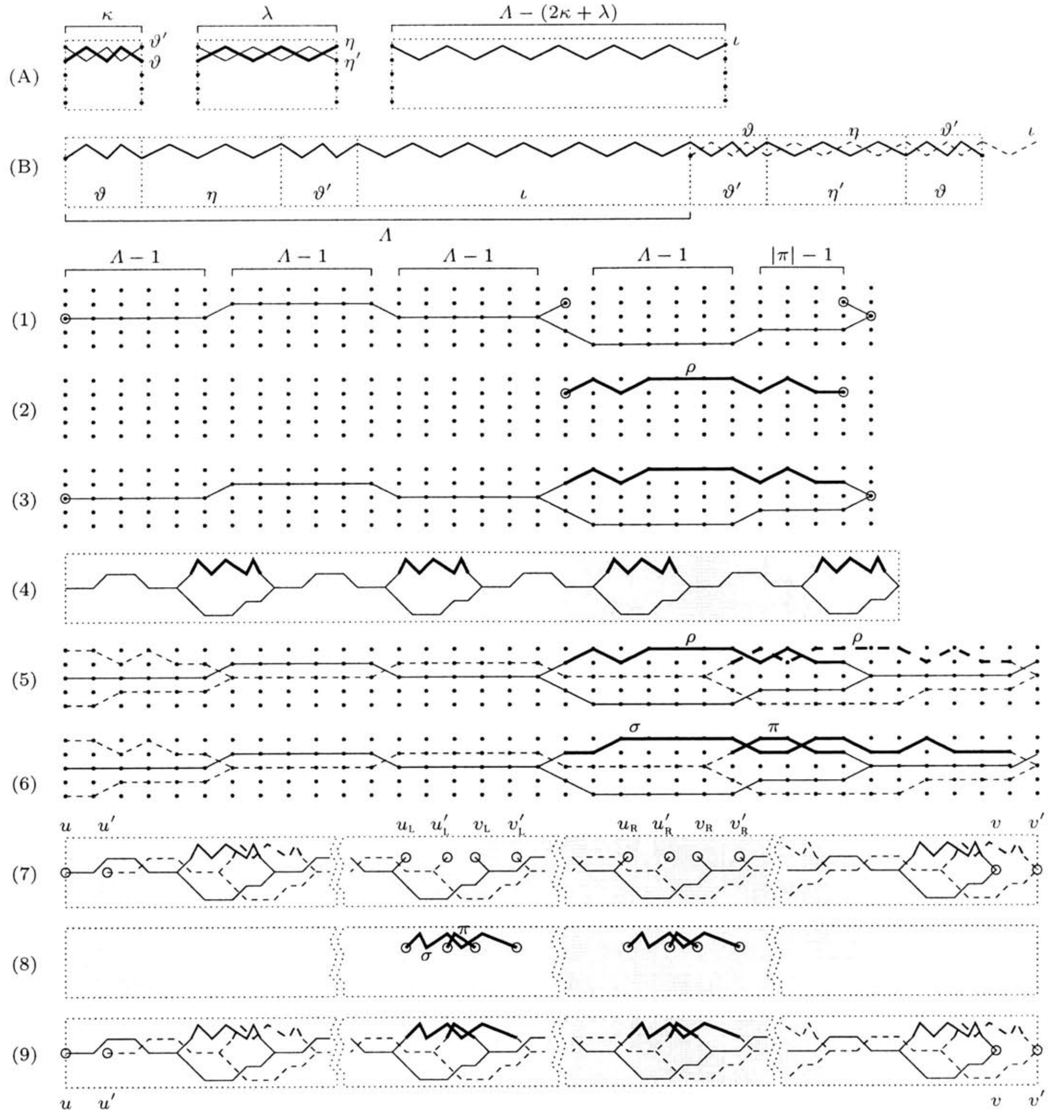


FIGURE 16. (a) in A: picking η, ι; then the mirrors ϑ', η'. (b) in B: the path ρ and how it is Λ-disjoint on itself. (c) in each of 1, 2, 3: a 29-long string, 6-disjoint on itself; see 5. (d) in each of 1, 2, 3: a maze; gates marked with circles. (e) in 3: the composition of the mazes of 1, 2. (f) in 1, 2, 3: examples of τ_2, τ_1, τ, respectively, for a schematic case $\Lambda = 6$, $|\pi| = 4$, and a schematic ρ. (g) in 4: a schematic of τ^4; in 5: a snippet of the union of a τ^i with a Λ-shifted copy of itself. (h) in 7: a schematic of χ', focusing on the snippets around the leftmost, i_1th distinguished, i_2th distinguished, and rightmost pairs of copies of τ. (i) in 8: a schematic of ψ', for the same snippets. (j) in 9: a schematic of $\omega' = \chi' \circ \psi'$, for the same snippets; in 5, 6: a better view of how σ, π connect the two disjoint graphs of x' when they replace two copies of ρ.

and ending gates differ. The bit $b_{i,d}$ records the parity of the number of such fragments. In other words: $b_{i,d} = 0 \iff d$ exhibits an even number of crossing even fragments.

Intuitively, as the mole develops a crossing computation on ω, each distinguished copy of τ asks: "odd or even?" The mole answers this question with the parity of the number of times that it fully crosses ρ in that copy. The bits $b_{i,d}$ record exactly these answers.

Organizing these $m \times k$ bits into m k-long vectors $b_i := (b_{i,d})_d$, for $i = 1, \dots, m$, we see that there are more vectors than values for them: $2^k \leq 2^{|Q|} < 2^{|Q|} + 1 = m$. Hence, $b_{i_1} = b_{i_2}$ for some $1 \leq i_1 < i_2 \leq m$. This means that, in each crossing finite computation, the answer to the i_1th question equals the answer to the i_2th one.

2.5-III. *A more complex maze.* We now return to $\omega = (z, \{u, v\})$. We remove ρ from the i_1th and i_2th distinguished copies of τ, and name the four natural new gates u_L, v_L (endpoints of ρ in the i_1th copy) and u_R, v_R (endpoints of ρ in the i_2th copy) to get the new maze

$$\chi = (x, X) := \big((\tau_\text{L}\tau^{i_1-1})\ \tau_2\ (\tau^{i_2-i_1-1})\ \tau_2\ (\tau^{m-i_2}\tau_\text{R}),\ \ \{u, v, u_\text{L}, v_\text{L}, u_\text{R}, v_\text{R}\}\big).$$

As previously, the 'complementary' maze (remove everything *except* the two ρ's) is

$$\psi = (y, Y) := \big((\cdots)\ [\,]^{3\Lambda}\rho[\,]\ (\cdots)\ [\,]^{3\Lambda}\rho[\,]\ (\cdots),\ \ \{u_\text{L}, v_\text{L}, u_\text{R}, v_\text{R}\}\big),$$

where ellipses stand for appropriately many $[\,]$s. Obviously, $\omega = \chi \circ \psi$. In this section, we will construct a maze $\omega' = \chi' \circ \psi'$, where the mazes χ' and ψ' are complex versions of χ and ψ.

We start by noting that x is Λ-disjoint on itself (because z is). So, in the union $x' := x \cup ([\,]^\Lambda x)$ of x with a Λ-shifted copy of itself, the two graphs do not intersect. (Figure 16h.) So, letting $\chi' := (x', X')$, where $X' := X \cup \{u', v', u'_\text{L}, v'_\text{L}, u'_\text{R}, v'_\text{R}\}$ contains all gates of χ plus their counterparts in the shifted copy, we know every computation on χ' visits and depends on exactly one of the two disjoint graphs.

Similarly, y is Λ-disjoint on itself (because ρ is), the union $y \cup ([\,]^\Lambda y)$ contains two pairs of disjoint copies of ρ, and $Y' := Y \cup \{u'_\text{L}, v'_\text{L}, u'_\text{R}, v'_\text{R}\}$ contains their endpoints. Viewing each pair of copies of ρ as a copy of the string $\rho \cup ([\,]^\Lambda \rho)$, we can replace it with a copy of the string $\rho' := \sigma \cup ([\,]^\Lambda \pi)$. If y' is the new string, we set $\psi' := (y', Y')$. (Figure 16i.) Crucially, this substitution preserved (i) *the lengths of strings*: $|y'| = |y \cup ([\,]^\Lambda y)|$, because

$$|\rho'| = |\sigma \cup ([\,]^\Lambda \pi)| = |\sigma| = 2\kappa + \lambda + 2\Lambda = |\rho| + \Lambda = |\rho \cup ([\,]^\Lambda \rho)|;$$

(ii) *the number and disjointness of paths*: since π is Λ-disjoint on σ, we know ρ' also contains two disjoint paths; and (iii) *the set of endpoints of paths*: e.g., on the copy of ρ' on the left, σ and π have endpoints u_L, v'_L and u'_L, v_L. Note that every computation on ψ' visits and depends on exactly one of the paths.

Clearly, the graphs of x' and y' intersect only at the gates in Y'. Therefore, χ' and ψ' are composable, into $\omega' = (z', Z') := \chi' \circ \psi' = (x' \cup y', \{u, v, u', v'\})$. (Figure 16j.) Note that u and u' are on the far left; v and v' are on the far right; and the four paths of y' connect the two graphs of x': the mole can switch graphs only if it fully crosses one of the paths.

2.5-IV. *The hidden gate.* Consider the dead input $z'[\,]$ and the computation $c' := \text{LCOMP}_{M,s}(\vdash z'[\,]\dashv)$ on it. From now on, our goal is to prove that c' never visits $[\,]$. Equivalently, we want to show that M never visits the 0-degree side of the rightmost node v' of z'. Intuitively, this is the same as saying that *the maze* *implied by z' hides v' from the mole.* Note that this will immediately imply the failure of M: on the *live* input z'[33] the mole will compute exactly as on the *dead* input z[], as it will never visit the 0-degree side of v' to note the difference.

We start by remarking that, since the first symbol of z' is [33], any attempt of the mole to depart from $\vdash$ into a state of focus other than $(3,1)$ is followed by a step back to $\vdash$. Ignoring these attempts and also noting that the mole can never move past [], we see that c' consists essentially of zero or more computations of the form $\text{COMP}_{M,p,1}(z'[])$ with $\phi(p) = (3,1)$. For our purposes, it is enough to study the case where c' consists of exactly one such computation.

So, suppose $c' := \text{COMP}_{M,p,1}(z'[])$, where $\phi(p) = (3,1)$. As a mole, every time M visits the 0-degree side of the nodes u', v, v', it changes direction to 'return into the graph' of z'. Call every such move a *turn* and break c' into *segments* $c'_1, c'_2, \ldots$ so that successive segments are joined at a turn: the later segment starts at the state and position following the last state and position of the earlier segment. Clearly: each segment is a computation on ω'; the first segment is $c'_1 = \text{COMP}_{M,p,u}(\omega')$, but later segments start at a gate in $\{u', v, v'\}$; and *either* all segments are finite, in which case their list is finite iff c' is, *or* not, in which case the list is finite and only the last segment is infinite.

To prove that c' never visits [], it is enough to show that no segment ends in v'. This, in turn, is a corollary of the following:

- the first segment starts at gate u,
- every finite segment that starts at gate u and does not hang necessarily ends either at gate u or at gate v, and
- every finite segment that starts at gate v and does not hang necessarily ends either at gate u or at gate v.

We only prove the second statement, in the next section. The third statement can be proved by a similar argument, whereas the first statement is already known.

2.5-V. *The final argument.* Let d' be a non-hanging finite segment of c' that starts at u. As a finite computation on $\omega' = \chi' \circ \psi'$, it can be broken into finitely many, finite fragments $d'_1, d'_2, \ldots, d'_\nu$; odd(-indexed) fragments compute on χ' and even(-indexed) fragments compute on ψ' (Lemma 3). By previous remarks, every odd fragment visits and depends on exactly one of the two graphs (non-shifted and shifted) inside x'; and every even fragment visits and depends on exactly one of the four paths in y'. Calling an even fragment *crossing* if its start and final gates differ, we clearly see that two successive odd fragments visit different graphs in x' iff the even fragment between them is crossing. Generalizing, and since d' starts on u, *each odd fragment visits the* shifted *graph in x' iff the number of crossing even fragments that precede it is* odd.

Towards a contradiction, assume d' does not end in u or v. Then it ends in either u' or v'. Hence, d'_ν is an odd fragment that visits the shifted graph in x'. This immediately implies that *the total number of crossing even fragments* (before d'_ν, and so throughout d') *is odd.* In particular, even fragments exist and d'_1 *necessarily ends at a gate in Y.*

To reach a contradiction, we will show that, by replacing every fragment d'_i of d' with an appropriate computation d_i on the original maze ω, we can create a computation d on ω that cannot really exist. Before we start, let $h : X' \to X$ be the function that maps every gate in X' to its 'unprimed' version in X: for example, $h(u_\text{L}) = h(u'_\text{L}) = u_\text{L}$. We distinguish two cases:

- *If d'_i is an odd fragment* (a computation on exactly one of the two graphs in χ') from state q and gate ε to state r and gate ζ, we let d_i be the computation on (the one graph of) χ from q and $h(\varepsilon)$. Clearly, d_i ends at r and $h(\zeta)$. In particular, d_1 starts at $h(u) = u$ and ends at a gate in $h(Y) = Y$.
- *If d'_i is an even fragment* (a computation on exactly one of the four paths in ψ') from state q and gate ε to state r and gate ζ, we let d_i be the computation on (one of the two copies of ρ in) ψ from q and $h(\varepsilon)$. Since ρ is indistinguishable from each of π and σ, we know d_i ends at r and $h(\zeta)$. Note here the critical use of the inability of the mole to detect the big difference in the lengths of π, ρ, and σ.

Reviewing the list $d_1, d_2, \dots, d_\nu$, we see that: d_1 starts at $h(u) = u$; for every $1 \leq i < \nu$, fragment d_i ends at the state and gate where d_{i+1} starts; fragment d_ν ends on $h(u') = u$ or $h(v') = v$; and every even fragment d_i is crossing (on the path of ψ that it visits) iff d'_i is (on the path of ψ' that it visits). Hence, by concatenating, we can build a computation d on $\chi \circ \psi = \omega$ that starts at u, ends at u or v, and contains an odd number of crossing even fragments.

But is this possible?

If d ends at u, then it never moves beyond τ_{L} (if it did, it would traverse the L-dilemma and get 'blocked' away from u). In particular, d_1 never reaches a gate in Y. But (by a previous remark) this is where it is supposed to end. Contradiction.

If d ends in v, then it is a crossing computation on ω. As ω equals each of the compositions $\chi \circ \psi$, $\chi_{i_1} \circ \psi_{i_1}$, and $\chi_{i_2} \circ \psi_{i_2}$, we know d can be fragmented in three different ways. Clearly, every *even* fragment with respect to either $\chi_{i_1} \circ \psi_{i_1}$ or $\chi_{i_2} \circ \psi_{i_2}$ is also an *even* fragment with respect to $\chi \circ \psi$, and vice versa; and is *crossing* or not (on the copy of ρ that it visits) irrespective of which composition we look at it through. So, letting ξ, ξ_1, ξ_2 be the numbers of *crossing even* fragments with respect to the three compositions, we know $\xi = \xi_1 + \xi_2$ and (as established above) ξ is odd. Yet, by the selection of i_1 and i_2, the parities of ξ_1, ξ_2 are respectively $b_{i_1,d}$, $b_{i_2,d}$ and hence equal (as $b_{i_1} = b_{i_2}$), so that ξ should be even. Contradiction.

So, in both cases we reach a contradiction, as desired. ☐

3. Restricted Bidirectionality: Sweeping Automata

In this section we explore the approach that we described in Section 3.3 of the Introduction. After we formally define what it means for a 2NFA to be sweeping, we will prove that every sweeping 2NFA solving $\overline{B_n}$ needs $2^{\Omega(n)}$ states—here, $\overline{B_n}$ is the complement of liveness.

3.1. Preliminaries.

Our basic notation and definitions are as presented in Section 2 of Chapter 1. However, we will also need some extra notions and facts of special interest to this section. We present this additional material below.

3.1-I. *Sets, functions, and relations.* As usual, for any set U, we write $\overline{U}$, $|U|$, $\mathcal{P}(U)$, and U^2 for the complement, the size, the powerset, and the set of pairs of U. The following simple lemma plays a central role in our proof.

22. Lemma. *Let $(u_i)_{i \in I}$ and $(v_i)_{i \in I}$ be two sequences of subsets of a set U, where I is a set of indices totally ordered by $<$. If for all $i', i \in I$ we have*

$$i' < i \implies u_{i'} \cap v_i = \emptyset \qquad \text{and} \qquad i' = i \implies u_{i'} \cap v_i \neq \emptyset,$$

then $|I| \leq |U|$.

PROOF. For each $i \in I$, let a_i be any element of the non-empty intersection $u_i \cap v_i$. If the list $(a_i)_{i \in I}$ contains a repetition, say $a_{i'} = a_i =: a$ for two indices $i' < i$, then $a = a_{i'} \in u_{i'}$ and $a = a_i \in v_i$; hence $a \in u_{i'} \cap v_i$, a contradiction. Therefore the list $(a_i)_{i \in I}$ contains $|I|$ distinct elements of U. Hence, $|I| \leq |U|$. □

Let $V \subseteq \mathcal{P}(U)$ be a set of points in the lattice of subsets of U. For $u \in V$, the part of V below u is $V_u := \{u' \in V \mid u' \subseteq u\}$; the *height* $h_V(u)$ of u in V is the length of the longest chain $\emptyset \neq u_1 \subsetneq \cdots \subsetneq u_k$ in V_u. For $\zeta : V \to V$, we say ζ is *monotone* if it respects inclusion: $u' \subseteq u \implies \zeta(u') \subseteq \zeta(u)$; we say ζ is an *automorphism* if its restriction to V_u is a bijection from V_u to $V_{\zeta(u)}$, for all u. Clearly, every automorphism respects heights: $h_V(u) = h_V(\zeta(u))$, for all u. By ζ^t we mean the t-fold composition of ζ with itself; if $t = 0$, this is the identity.

23. LEMMA. *Suppose $\zeta : V \to V$, where $V \subseteq \mathcal{P}(U)$ is a* finite *set of points from the lattice of a set U. If ζ is* injective *and* monotone*, then it is an automorphism.*

PROOF. Pick any $u \in V$, set $v := \zeta(u)$, and let ζ_u be the restriction of ζ to V_u. We will show ζ_u is a bijection from V_u to V_v. Since ζ is monotone, ζ_u has all its values in V_v: $u' \in V_u \implies u' \subseteq u \implies \zeta(u') \subseteq \zeta(u) \implies \zeta_u(u') \in V_v$. Since ζ is injective, so is ζ_u. So, ζ_u is an injection from V_u to V_v. To show that it is a bijection, it is sufficient to show that V_v does not have more elements than V_u.

Since ζ is injective and V is finite, ζ is a permutation of V. Hence, for some $t \geq 1$, ζ^t is the identity. Let $\zeta' := \zeta^{t-1}$. Since ζ is injective and monotone, ζ' is also injective and monotone. Moreover, $u = \zeta^t(u) = \zeta^{t-1}(\zeta(u)) = \zeta'(v)$. Now the same argument as in the previous paragraph shows that the restriction ζ'_v of ζ' to V_v is an injection from V_v to V_u. Consequently, $|V_v| \leq |V_u|$. □

Let $R \subseteq U^2$ be a binary relation. We write $R(\cdot)$ for the mapping of each $u \subseteq U$ to the set $R(u) := \{b \in U \mid (\exists a \in u)(aRb)\}$ of all elements related to elements of u; we usually write $R(a)$ instead of $R(\{a\})$. Clearly, $R(\cdot)$ is a monotone function. If $R' \subseteq U^2$ is also a binary relation, we write $R' \circ R$ for the composition: $a(R' \circ R)b \iff (\exists c \in U)(aR'c \ \& \ cRb)$. Clearly, $(R' \circ R)(u) = R(R'(u))$, for all u.

A total order $<$ on $\mathcal{P}(U)^2$ is *nice* if each pair "escapes" from every strictly smaller pair in at least one component: $(u', v') < (u, v) \implies u' \not\supseteq u \vee v' \not\supseteq v$. As shown in the next lemma, such orders exist no matter what finite set we pick as U.

24. LEMMA. *For every finite set U, there exist nice orders on $\mathcal{P}(U)^2$.*

PROOF. Given any finite U, we simply exhibit such an order. Specifically, we declare $(u', v') < (u, v)$ whenever $|u'| + |v'| < |u| + |v|$, and pick any order whenever $|u'| + |v'| = |u| + |v|$. (Note that we can make these comparisons between the sizes of the sets, as they are all finite.) The result is indeed a nice order. To prove this, we pick arbitrary $u, v \subseteq U$ and show the contrapositive of the necessary condition:

$$u' \supseteq u \ \& \ v' \supseteq v \implies (u', v') \not< (u, v).$$

So, suppose $u' \supseteq u \ \& \ v' \supseteq v$. If $u' \neq u$ or $v' \neq v$, then one of the two inclusions is strict, hence $|u'| + |v'| > |u| + |v|$, and thus $(u', v') > (u, v)$. Otherwise, $u' = u$ and $v' = v$, and thus $(u', v') = (u, v)$. In both cases, $(u', v') \not< (u, v)$, as needed. □

3.1-II. *Strings over Σ_n.* Recall the alphabet Σ_n over which liveness is defined. In this section, it will be convenient to have a concise way of describing how the edges of a string over Σ_n connect the vertices of its outer columns. So, given any

$z \in \Sigma_n^*$, we say that z has *connectivity* $\xi \subseteq [n]^2$ if the following holds: $(a, b) \in \xi$ iff z contains a path from the a-th node of its leftmost column to the b-th node of its rightmost column. For example, the connectivity of the string on page 17 is $\{(3,1),(3,4)\}$; the connectivity of the empty string ϵ is the identity relation $\{(a,a) \mid a \in [n]\}$; and the connectivity of any single symbol is the symbol itself. The set of all strings of connectivity ξ is written as $B_{n,\xi}$. In this notation,

$$\overline{B_n} = B_{n,\emptyset}.$$

In other words, the dead strings are exactly those with connectivity $\emptyset$. Any other connectivity implies that a string is live.

3.2. Sweeping automata. One way to define sweeping 2NFAs is to start with our standard definition for 2NFAs (cf. Chapter 1, Section 2) and simply impose the restriction that the transition function is such that the direction of the input head never changes strictly inside the input, for all inputs and all branches of the corresponding nondeterministic computations. Note that, with a definition of this kind, it becomes meaningful to ask whether a given 2NFA is sweeping or not.

Our approach will be different. We will give an entirely new definition, with the restriction about the direction of motion built-in. The best way to explain what this means to give the definition right away. So, here it is. As usual, we start with the deterministic version and leave the straightforward generalization to nondeterminism for later.

3.2-I. *The deterministic case.* By a *sweeping deterministic finite automaton* (SDFA) *over the states of a set Q and the symbols of an alphabet Σ* we mean any triple $M = (s, \delta, f)$, where δ is the *transition function*, partially mapping $Q \times \Sigma_e$ to Q, and $s, f \in Q$ are the *start* and the *final* states. An input $w \in \Sigma^*$ is presented to M surrounded by the end-markers, as $\vdash w \dashv$. The computation starts at s and on the symbol to the right of $\vdash$, heading rightward. 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 $\vdash$ or when the current symbol is $\dashv$ and the next state is not f, in which two cases the next position is the adjacent one in the opposite direction. Note that the computation can either loop, or hang, or move past $\dashv$ into f. In this last case we say that *M accepts w.*

We stress that the values of the transition function do not contain any direction information. In contrast, this information is derived implicitly from the assumption that the automaton is sweeping. This greatly simplifies the setting and helps us stay closer to the combinatorial essence of sweeping automata, avoiding the distraction caused by irrelevant inherited features. Moreover, *this definitional shift does not invalidate our conclusions* in this section. Specifically, it is not hard to verify that the size of a sweeping 2NFA under the new definition is linearly related to the size of a smallest equivalent sweeping 2NFA under the standard definition, and vice versa. Hence, any exponential lower bound under either definition implies an exponential lower bound under the other one. Of course, for exact trade-offs, the choice of definition will matter—but we will not be worrying about them here.

The simplified definition allows for a simplified notion of computation, as well. In particular, for any $z \in \Sigma^*$ and $p \in Q$, 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 last provision fails. If $m = |z|+1$, the computation *exits into* q_m; otherwise, $1 \le m \le |z|$ and the computation *hangs at* q_m. The *right computation of* M *from* p *on* z is defined symmetrically, as the sequence $\text{RCOMP}_{M,p}(z) = (q_t)_{1\le t\le m}$ with $q_{t+1} = \delta(q_t, z_{|z|+1-t})$.

3.2-II. *The nondeterministic case.* If M is allowed more than one next move at each step, we say that it is *nondeterministic* (SNFA). Formally, this means that δ *totally* maps $Q \times \Sigma_e$ to the *powerset* of Q and implies that, on input any $w \in \Sigma^*$, M exhibits a *set* of computations on $\vdash w \dashv$. If at least one of them moves past $\dashv$ into f, then M accepts w. Similarly, $\text{LCOMP}_{M,p}(z)$ and $\text{RCOMP}_{M,p}(z)$ are now *sets* of computations.

We also introduce a notion to describe how the states of M connect via left and right computations on some string. For *left* computations on some $z \in \Sigma^*$, we encode these connections into a binary relation $\text{LVIEW}_M(z) \subseteq Q^2$, which we refer to as the *left behavior of* M *on* z, defined as:

$$(p,q) \in \text{LVIEW}_M(z) \iff \big(\exists c \in \text{LCOMP}_{M,p}(z)\big)(c \text{ exits into } q),$$

Then, for any $u \subseteq Q$, the set $\text{LVIEW}_M(z)(u)$ of states reachable from within u via left computations on z is the *left view of* u *on* z. The *right behavior* $\text{RVIEW}_M(z)$ *of* M *on* z and the *right view* $\text{RVIEW}_M(z)(u)$ *of* u *on* z are defined symmetrically.

Note that, if $|z| = 1$, the automaton has the same behavior in both directions:

25. LEMMA. $|z| = 1 \implies \text{LVIEW}_M(z) = \text{RVIEW}_M(z) = \{(p,q) \mid \delta(p,z) \ni q\}$.

We also note that, if extending the string z does not cause a view to include any new states, then this remains true on all identical further extensions:

26. LEMMA. *The following implications are true, for all* $t \ge 1$:
- $\text{LVIEW}_M(z)(u) \supseteq \text{LVIEW}_M(z\tilde{z})(u) \implies \text{LVIEW}_M(z)(u) \supseteq \text{LVIEW}_M(z\tilde{z}^t)(u)$,
- $\text{RVIEW}_M(z)(u) \supseteq \text{RVIEW}_M(\tilde{z}z)(u) \implies \text{RVIEW}_M(z)(u) \supseteq \text{RVIEW}_M(\tilde{z}^t z)(u)$.

PROOF. For the first implication, suppose that the set $\text{LVIEW}_M(z)(u)$ contains the set $\text{LVIEW}_M(z\tilde{z})(u)$. To show that it also contains $\text{LVIEW}_M(z\tilde{z}^t)(u)$, we use induction on t. The case $t = 1$ is the assumption itself. For $t \ge 1$, we calculate:

$$\begin{aligned}
\text{LVIEW}_M(z\tilde{z}^{t+1})(u) &= \text{LVIEW}_M(\tilde{z})\big(\text{LVIEW}_M(z\tilde{z}^t)(u)\big) && \text{(algebra of relations)}\\
&\subseteq \text{LVIEW}_M(\tilde{z})\big(\text{LVIEW}_M(z)(u)\big) && \text{(inductive hypothesis)}\\
&= \text{LVIEW}_M(z\tilde{z})(u) && \text{(algebra of relations)}\\
&\subseteq \text{LVIEW}_M(z)(u). && \text{(original assumption)}
\end{aligned}$$

In the 1st step, note that $\text{LVIEW}_M(z\tilde{z}^{t+1}) = \text{LVIEW}_M(z\tilde{z}^t) \circ \text{LVIEW}_M(\tilde{z})$. In the 2nd step, along with the inductive hypothesis, we use the fact that $\text{LVIEW}_M(\tilde{z})(\cdot)$ is monotone. The 3rd step uses the fact that $\text{LVIEW}_M(z\tilde{z}) = \text{LVIEW}_M(z) \circ \text{LVIEW}_M(\tilde{z})$.

The second implication can be proved symmetrically. □

3.3. Proof outline. We are now ready to present an outline of our argument. As explained in the introduction, to prove that small SNFAs are not closed under complement (SN $\neq$ coSN), it is enough to prove the following.

27. THEOREM. *Every* SNFA *that recognizes* $B_{n,\emptyset}$ *has* $2^{\Omega(n)}$ *states.*

The remainder of Section 3 is a proof is this fact. We fix n and an SNFA $M = (s, \delta, f)$ over a set of k states Q that solves $B_{n,\emptyset}$, and we prove that $k = 2^{\Omega(n)}$.

The proof is based on Lemma 22. We build two sequences $(X_\iota)_{\iota \in \mathcal{I}}$ and $(Y_\iota)_{\iota \in \mathcal{I}}$ that are related as in the lemma. The indices are all pairs of non-empty subsets of $[n]$, the universe is all sets of 1 or 2 steps of M:[4]

$$\mathcal{I} := \{(\alpha, \beta) \mid \emptyset \neq \alpha, \beta \subseteq [n]\} \qquad \mathcal{E} := \{\{e', e\} \mid e', e \in Q^2\},$$

and the total order $<$ is the restriction on $\mathcal{I}$ of some *nice* order on $\mathcal{P}([n])^2$. If we indeed construct these sequences, then the lemma says that $|\mathcal{I}| \leq |\mathcal{E}|$, therefore

$$(2^n - 1)^2 \leq k^2 + \tbinom{k^2}{2},$$

which implies $k = 2^{\Omega(n)}$, as required. For the remainder, we fix $\mathcal{I}$ and $\mathcal{E}$ as here.

Note that from now on some subscripts in our notation are redundant. We thus drop them: e.g., $B_{n,\emptyset}$ and $\text{LVIEW}_M(z)(u)$ become $B_\emptyset$ and $\text{LVIEW}(z)(u)$.

Before moving on, let us also quickly prove a fact that will be useful later: In order to accept a dead string but reject a live one, M must produce on the dead string a single-state view that "escapes" the corresponding view on the live string.

28. LEMMA. *If z' is live and z is dead, then at least one of the following holds:*
- $\text{LVIEW}(z')(p) \not\supseteq \text{LVIEW}(z)(p)$ *for some* $p \in Q$.
- $\text{RVIEW}(z')(p) \not\supseteq \text{RVIEW}(z)(p)$ *for some* $p \in Q$.

PROOF. Towards a contradiction, suppose that $\text{LVIEW}(z')(p) \supseteq \text{LVIEW}(z)(p)$ and $\text{RVIEW}(z')(p) \supseteq \text{RVIEW}(z)(p)$, for all p. Pick any *accepting* computation c of M on z and break it into its *traversals* $c_1, \dots, c_m$, in the natural way: for all $j < m$,
- c_j starts at a state p_j next to $\vdash$ and ends at a state q_j on $\dashv$, if j is odd;
- c_j starts at a state p_j next to $\dashv$ and ends at a state q_j on $\vdash$, if j is even;

$p_1 = s$ and p_{j+1} is in $\delta(q_j, \dashv)$ or $\delta(q_j, \vdash)$, depending on whether j is odd or even, respectively; m is even and the last fragment is not really a traversal, but simply $c_m = (f)$. Then, for each $j < m$, we know that
- q_j is in $\text{LVIEW}(z)(p_j)$ and thus also in $\text{LVIEW}(z')(p_j)$, if j is odd,
- q_j is in $\text{RVIEW}(z)(p_j)$ and thus also in $\text{RVIEW}(z')(p_j)$, if j is even.

Hence, in both cases, some computation c'_j of M on z' starts and ends identically to c_j. If we also set $c'_m := (f)$ and concatenate the computations $c'_1, \dots, c'_m$, we end up with a computation c' of M on z' which is also accepting. So, M accepts the live string z', a contradiction. □

3.4. Hard inputs and the two sequences. In this section, we will construct a set of inputs that collectively force M to use exponentially many states. Similarly to what we did for moles, here we will again need to start with strings that are long and rich enough to strain the ability of M to process their information, and then use those strings as building blocks for constructing the hard inputs. Once again, we call these strings *generic*, and we base their construction on the same general idea of [**55**].

[4]A *step* of M is any $e \in Q^2$. Also, note that $\{e', e\}$ represents a singleton when $e' = e$.

3.4-I. *Generic strings.* Consider any $y \in \Sigma^*$ and the set of views produced via left computations on it:

$$\text{LVIEWS}(y) := \{\text{LVIEW}(y)(u) \mid u \subseteq Q\},$$

i.e., the range of the function $\text{LVIEW}(y)(\cdot)$. How does this set change when we extend y into a longer string yz?

Let $\text{LMAP}(y,z)$ be the function that for every left view produced on y returns its left view on z—namely, $\text{LMAP}(y,z)$ is the restriction of $\text{LVIEW}(z)(\cdot)$ to $\text{LVIEWS}(y)$.

It is easy to verify that the values of $\text{LMAP}(y,z)$ are all inside $\text{LVIEWS}(yz)$. Indeed, consider any u in the range of $\text{LMAP}(y,z)$. Then some u' in the domain of $\text{LMAP}(y,z)$ is such that $\text{LMAP}(y,z)(u') = u$. Since this domain is $\text{LVIEWS}(y)$, some $u'' \subseteq Q$ is such that $\text{LVIEW}(y)(u'') = u'$. Then,

$$\begin{aligned} u = \text{LMAP}(y,z)(u') &= \text{LMAP}(y,z)\big(\text{LVIEW}(y)(u'')\big) \\ &= \text{LVIEW}(z)\big(\text{LVIEW}(y)(u'')\big) \\ &= \text{LVIEW}(yz)(u''), \end{aligned}$$

so that u is indeed in $\text{LVIEWS}(yz)$. (Note that in the last equality we used the fact that $\text{LVIEW}(yz) = \text{LVIEW}(y) \circ \text{LVIEW}(z)$.)

Moreover, the values of $\text{LMAP}(y,z)$ cover $\text{LVIEWS}(yz)$. Indeed, consider any $u \in \text{LVIEWS}(yz)$. Then there exists $u'' \subseteq Q$ is such that $\text{LVIEW}(yz)(u'') = u$. Letting $u' := \text{LVIEW}(y)(u'')$, we see that $u' \in \text{LVIEWS}(y)$. Therefore, u' is in the domain of $\text{LMAP}(y,z)$. Moreover,

$$\begin{aligned} \text{LMAP}(y,z)(u') &= \text{LVIEW}(z)(u') \\ &= \text{LVIEW}(z)\big(\text{LVIEW}(y)(u'')\big) \\ &= \text{LVIEW}(yz)(u'') = u, \end{aligned}$$

so that u is indeed in the range of $\text{LMAP}(y,z)$. (Again, in the last equality we used the fact that $\text{LVIEW}(yz) = \text{LVIEW}(y) \circ \text{LVIEW}(z)$.)

Overall, $\text{LMAP}(y,z)$ is a *surjection* from $\text{LVIEWS}(y)$ to $\text{LVIEWS}(yz)$. This immediately implies that $|\text{LVIEWS}(y)| \geq |\text{LVIEWS}(yz)|$.

The next fact encodes this conclusion, along with the obvious remark that $\text{LMAP}(y,z)$ is monotone. It also shows the symmetric facts, for left extensions and right views. The set $\text{RVIEWS}(y)$ consists of all views produced on y via right computations, and $\text{RMAP}(z,y)$ is the restriction of $\text{RVIEW}(z)(\cdot)$ on $\text{RVIEWS}(y)$.

29. LEMMA. *For all y,z: $\text{LMAP}(y,z)$ is a monotone surjection of $\text{LVIEWS}(y)$ onto $\text{LVIEWS}(yz)$, so $|\text{LVIEWS}(y)| \geq |\text{LVIEWS}(yz)|$; similarly, $\text{RMAP}(z,y)$ is a monotone surjection of $\text{RVIEWS}(y)$ onto $\text{RVIEWS}(zy)$, so $|\text{RVIEWS}(y)| \geq |\text{RVIEWS}(zy)|$.*

Now suppose y belongs to an infinitely right-extensible property $P \subseteq \Sigma^*$. What happens to the size of $\text{LVIEWS}(y)$ if we keep extending y into $yz, yzz', \ldots$ inside P? Although there are infinitely many extensions, the size of the set can decrease only finitely many times. So, at some point it must stop changing. When this happens, we have arrived at a very useful tool. We define it as follows.

30. DEFINITION. Let $P \subseteq \Sigma^*$. A string y is L-*generic over P* if $y \in P$ and

$$(\forall yz \in P)\big[|\text{LVIEWS}(y)| = |\text{LVIEWS}(yz)|\big].$$

An R-*generic* string over P is defined similarly, with left-extensions and $\text{RVIEWS}(\cdot)$. A string that is both L-generic and R-generic over P is called *generic*.

31. LEMMA. *Let $P \subseteq \Sigma^*$. If P is non-empty and infinitely right-extensible (resp., left-extensible), then there exist* L*-generic (resp.,* R*-generic) strings over P. If y_{L} is* L*-generic and y_{R} is* R*-generic, then every string $y_{\text{L}} x y_{\text{R}} \in P$ is generic.*

PROOF. For the last claim, we just note that all right-extensions of an L-generic string inside P are also L-generic, and the same is true in the other direction. □

Intuitively, from the perspective of M, a generic string is among the *richest* inputs that have property P, in the sense that it exhibits a greatest subset of the "features" that M is "prepared to pay attention to". This makes generic strings useful in building hard inputs, as described in the Lemma 33 below and in Section 3.4-II.

32. LEMMA. *For any $y, z \in \Sigma^*$: $\text{LVIEWS}(yz) \subseteq \text{LVIEWS}(z)$ and $\text{RVIEWS}(zy) \subseteq \text{RVIEWS}(z)$.*

PROOF. By Lemma 29, $\text{LVIEWS}(yz)$ is the range of $\text{LMAP}(y, z)$, which is a restriction of $\text{LVIEW}(z)(\cdot)$. So, the first containment follows. The argument in the other direction is similar. □

33. LEMMA. *Let y be generic over $P \subseteq \Sigma^*$, and let $x \in \Sigma^*$. If $yxy \in P$, then*
- *$\text{LMAP}(y, xy)$ is an automorphism on $\text{LVIEWS}(y)$, and*
- *$\text{RMAP}(yx, y)$ is an automorphism on $\text{RVIEWS}(y)$.*

PROOF. Suppose $yxy \in P$. Then $|\text{LVIEWS}(y)| = |\text{LVIEWS}(yxy)|$ (since y is generic) and $\text{LVIEWS}(yxy) \subseteq \text{LVIEWS}(y)$ (by Lemma 32). Therefore, we know that $\text{LVIEWS}(y) = \text{LVIEWS}(yxy)$. By this and Lemma 29, we conclude $\text{LMAP}(y, xy)$ surjects $\text{LVIEWS}(y)$ onto itself, which is possible only if it is injective. Since $\text{LMAP}(y, xy)$ is also monotone, Lemma 23 implies it is an automorphism.

The fact about $\text{RMAP}(yx, y)$ is proved similarly. □

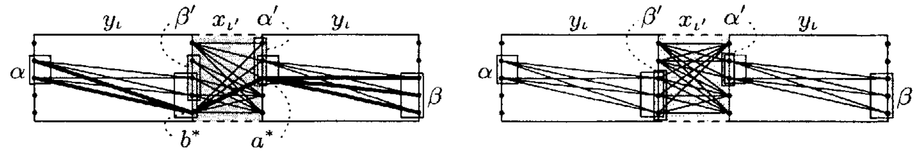
3.4-II. *Constructing the hard inputs.* Fix $\iota = (\alpha, \beta) \in \mathcal{I}$ and let $P_\iota := B_{\alpha \times \beta}$ be the property of connecting exactly every leftmost node in α to every rightmost node in β. Easily, P_ι is non-empty and infinitely extensible in both directions. Therefore, an L-generic string y_{L} and an R-generic string y_{R} exist (Lemma 31). Then, for $\eta := [n]^2$ the complete symbol, we easily see that $y_{\text{L}} \eta y_{\text{R}} \in P_\iota$, too. Hence, this string is generic over P_ι (Lemma 31). We define $y_\iota := y_{\text{L}} \eta y_{\text{R}}$. We also define the symbol $x_\iota := \overline{\beta \times \alpha}$.

34. LEMMA. *The two sequences $(y_\iota)_{\iota \in \mathcal{I}}$ and $(x_\iota)_{\iota \in \mathcal{I}}$ are such that:*

$$\iota' < \iota \implies y_\iota x_{\iota'} y_\iota \in P_\iota \qquad \textit{and} \qquad \iota' = \iota \implies y_\iota x_{\iota'} y_\iota \in B_\emptyset.$$

for all $\iota', \iota \in \mathcal{I}$.

PROOF. Fix $\iota' = (\alpha', \beta')$ and $\iota = (\alpha, \beta)$ and let $z := y_\iota x_{\iota'} y_\iota$. Note that the connectivities of y_ι and $x_{\iota'}$ are respectively $\xi := \alpha \times \beta$ and $\xi' := \overline{\beta' \times \alpha'}$.



If $\iota' < \iota$ (on the left), then $\alpha' \not\supseteq \alpha$ or $\beta' \not\supseteq \beta$ (since $<$ is nice). Suppose $\beta' \not\supseteq \beta$ (if $\alpha' \not\supseteq \alpha$, use a similar argument) and fix any $b^* \in \beta \setminus \beta'$ and any $a^* \in \alpha$. For any $a, b \in [n]$, consider the a-th leftmost and b-th rightmost nodes of z. If $a \notin \alpha$ or $b \notin \beta$, then the two nodes do not connect in z, since neither can "see through" y_ι. If $a \in \alpha$ and $b \in \beta$, then $(a, b^*) \in \xi$ and $(b^*, a^*) \in \xi'$ and $(a^*, b) \in \xi$, so the two nodes connect via a path of the form $a \rightsquigarrow b^* \rightarrow a^* \rightsquigarrow b$. Overall, $z \in P_\iota$.

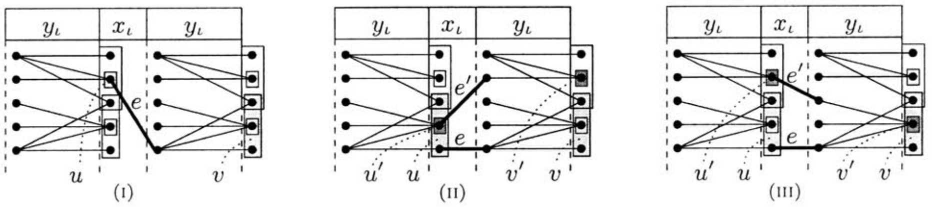
If $\iota' = \iota$ (on the right), then $\xi' = \overline{\beta \times \alpha}$. Suppose $z \notin B_\emptyset$. Then some path in z connects the leftmost to the rightmost column. Suppose it is of the form $a \rightsquigarrow b^* \rightarrow a^* \rightsquigarrow b$. Then $b^* \in \beta$ and $(b^*, a^*) \in \xi'$ and $a^* \in \alpha$, a contradiction. □

3.4-III. *Constructing the two sequences.* Suppose $\iota' < \iota$. Since the extension $y_\iota x_{\iota'} y_\iota$ of y_ι preserves P_ι (Lemma 34), each of $\text{LMAP}(y_\iota, x_{\iota'} y_\iota)$ and $\text{RMAP}(y_\iota x_{\iota'}, y_\iota)$ is an automorphism (Lemma 33). Put another way, the interaction between the steps of M on $x_{\iota'}$ and its two behaviors on y_ι is such that these two mappings are automorphisms. Put formally, both

- the restriction of $\big(E_{\iota'} \circ \text{LVIEW}(y_\iota)\big)(\cdot)$ on $\text{LVIEWS}(y_\iota)$ and
- the restriction of $\big(E_{\iota'} \circ \text{RVIEW}(y_\iota)\big)(\cdot)$ on $\text{RVIEWS}(y_\iota)$

are automorphisms, for $E_{\iota'} := \{(p, q) \mid \delta(p, x_{\iota'}) \ni q\} = \text{LVIEW}(x_{\iota'}) = \text{RVIEW}(x_{\iota'})$.

What if $\iota' = \iota$? What is then the status of $\text{LMAP}(y_\iota, x_\iota y_\iota)$ and $\text{RMAP}(y_\iota x_\iota, y_\iota)$? We can show that, since $y_\iota x_\iota y_\iota$ is dead (Lemma 34), we cannot have both functions be automorphisms.[5] However, something stronger is also true: *we can even convince ourselves that one of the functions is not an automorphism by pointing at only 1 or 2 of the steps of M on x_ι.* The next figure shows three examples of this. In each, we sketch the left behavior of M on y_ι and all single-state views.



Example I shows only 1 of the steps of M on x_ι, say $e = (p, q)$ —many more may be included in E_ι. Is $\text{LMAP}(y_\iota, x_\iota y_\iota)$ an automorphism? Normally, we would need to know the entire E_ι to answer this question. Yet, in this case e is enough to answer no. To see why, note that the view v of q on y_ι has height 2, while one of the views that contain p is u, of height 1. Irrespective of the rest of E_ι, $\text{LMAP}(y_\iota, x_\iota y_\iota)$ will map u to a view that contains v and thus has height 2 or more. So, it does not respect heights, which implies it is not an automorphism.

Example II shows 2 of the steps in E_ι, say $e' = (p', q')$ and $e = (p, q)$. Is $\text{LMAP}(y_\iota, x_\iota y_\iota)$ an automorphism? Observe that neither step alone can force a negative answer: the view v' of q' on y_ι has height 1, as does the lowest view u' containing p'; similarly for e, u, v, and height 2. Hence, individually each of e' and e may very well participate in sets of steps that induce automorphisms. Yet, they cannot belong to the same such set. To see why, suppose they do. Since $u' \subseteq u$,

[5]If they were, they would be bijections (because each of $\text{LVIEWS}(y_\iota)$ and $\text{RVIEWS}(y_\iota)$ has a maximum). Hence, M would not be able to distinguish between the live y_ι and the dead $y_\iota(x_\iota y_\iota)^t$, for t any exponent that turns both bijections into identities. (Note that this is true even for the n-state SNFA that solves liveness. Therefore, this observation alone can give rise to no interesting lower bound for k.)

the image of u would be $v' \cup v$ or a superset. Since $v' \nsubseteq v$, the height of that image would be greater than the height of v, and thus greater than the height of u, violating the respect to heights.

Example III also shows 2 of the steps in E_ι, say $e' = (p', q')$ and $e = (p, q)$, neither of which can disqualify $\textsc{lmap}(y_\iota, x_\iota y_\iota)$ from being an automorphism. Yet, together they can. To see why, suppose both steps participate in the same automorphism. Then the image of u' must be exactly v': otherwise, it would be some strict superset of v', of height 2 or more, disrespecting the height of u'. On the other hand, u must map to a set that contains v, and thus also $v' \subseteq v$. Hence, v' must be the exact image of some $u^* \subseteq u$. But then both u^* and u' map to v', when $u^* \neq u'$ (since $u' \nsubseteq u$), a contradiction to the map being injective.

In short, each step in E_ι severely restricts the form of $\textsc{lmap}(y_\iota, x_\iota y_\iota)$ and $\textsc{rmap}(y_\iota x_\iota, y_\iota)$. And, either individually or in pairs, some steps can be so restrictive that they cannot be part of any set of steps that induces an automorphism in both directions. To describe this formally, we introduce the next definition.

35. DEFINITION. A set of steps $E \subseteq Q^2$ is *compatible* with y_ι if there exists a set $\hat{E}$ such that $E \subseteq \hat{E} \subseteq Q^2$ and the following are both automorphisms:

- the restriction of $\big(\hat{E} \circ \textsc{lview}(y_\iota)\big)(\cdot)$ on $\textsc{lviews}(y_\iota)$, and
- the restriction of $\big(\hat{E} \circ \textsc{rview}(y_\iota)\big)(\cdot)$ on $\textsc{rviews}(y_\iota)$.

E.g., $\{e\}$ in Example I and $\{e', e\}$ in Examples II,III are incompatible with y_ι.

We are now ready to define the sequences promised in Section 3.3. For each $\iota \in \mathcal{I}$, we let X_ι consist of all sets of 1 or 2 steps of M on x_ι, and Y_ι consist of all sets of 1 or 2 steps of M that are incompatible with y_ι:

$$X_\iota := \{E \in \mathcal{E} \mid E \subseteq E_\iota\}, \qquad Y_\iota := \{E \in \mathcal{E} \mid E \text{ is incompatible with } y_\iota\}.$$

We need, of course, to show that the sequences relate as in Lemma 22.

The case $\iota' < \iota$ is easy. Each $E \in X_{\iota'}$ can be extended to the set of all steps of M on $x_{\iota'}$ (i.e., $\hat{E} := E_{\iota'}$), which does induce automorphisms, so $X_{\iota'} \cap Y_\iota = \emptyset$.

The case $\iota' = \iota$ is harder. We analyze it in the next section.

3.5. The main argument. Suppose $\iota' = \iota$. Our goal is to exhibit a singleton or two-set $E \subseteq E_\iota$ that is incompatible with y_ι. First, some preparation.

3.5-I. *The witness.* Consider the strings $y_\iota(x_\iota y_\iota)^t = (y_\iota x_\iota)^t y_\iota$, for all $t \geq 1$. Since $y_\iota x_\iota y_\iota$ is dead, the same is true of all these strings. Since y_ι is live, Lemma 28 says that for all $t \geq 1$:

- $\textsc{lview}(y_\iota)(p) \nsupseteq \textsc{lview}\big(y_\iota(x_\iota y_\iota)^t\big)(p)$ for some $p \in Q$, or
- $\textsc{rview}(y_\iota)(p) \nsupseteq \textsc{rview}\big((y_\iota x_\iota)^t y_\iota\big)(p)$ for some $p \in Q$.

Namely, in order to accept the extensions $y_\iota(x_\iota y_\iota)^t = (y_\iota x_\iota)^t y_\iota$ but reject the original y_ι, M must exhibit on each of them a single-state view that 'escapes' its counterpart on the original. In a sense, among all $2k$ single-state views on each extension, the escaping one is a 'witness' for the fact that the extension is accepted, and Lemma 28 says that *every extension has a witness.* Of course, this allows for the possibility that different extensions may have different witnesses. However, we can actually find the same witness for all extensions:

36. LEMMA. *At least one of the following is true:*

- $\textsc{lview}(y_\iota)(p) \nsupseteq \textsc{lview}\big(y_\iota(x_\iota y_\iota)^t\big)(p)$ *for some* $p \in Q$ *and all* $t \geq 1$.
- $\textsc{rview}(y_\iota)(p) \nsupseteq \textsc{rview}\big((y_\iota x_\iota)^t y_\iota\big)(p)$ *for some* $p \in Q$ *and all* $t \geq 1$.

PROOF. Suppose neither is true. Then each of the $2k$ single-state views has an extension on which it fails to escape from its counterpart on y_ι. Namely, every p has some $t_{p,\text{L}} \geq 1$ such that $\text{LVIEW}(y_\iota)(p) \supseteq \text{LVIEW}\big(y_\iota(x_\iota y_\iota)^{t_{p,\text{L}}}\big)(p)$ and some $t_{p,\text{R}} \geq 1$ such that $\text{RVIEW}(y_\iota)(p) \supseteq \text{RVIEW}\big((y_\iota x_\iota)^{t_{p,\text{R}}} y_\iota\big)(p)$. Consider the exponent

$$t^* := \left(\prod\nolimits_{p\in Q} t_{p,\text{L}}\right) \cdot \left(\prod\nolimits_{p\in Q} t_{p,\text{R}}\right)$$

and the extension $z := y_\iota(x_\iota y_\iota)^{t^*} = (y_\iota x_\iota)^{t^*} y_\iota$. Then each p has some $t \geq 1$ such that $z = y_\iota((x_\iota y_\iota)^{t_{p,\text{L}}})^t$, and thus Lemma 26 implies $\text{LVIEW}(y_\iota)(p) \supseteq \text{LVIEW}(z)(p)$; similarly, $\text{RVIEW}(y_\iota)(p) \supseteq \text{RVIEW}(z)(p)$. Overall, all single-state views on z fall within their counterparts on y_ι, contradicting Lemma 28. $\square$

We fix p to be a witness as in Lemma 36. We assume p is of the first type, involving left views (otherwise, a symmetric argument applies). Moreover, among all witnesses of this type, we select p so as to minimize the height of $\text{LVIEW}(y_\iota)(p)$ in $\text{LVIEWS}(y_\iota)$. We let $V := \text{LVIEWS}(y_\iota)$, $h := h_V$, and $v_0 := \text{LVIEW}(y_\iota)(p)$.

By the selection of p, no $\tilde{p}$ with $\text{LVIEW}(y_\iota)(\tilde{p}) \subsetneq v_0$ can be a witness of the first type. Hence, for every such $\tilde{p}$ there is some $\tilde{t} \geq 1$ such that $\text{LVIEW}(y_\iota)(\tilde{p}) \supseteq \text{LVIEW}\big(y_\iota(x_\iota y_\iota)^{\tilde{t}}\big)(\tilde{p})$. We fix t^* to be the product of all such $\tilde{t}$. Then:

37. LEMMA. *For all such $\tilde{p}$ and $\lambda \geq 1$:* $\text{LVIEW}(y_\iota)(\tilde{p}) \supseteq \text{LVIEW}(y_\iota(x_\iota y_\iota)^{\lambda t^*})(\tilde{p})$.

PROOF. Fix such a $\tilde{p}$ and the $\tilde{t}$ for which $\text{LVIEW}(y_\iota)(\tilde{p}) \supseteq \text{LVIEW}\big(y_\iota(x_\iota y_\iota)^{\tilde{t}}\big)(\tilde{p})$. Fix any $\lambda \geq 1$. Then λt^* is a multiple of $\tilde{t}$ and Lemma 26 applies. $\square$

3.5-II. *Escape computations.* For all $t \geq 1$, collect into a set $\mathcal{C}_t$ all computations $c \in \text{LCOMP}_p(y_\iota(x_\iota y_\iota)^t)$ that exit into some state $q \notin v_0$. These are the *escape computations* for p on the t-th extension. We also define $\mathcal{C} := \cup_{t\geq 1}\mathcal{C}_t$.

Let us see how an escape computation looks like. Pick any $c \in \mathcal{C}$ (Figure 17a), say on the t-th extension, exiting into q. Let $e_1, \dots, e_t$ be the steps of c on x_ι, where $e_j = (p_j, q_j) \in E_\iota$. These are the *critical steps* along c. Let $v_j := \text{LVIEW}(y_\iota)(q_j)$ be the view of the right end-point of e_j. Along with v_0, these views form the list $v_0, v_1, \dots, v_t$ of the *major views* along c. Clearly, each of them contains the left end-point of the following critical step: $v_{j-1} \ni p_j$ (similarly, $v_t \ni q$). So, for each e_j there exist views $u \in V$ that contain its left end-point and are contained in the preceding major view: $v_{j-1} \supseteq u \ni p_j$ (similarly, $v_t \supseteq u \ni q$). Among them, let u_{j-1} be one of minimum height in V (select u_t similarly). Then the list $u_0, \dots, u_{t-1}, u_t$ are the *minor views* along c. This concludes our description of how escape computations look like.

3.5-III. *The incompatible set.* We are now ready to find the incompatible set E that we are looking for. We will find its one or two steps among the critical steps of escape computations. We distinguish two cases.

Case 1: Some $c \in \mathcal{C}$ contains some critical step e such that the singleton $\{e\}$ is incompatible with y_ι. Then we can select $E := \{e\}$, and we are done.

Case 2: For all $c \in \mathcal{C}$ and all critical steps e in c, the singleton $\{e\}$ is compatible with y_ι. In this case, we will find an incompatible two-set.

Steepness. First of all, every $c \in \mathcal{C}$ (say with t, e_j, v_j, u_j as above) has every major view at least as high as the next minor one ($h(v_j) \geq h(u_j)$, since $v_j \supseteq u_j$) and every minor view at least as high as the next major one ($h(u_j) \geq h(v_{j+1})$, otherwise $\{e_{j+1}\}$ would be incompatible, as in Example I). Hence, every $c \in \mathcal{C}$ has

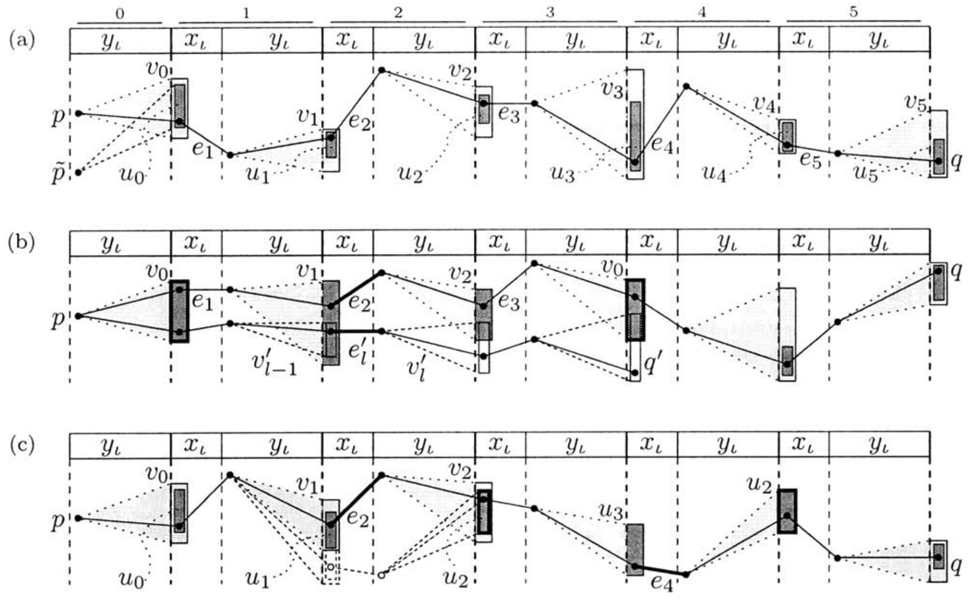


FIGURE 17. (a) An escape computation $c \in \mathcal{C}_5$, exiting into q. (b) An example of Case 2A, for $j = 3$ and $l = 2$; in dashes, the new computation $c' \in \mathcal{C}_j$. (c) An example of Case 2B, for $j' = 2$ and $j = 4$; in dashes, the hypothetical case $u_{j'-1} \supsetneq u_{j-1}$ and c'.

views of monotonically decreasing height $(h(v_0) \geq h(u_0) \geq h(v_1) \geq \cdots \geq h(u_t))$. To capture the "rate" of this decrease, we record the list of minor view heights $H_c := \big(h(u_j)\big)_{0 \leq j \leq t}$, and order each $\mathcal{C}_t$ lexicographically: $c' \leq c$ iff $H_{c'} \leq_{\text{lex}} H_c$. With respect to this total order, "smaller" computation means "steeper".

Long and steepest computation. We fix t to be a multiple of t^* which is at least $|V|$, and select c to be steepest in $\mathcal{C}_t$. We let q, e_j, v_j, u_j be as usual.

Since $t \geq |V|$, the list $u_0, \ldots, u_t$ contains repetitions. Let $j' < j$ be the indices for the earliest one. Then $u_{j'} = u_j$, so $h(u_{j'}) = h(u_j)$, and thus all views in between have the same height: $h(u_{j'}) = h(v_{j'+1}) = \cdots = h(v_j) = h(u_j)$. As a result, each major view equals the next minor one: $v_{j'+1} = u_{j'+1}, \ldots, v_j = u_j$.

Case 2A: $j' = 0$. Then $h(u_0) = h(v_1) = \cdots = h(v_j) = h(u_j)$, and therefore $v_1 = u_1, \ldots, v_j = u_j$. In fact, we also have $h(v_0) = h(u_0)$, and therefore $v_0 = u_0$.

To see why, suppose $h(v_0) \neq h(u_0)$. Then $v_0 \supsetneq u_0$. Since $u_0 \in V$, some state $\tilde{p}$ has $\text{LVIEW}(y_\iota)(\tilde{p}) = u_0$ (Figure 17a), and thus Lemma 37 applies to it (since $u_0 \subsetneq v_0$). In particular, $\text{LVIEW}(y_\iota)(\tilde{p}) \supseteq \text{LVIEW}\big(y_\iota(x_\iota y_\iota)^t\big)(\tilde{p})$ (since t is a multiple of t^*). On the other hand, u_0 contains the left end-point of e_1, so the part of c after e_1 shows that $q \in \text{LVIEW}\big(y_\iota(x_\iota y_\iota)^t\big)(\tilde{p})$, and thus $q \in \text{LVIEW}(y_\iota)(\tilde{p}) = u_0$. Since $u_0 \subseteq v_0$, this means that c is not an escape computation, a contradiction.

So, $h(v_0) = h(u_0) = \cdots = h(v_j) = h(u_j)$ and $v_0 = u_0, \ldots, v_j = u_j$ (Figure 17b). By the selection of p, its view on the j-th extension escapes v_0. Pick any $c' \in \mathcal{C}_j$, with exit state $q' \notin v_0$, critical steps $e'_1, \ldots, e'_j$, and major views $v'_0, \ldots, v'_j$. Then $v'_0 = v_0$ (since both c' and c start at p) and $q' \in v'_j \setminus v_j$ (since $v_j = u_j = u_0 = v_0$

and $q' \notin v_0$). So, the respective major views start with inclusion $v'_0 \subseteq v_0$ but end with non-inclusion $v'_j \not\subseteq v_j$. So there is $1 \leq l \leq j$ so that $v'_{l-1} \subseteq v_{l-1}$ but $v'_l \not\subseteq v_l$.

We are now ready to prove that $\{e'_l, e_l\}$ is incompatible with y_ι. The argument is as in Example II. Suppose the two steps participate in a set inducing an automorphism ζ. Since $v'_{l-1} \subseteq v_{l-1}$, both e'_l and e_l have their left end-points in v_{l-1}. Hence, $\zeta(v_{l-1}) \supseteq v'_l \cup v_l$. Since $v'_l \not\subseteq v_l$, the height of $\zeta(v_{l-1})$ is greater than that of v_l. But $h(v_{l-1}) = h(v_l)$. Therefore $h(\zeta(v_{l-1})) > h(v_{l-1})$, a contradiction.

Case 2B: $j' \neq 0$. Then we can talk of the minor views $u_{j'-1}$ and u_{j-1} that precede the first repetition. Of course, $u_{j'-1} \neq u_{j-1}$. In fact, $u_{j'-1} \not\supseteq u_{j-1}$.

To see why, suppose $u_{j'-1} \supseteq u_{j-1}$ (Figure 17c). Then $u_{j'-1} \supsetneq u_{j-1}$ (since $u_{j'-1} \neq u_{j-1}$) and thus $h(u_{j'-1}) > h(u_{j-1})$. Moreover, e_j has its left end-point in $v_{j'-1}$ (since $v_{j'-1} \supseteq u_{j'-1} \supseteq u_{j-1}$) while its right end-point has view $u_{j'}$ (since $v_j = u_j = u_{j'}$). Hence, by replacing $e_{j'}$ with e_j, we get a new computation c' that is also in C_t. In addition, $H_{c'}$ differs from H_c only in that $h(u_{j'-1})$ is replaced by $h(u_{j-1})$. But then c' is strictly steeper than c, a contradiction.

We are now ready to prove that $\{e_{j'}, e_j\}$ is incompatible with y_ι. The argument is as in Example III. Suppose the two steps participate in a set inducing an automorphism ζ. Because of e_j, $\zeta(u_{j-1}) \supseteq u_j$; but $h(u_{j-1}) = h(u_j)$ and ζ respects heights, so in fact $\zeta(u_{j-1}) = u_j$. Because of $e_{j'}$, $\zeta(u_{j'-1}) \supseteq u_{j'} = u_j$; so there exists $u^* \subseteq u_{j'-1}$ such that $\zeta(u^*) = u_j$. Overall, $u^* \neq u_{j-1}$ (since exactly one is in $u_{j'-1}$) and $\zeta(u^*) = \zeta(u_{j-1})$. Hence ζ is not injective, a contradiction.

This concludes the analysis of the case $\iota' = \iota$ and thus the proof of Theorem 27.

3.6. 2DFAs versus SNFAs.

3.6. 2DFAs versus SNFAs. As mentioned in the Introduction, an easy modification of the proof of Theorem 27 allows us to also establish the following.

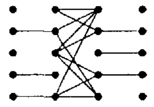
38. THEOREM. *The trade-off from* 2DFAS *to* SNFAS *is exponential.*

In other words, there exists a problem that can be solved by small 2DFAS but cannot be solved by small SNFAS. This problem is simply an appropriate restriction of liveness and the small 2DFA solving it is actually single-pass.

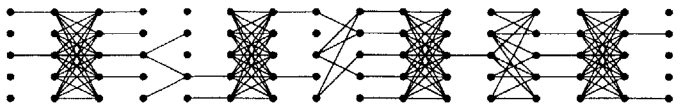
To describe this restriction, let us use Σ'_n to denote the subset of Σ_n containing only the 2^n 'parallel' symbols of the form $\{(a,a) \mid a \in \alpha\}$ for $\alpha \subseteq [n]$. For example, the leftmost symbol in Figure 14a is in Σ'_5, for $\alpha = \{2,3,4,5\} \subseteq [5]$. Let us also recall the complete symbol $\eta = [n]^2$ from Section 3.4-II. The restriction of liveness that we have in mind can be described as the promise problem B'_n where all inputs are promised to follow the pattern

$$\Sigma'_n(\eta\Sigma'_n\Sigma_n\Sigma'_n)^*\eta\Sigma'_n$$

In other words, according to this promise, every input z starts and ends with a parallel symbol and the rest of it consists of one or more copies of η separated by 3-symbol snippets of the form



where the outer symbols have to parallel and the middle one can be anything. For example, here is an input that obeys this promise:



Notice that *every such string is live iff its first and last symbol are non-empty and its snippets are all live.* Intuitively, the copies of the complete symbol 'reset' liveness every four symbols.

This last observation immediately suggests a small 2DFA algorithm for solving liveness under this promise: just check that the first and last symbols are non-empty and that every snippet is live. More carefully, the algorithm is as follows:

> We read the first symbol. If it is empty, we hang. Otherwise, we start scanning the input from left to right. Every time that we read a copy of η, we use a depth-first search to check whether the string of the next 3 symbols is live. If it is not, we hang. If it is, we move to the next copy of η. If there is only 1 next symbol, we check whether it is empty or not. If it is, we hang. Otherwise, we accept.

Easily, this algorithm can be implemented on a ZDFA with only $O(n^2)$ states.[6]

On the other hand, even under this promise, every SNFA solving liveness still needs $2^{\Omega(n)}$ states. This is true simply because the promise does not invalidate our argument for the general case, as the hard inputs constructed in Section 3.4-II can all be drawn so as to obey the promise. More specifically, we can replace property P_ι with the property $P'_\iota \subseteq P_\iota$ which contains only the strings of P_ι that obey the promise. Easily, P'_ι is still non-empty and infinitely extensible in both directions. So, we can again find an L-generic string $y'_{\rm L}$ and an R-generic string $y'_{\rm R}$ over P'_ι and construct $y'_\iota := y'_{\rm L}\eta y'_{\rm R}$, which is clearly in P'_ι and is thus generic over P'_ι. Then, it is trivial to verify that the sequence $(y'_\iota)_{\iota\in\mathcal{I}}$ is related to $(x_\iota)_{\iota\in\mathcal{I}}$ exactly as $(y_\iota)_{\iota\in\mathcal{I}}$ is in Lemma 34. The rest of the proof remains the same.

4. Conclusion

In the first part of this chapter, we focused on a natural class of restricted but still fully bidirectional 2NFA algorithms for liveness, which includes the small 1NFA solvers. We asked whether *small* 2DFAs from that class can succeed and proved that they cannot, *no matter how large they are.*

It is certainly good to provably know that graph exploration alone can never be a sufficient strategy. However, as already mentioned in the Introduction, in the context of the full conjecture the emphasis above stresses an alarming mismatch: a complexity question received a computability answer. This suggests that *the reasons why deterministic moles fail against liveness are only loosely related to the reasons why small* 2DFA*s fail —if they really do.* In order for this approach to ultimately be of any use against the full conjecture, we need restricted versions of fully bidirectional 2DFAs that are *both* weak enough to succumb to our arguments *and* strong enough to keep us in complexity: large 2DFAs of this kind should be able to solve liveness.

In the second part of the chapter, we proved that a SNFA must be exponentially large to solve the complement of liveness, and thus that small SNFAs are not closed

[6]In fact, replacing the depth-first search on the 3-symbol snippets with a cleverer search, we can reduce the size of this ZDFA to only $O(n^2/\log n)$ states—which, by the way, is asymptotically optimal for this restriction of liveness. However, the algorithm is too complicated to describe here and is not necessary for Theorem 38, anyway.

under complement. With an easy modification, our proof also showed that 2DFAs can be exponentially more succinct than SNFAs.

An interesting next question concerns the exact value of our lower bound. The smallest known SNFA for $B_{n,\emptyset}$ is the obvious 2^n-state 1DFA. *Is this really the best* SNFA *algorithm?* If so, then nondeterminism and sweeping bidirectionality together are completely useless in this context.

A preliminary version of the contents of Section 2 can be found in [**27**]. The contents of Section 3 can be found in an article to be included in [**30**].

CHAPTER 3

Non-Recursive Trade-Offs

In Chapters 1 and 2 we compared the relative succinctness of several pairs of types of machines. In each case, the two types had the same computational power, in the sense that they could solve the same class of problems—the regular languages. Moreover, the associated trade-off was easily seen to be bounded from above by some recursive function. In contrast, the comparisons that we will consider in this chapter are more general. We will discuss conversions between types of machines of different computational power and, most often, the associated trade-offs will be growing faster than any computable function.

One of the earliest studies of the relative succinctness of types of machines of different power was conducted in [**56**], as part of a proof that we can algorithmically check whether the language of a deterministic pushdown automaton (1DPA) is regular or not. Stearns showed that, although not every 1DPA has equivalent 1DFAs, whenever such equivalent automata exist, the smallest among them are at most triple-exponentially larger than the 1DPA.

This naturally lead to the corresponding question for one-way nondeterministic pushdown automata (1NPAs): in the case where a 1NPA has equivalent 1DFAS, what is an upper bound for the size of the smallest among them? The answer was qualitatively new, by Meyer and Fischer [**35**], who showed that every such bound grows (as a function of the size of the 1NPA) faster than any computable function. Hence, among the cases where it is possible to convert a 1NPA into a 1DFA, the trade-off in the size of description is in general *non-recursive.*[1] Several refinements of this result followed [**59, 50**].

In an important development, Hartmanis [**16**] later explained that the recursiveness of the trade-off from a type of machines A to a not-as-powerful type of machines B typically implies the recognizability (semi-decidability) of the corresponding *inadequacy problem*: "given a machine of type A, check that it has no B-equivalents". This greatly simplified the proofs of [**35, 59, 50**], while it nicely revealed the connections of the entire discussion to Gödel's theorem that the addition of an extra axiom to a formal system typically results in non-recursively shorter proofs for some of its theorems [**17**].

[1]As already noted in the introduction, this name can be misleading. Our intention here is to characterize the trade-offs that admit no recursive upper bounds. Clearly, every such trade-off is non-recursive. However, it is conceivable that there exist non-recursive trade-offs that admit recursive upper bounds. (It is easy to present natural functions that fall into this category. However, we do not know of one that is also the trade-off of some natural conversion.) So, strictly speaking, the class of non-recursive trade-offs is a subclass of the class of trade-offs that do not admit recursive upper bounds and, if the two classes are in fact equal, then an argument is needed to support this. With this clarification, we will move on using the popular choice. Note that, under this choice, a "recursive trade-off" is one that admits recursive upper bounds, and the "(non-)recursiveness of a trade-off" refers to the (non-)existence of a recursive upper bound for it.

Today many refinements of the above results are known and non-recursive trade-offs have emerged in numerous other comparisons between different types of machines. Comprehensive surveys can be found in [**15, 32**].

1. Two-Way Multi-Pointer Machines

In a remark in [**19**], Hartmanis and Baker showed that *a non-recursive trade-off can occur even when an optimal algorithm replaces a near-optimal one.*[2] For example, converting an $n^{2+\epsilon}$-space deterministic Turing machine (DTM) into one that uses only n^2-space involves a non-recursive blowup in the size of description. In the pattern of [**16**], they derived this observation from the unrecognizability of the inadequacy problem from near-optimal to optimal machines (from $n^{2+\epsilon}$-space to n^2-space DTMs), which in turn was shown to be a consequence of the fact that the near-optimal complexity class is strictly larger than the optimal one (some $n^{2+\epsilon}$-space DTMs have no n^2-space equivalent).

In this chapter we will refine that argument. We will prove a general theorem that directly shows the non-recursiveness of the trade-off in many conversions between machines of different power. In loose terms, our theorem states the following:

If two types of machines A *and* B *are such that*

1. *some machine of type* A *has no equivalent machine of type* B*, and*
2. *a machine of type* A *has enough resources to simulate a unary two-way deterministic finite automaton which has access to a linearly-bounded counter,*

then the trade-off from machines of type A *to machines of type* B *is non-recursive.*

For example, for the previous remark on space, we argue that, since $n^2 = o(n^{2+\epsilon})$, there exist $n^{2+\epsilon}$-space DTMs with no n^2-space equivalent, so that condition 1 is clearly true; that condition 2 is true, too, follows from the easy observation that any $\Omega(\lg n)$ amount of space suffices for the simulation of a linearly-bounded counter.

The most characteristic applications of our theorem concern the successive levels of hierarchies of two-way multi*pointer* automata, where by 'pointer' we mean any of the following accessories (in order of nondecreasing power): a linearly-bounded counter; a *blind* read-only head, namely a head that cannot distinguish between different input symbols (but can distinguish between input symbols and end-markers); an ordinary read-only head; a *sensing* read-only head, namely one that can sense which of the other heads are at the same cell as itself; or a pebble.

For example, we can establish the non-recursiveness of the following trade-offs (in each case, the reference indicates where condition 1 of the theorem has been established; for condition 2, it is always easy to see that it is also satisfied):

- from $k+1$ to k counters, on linearly-bounded two-way deterministic counter automata (unary or not) [**42**],
- from $k+1$ to k heads, on two-way multi-head finite automata (deterministic or not, unary or not) [**40, 41, 42**],
- from $k+1$ to k heads, on two-way multi-head pushdown automata (deterministic or nondeterministic) [**23**],

for all k. Sometimes, we can only be as refined as the hierarchy is known to be:

- from $k+2$ to k registers, on linearly-bounded register machines (deterministic or nondeterministic) [**42**],

[2]The reader is referred to [**19, 17, 18**] for a quite interesting discussion of the implications that this might have to our search for optimal algorithms.

- from $k+2$ to k counters, on linearly-bounded two-way nondeterministic counter automata (unary or not) [**42**],

for all k. Similarly, the trade-off is non-recursive

- from 3 to 2 heads, on a *simple* two-way deterministic finite automaton [**9**] (a multi-head automaton is simple if every input head after the first one is blind). It remains non-recursive even when we start from a 2-head two-way deterministic finite automaton, or from a 1-head two-way deterministic pushdown automaton [**9**].

Finally, we can also conclude the non-recursiveness of the trade-off, for $k \geq 2$:

- from $k+1$ to k work-tape symbols, on Turing machines (deterministic or not) that, on every input of length n, use no more than $\lg n$ work-tape cells [**52**] (even if the starting Turing machine has unary input alphabet, but then only for sufficiently large k).

Other conversions between machines of different power can be treated similarly.

Returning to the statement of the theorem above, we warn that it is, in fact, incomplete. Additional conditions have to be met, concerning A and B, their descriptions, and how 'size' is measured. However, in most interesting cases these conditions are trivially satisfied (in the above examples they are), so that listing them in this introduction would be a distraction. The complete list is contained in the formal statement of the theorem in Section 3.

The next section describes the formal framework of this study in more detail. Section 3 states and proves the theorem, except for an important lemma, which is proved in Section 5, after some preparation in Section 4. We warn that the discussion in this chapter is going to be much more abstract than in Chapters 1 and 2, so as to ensure that the conclusions cover as many conversions as possible—including cases where A or B denote types of language descriptors other than machines (e.g., regular expressions, grammars). A more concrete discussion can be found in [**26**], where the theorem is proved specifically for the conversion from $k+1$ to k heads on two-way multi-head finite automata.

2. Preliminaries

We write $\mathbb{N}$ for the set of positive integers and $\lg_a n$ for $\lfloor \log_a n \rfloor$, for all $n, a \in \mathbb{N}$.

As usual, given any problem $\Pi = (\Pi_{\text{yes}}, \Pi_{\text{no}})$ over some alphabet Σ and any DTM M, we say that M *recognizes* Π if M accepts all $w \in \Pi_{\text{yes}}$ and rejects (possibly by looping) all $w \in \Pi_{\text{no}}$. If some DTM recognizes Π, we say Π is (*Turing-*) *recognizable*. If Π' is also a problem over Σ, we write $\Pi \leq \Pi'$ and say that Π *reduces to* Π' iff there is a DTM that, on input $w \in \Pi_{\text{yes}} \cup \Pi_{\text{no}}$, eventually halts with an output w' such that

$$w \in \Pi_{\text{yes}} \implies w' \in \Pi'_{\text{yes}} \qquad \text{and} \qquad w \in \Pi_{\text{no}} \implies w' \in \Pi'_{\text{no}}.$$

Clearly, if some unrecognizable Π reduces to Π', then Π' is also unrecognizable.

If Π is a language ($\Pi_{\text{yes}} + \Pi_{\text{no}} = \Sigma^*$) and Π_{yes} contains exactly all sufficiently long strings, for some interpretation $0 \leq l \leq \infty$ of 'sufficiently long'

$$\Pi_{\text{yes}} = \{w \in \Sigma^* \mid |w| \geq l\},$$

we say that Π *obeys a threshold.* Note that then Π_{yes} is empty iff this threshold is infinite. A machine that solves Π is similarly said to obey the same threshold.

2.1. Descriptional systems. A *descriptional system* over the alphabets Γ and Σ is any set $D \subseteq \Gamma^*$ of *names* (or *descriptors*), along with two total functions $(\cdot)_D$ and $|\cdot|_D$, mapping every name $d \in D$ to its *language* $(d)_D \subseteq \Sigma^*$ and its *size* $|d|_D \in \mathbb{N}$, respectively. For example, suppose that we fix a binary encoding of all 1DFAs over, say, the input alphabet $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$. This induces the descriptional system over $\{0, 1\}$ and $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ that contains all encoding strings as names and maps each of them to the language accepted by the corresponding 1DFA (as its language) and to the number of states in that 1DFA (as its size). Alternatively, the size of a name could just be its length.

A system D is *decidable* if the membership problem for its names is decidable. That is, if there exists a DTM U_D that always halts and is such that:

$$\text{for all } d \in D \text{ and } w \in \Sigma^*: \qquad U_D(d, w)\text{accepts} \iff w \in (d)_D.$$

Thus, the system of the previous example is clearly decidable, whereas a system containing binary encodings of DTMs would be undecidable.

In order to be able to compare two descriptional systems D and E in terms of their relative succinctness, we require that they are *comparable*, in the sense that [i] they are defined over the same alphabets, and that [ii] their $(\cdot)$ and $|\cdot|$ mappings agree on all common names,[3]

$$\text{for all } z \in D \cap E: \quad (z)_D = (z)_E \quad \text{and} \quad |z|_D = |z|_E,$$

so that subscripts can be dropped: for all $z \in D \cup E$, (z) and $|z|$ are unambiguous. For such systems, the comparison of E against D involves two natural notions:

I. For a name $e \in E$, there *may* or *may not* exist a name in D that maps to the same language. In the latter case, we say that D *is inadequate for describing the language of* e and, accordingly, we call the associated computational problem, "given an $e \in E$, check that no $d \in D$ maps to (e)", the *inadequacy problem from* E *to* D. Formally, this is the promise problem $\mathsf{I} = (\mathsf{I}_{\text{yes}}, \mathsf{I}_{\text{no}})$, with:

$$\mathsf{I}_{\text{yes}} := \{e \in E \mid (d) \neq (e) \text{ for all } d \in D\},$$
$$\mathsf{I}_{\text{no}} := \{e \in E \mid (d) = (e) \text{ for some } d \in D\}.$$

Notice that e is promised to be in E, so that solving I does not require checking membership in E (which might be hard, even impossible).

II. When a name $e \in E$ does have equivalent names in D (i.e., names mapping to (e)), we naturally ask how larger than e the smallest of these D-equivalents are. As usual, we answer this question with a function $f : \mathbb{N} \to \mathbb{N}$ that upper bounds this increase in size in terms of the size of e. Namely, f is such that

for all $s \in \mathbb{N}$ and all $e \in E$ of size s: if D contains names that are equivalent to e, then at least one of them is of size at most $f(s)$.

We say that f *upper bounds the trade-off* (for the conversion) *from* E *to* D.[4] When a computable such upper bound exists, we say *the trade-off from* E *to* D *is recursive.*[5]

As first noted in [**16**], discussions I and II are not unrelated: unrecognizability of the inadequacy problem typically implies that the trade-off is non-recursive.

[3]It is only for simplicity that we require the agreement for $|\cdot|$; we do not actually need it.

[4]Note that this defines directly the notion of an *upper bound* for the trade-off from E to D. A more natural approach would be to first define the notion of the trade-off from E to D (in the sense of Chapters 1 and 2), and only then say what an upper bound for it is. However, that would be redundant, as our goal in this chapter is to show the non-recursiveness of the upper bounds.

[5]See the discussion in Footnote 1 on page 81 about what "recursive trade-off" means.

1. LEMMA (Hartmanis). *Suppose D, E are two* comparable *descriptional systems over alphabets Γ and Σ, and that the following conditions are met:*

H_1. *both D and E are decidable,*

H_2. *for every $e \in E$, we can effectively compute its size $|e|$, and*

H_3. *there is a halting* DTM *that, given $s \in \mathbb{N}$, produces a list $Z \subseteq \Gamma^*$ such that*

i. *the non-D names can be recognized in Z: $(Z \cap \overline{D}, Z \cap D)$ is recognizable.*

ii. *the languages of the D-names in Z cover all and only those languages over Σ that are supported by a name in D of size at most s:*

$$\{(z) \mid z \in Z \cap D\} = \{(d) \mid d \in D \ \& \ |d| \leq s\}.$$

Then, recursiveness of the trade-off from E to D implies that the corresponding inadequacy problem is recognizable.

Before giving the proof, let us remark how mild conditions H_1–H_3 are. For most interesting cases, the first two of them are trivially true and H_3 is satisfied via the DTM that simply lists all names in D that have size $\leq s$ (so that the problem of H_3i is trivially decidable and the two sets of H_3ii trivially identical). Having H_3 as complicated simply covers some special cases (e.g., comparing general to unambiguous context-free grammars [**17**, Example 2]).

PROOF. Suppose D, E are as in the statement and f is a computable upper bound for the trade-off from E to D. To check that a given $e \in E$ has no D-equivalents, we first compute $s := f(|e|)$ (by H_2 and since f is computable). We then run the DTM guaranteed by H_3 on s, to produce a (finite, since the DTM is halting) list of names $Z := \{z_1, z_2, \dots, z_k\}$. At that moment, we know (by the selection of f and H_3ii) that we should accept iff every D-name in Z maps to a language different from (e).

Equivalently, we should accept iff: for every $z \in Z$, *either* z is not a D-name *or* z is a D-name and (z), (e) differ at one or more $w \in \Sigma^*$. In order to check this, we start simulating, in two parallel threads:

I. the recognizer guaranteed by H_3i on each of $z_1, z_2, \dots, z_k$ in parallel, and

II. for all $w \in \Sigma^*$: the machines U_E and U_D (guaranteed by H_1) respectively on (e, w) and on each of $(z_1, w), (z_2, w), \dots, (z_k, w)$.

Whenever a $z \in Z$ is accepted in thread I, we cross it off the list. Whenever a $z \in Z$ is found to disagree with e on some w in thread II, it is crossed off the list, as well. Finally, if the list ever gets empty, we accept.

Clearly, every string in Z that is not a D-name, will eventually be crossed off, in thread I. Similarly, each D-name that is inequivalent to e will also be eventually removed, in thread II. Moreover, neither thread can delete a D-name that is equivalent to e. Hence, the list will eventually get empty iff e had no D-equivalent in the original list Z, which is true iff e has no D-equivalent at all. □

2.2. Multi-counter automata. Our main theorem will need to make use of the natural notion of a *unary two-way deterministic finite automaton that has additional access to a number of counters.* Such models are of course known and well studied, but mainly for non-unary alphabets—see, for example, the two-way multi-counter machines of [**11, 42**]. Since we will only be interested in the unary case, it is possible to simplify the model in helpful ways. Most notably, we can avoid the notion of input tape, and assume instead that the input is the upper bound for one of the counters.[6] The simplified definition follows.

A *deterministic automaton with k counters* ($\textsc{dca}_k$) consists of a finite state control and k counters, each of which can store a nonnegative integer. One of the counters is distinguished as *primary*, the rest being referred to as *secondary*. The input to the automaton is a nonnegative upper bound n for the primary counter. The machine starts at a designated *start state* with all its counters set to 0. At each step, based on its current state, the automaton decides which counter it should act upon and whether it should decrease it or increase it. Then the action is *attempted*. An attempt to decrease fails iff the counter already contains 0; an attempt to increase fails iff the counter is the primary one and it already contains n; an attempt to increase a secondary counter never fails. A failed attempt leaves the counter contents intact; a successful attempt updates the counter contents accordingly. Based on its current state and on whether the attempt succeeded or not, the automaton selects a new state and moves to it. The input is *accepted* if the machine ever enters a designated *final state*. The *language* of the machine is exactly the set of inputs that it accepts. If, for all n, the behavior of the automaton is such that no secondary counter ever grows larger than n, we say that the automaton is (*linearly*) *bounded*.

We will be interested in a special version of the emptiness problem for multi-counter automata. One way to introduce this problem is to start with the emptiness problem for DTMs ("given a description of a DTM, check that the language of the machine is empty"), which is well known to be unrecognizable [**21**], and to consider certain ways of 'simplifying' it:

- What happens if, instead of a full-fledged DTM, the machine we are given is 'simpler'? Say, a multi-counter automaton? Or just a $\textsc{dca}_2$? Clearly, checking emptiness becomes 'simpler', too. Does it also become recognizable?
- What if the given $\textsc{dca}_2$ is also promised to be *bounded*? And *terminating*, too? And to also *obey a threshold*? As the promise gets stronger, checking emptiness again becomes 'simpler'. But does it become recognizable?

So, the problem that we want to define is the following: "given a description of a $\textsc{dca}_2$ that is promised to be bounded and terminating and to obey a threshold, check that its language is empty." In formal dialect, $\mathsf{E} = (\mathsf{E}_{\text{yes}}, \mathsf{E}_{\text{no}})$, where

$$\mathsf{E}_{\text{yes}} := \{z \in \langle \textsc{dca}_2^* \rangle \mid (z) = \emptyset\}$$
$$\mathsf{E}_{\text{no}} := \{z \in \langle \textsc{dca}_2^* \rangle \mid (z) \neq \emptyset\}.$$

Here, we use $\langle \textsc{dca}_2^* \rangle$ to denote the set of descriptions (under a fixed encoding) of all terminating, bounded $\textsc{dca}_2$s that obey a threshold, whereas (z) stands for the language of the machine described by z. Interestingly, although not surprisingly, even for such a weak automaton and under such a strong promise, emptiness remains unrecognizable:[7]

2. LEMMA. E *is unrecognizable.*

We use this fact in the next section, but defer proving it until Section 5. In between, Section 4 discusses the capabilities of multi-counter automata.

[6]Here there is a difference from [**26**], where the upper bound is applied to all counters.

[7]Note that clearly $\mathsf{E} \in \Pi_1$ and that the proof of Lemma 2 will show E is Π_1-complete. We also remark that, under no promise and after non-trivially modifying the definition of $\textsc{dca}_2$s, E is the emptiness problem for 2-register machines, which is well known to be Π_1-complete [**38**].

3. The Main Theorem

We are now ready to state and prove the main theorem.

3. THEOREM. *Suppose D, E are two* comparable *descriptional systems that satisfy conditions* H_1–H_3 *of Lemma* 1. *If they also satisfy the following:*

C_1. *there exists a name $e_0 \in E$ that has no equivalent in D,*

C_2. *given a description z of a* terminating, bounded $\textsc{dca}_2$ *that* obeys a threshold, *we can effectively construct a name $e_z \in E$ such that*

$$(e_z) = (e_0) \cup \{w \in \Sigma^* \mid |w| \in (z)\}, \tag{9}$$

C_3. *every* co-finite *language has a name in D that maps to it,*

then the trade-off from E to D is non-recursive.

Before proving the theorem, we discuss how mild conditions C_1–C_3 really are. Since every co-finite language is regular, C_3 is trivially satisfied whenever the names in D describe machines that have some kind of finite state control.

The second condition essentially says that the machines described by E have enough resources to simulate a bounded $\textsc{dca}_2$. Because then, from a given z, we can always construct the description e_z of the E-machine that does the following:

> on input $w \in \Sigma^*$: first simulate on $|w|$ the $\textsc{dca}_2$ described by z; if this accepts, then halt and accept; otherwise, simulate on w the machine described by e_0 and accept, reject, or loop accordingly.

and which obviously satisfies (9) (note the importance of the promise that z describes a $\textsc{dca}_2$ that never rejects by looping). Given how weak bounded $\textsc{dca}_2$s are, most two-way machines with non-regular capabilities will easily meet C_2.

The important condition is C_1, which requires that the machines described by D are *not as powerful as* those described by E; in other words, a separation is needed between the complexity classes that correspond to the two systems.

PROOF. We essentially repeat Hartmanis' argument from [**17**, Example 4] (see also [**31**, Theorem 7]). Suppose D, E are as in the statement of the theorem. Since H_1–H_3 are satisfied, Lemma 1 implies that we only need to prove that the inadequacy problem I from E to D is unrecognizable. By Lemma 2, we just need to reduce E to it:

$$\mathsf{E} \leq \mathsf{I}.$$

Given a $z \in \langle \textsc{dca}'_2 \rangle$, we simply construct the name $e_z \in E$ guaranteed by conditions C_1 and C_2, so that

$$(e_z) = (e_0) \cup \{w \in \Sigma^* \mid |w| \in (z)\}.$$

If $z \in \mathsf{E}_{\mathsf{yes}}$, then the language of z is empty, so that $(e_z) = (e_0)$ and e_z has no D-equivalent (because e_0 does not); hence $e_z \in \mathsf{I}_{\mathsf{yes}}$. On the other hand, if $z \in \mathsf{E}_{\mathsf{no}}$, then the language of z contains all sufficiently large $w \in \Sigma^*$, so that (e_z) is co-finite and has D-equivalents (by C_3); hence $e_z \in \mathsf{I}_{\mathsf{no}}$. This concludes the proof. $\square$

As a side remark, we note that the proof has shown a slightly stronger fact: problem I remains unrecognizable even under the promise that the given $e \in E$ either has no D-equivalent or its language is co-finite. In addition, the promise that the given $\textsc{dca}'_2$ obeys a threshold can be slightly relaxed: we only need to know that its language is either empty or co-finite.

4. Programming Counters

In order to present the capabilities of multi-counter automata, we introduce some 'program' notation. First, the two atomic operations, the attempt to decrease a counter X and the attempt to increase it, are denoted respectively by

$$X \xleftarrow{f} X-1 \qquad \text{and} \qquad X \xleftarrow{f} X+1,$$

where, in each case, flag f is set to true iff the attempt succeeds. Then, the compound operation of setting X to 0, denoted by $X \longleftarrow 0$, can be described by

$$\texttt{repeat } X \xleftarrow{f} X-1 \texttt{ until } \neg f. \tag{10}$$

If a second counter Y is present, we can transfer the contents of Y into X: we set X to 0, then repeatedly decrease Y and increase X until Y is 0. We denote this by

$$(X,Y) \xleftarrow{f} (Y,0),$$

and describe it by a line similar to (10). Note that, if X is the primary counter and $Y > n$, then one of the attempts to increase X will fail; in that case, we restore the original value of Y returning X to 0, and set flag f to false. So, X's original contents are always lost, but this never happens to the original contents of Y.

Changing how fast X increases as Y decreases, we can multiply/divide Y into X by any constant $a \in \mathbb{N}$. We denote these operations by

$$(X,Y) \xleftarrow{f} (aY,0) \qquad \text{and} \qquad (X,Y) \xleftarrow{f,r} (\lfloor \tfrac{Y}{a} \rfloor, 0),$$

where, in the second operation, we also find the remainder and return it in r. As before, if X is the primary counter and $aY > n$ (respectively, $\lfloor Y/a \rfloor > n$) then one of the attempts to increase X will fail; we then restore the original value of Y returning X to 0, and set flag f to false.

At a higher level, we can try to multiply Y by a constant a (into Y) using X as an auxiliary counter and making sure Y changes only if the operation succeeds:

$$(X,Y) \xleftarrow{f} (aY,0); \quad \texttt{if } f \texttt{ then } (Y,X) \xleftarrow{\texttt{t}} (X,0).$$

Note the use of `t` in the place of a flag, indicating that the action is guaranteed to be successful. Division (with remainder) can be performed in a similar manner. We denote the two operations by

$$Y \xleftarrow{f,X} aY \qquad \text{and} \qquad Y \xleftarrow{f,r,X} \lfloor \tfrac{Y}{a} \rfloor. \tag{11}$$

Now, if X is primary, we can set Y to the largest power of a that can fit in n:

$$\begin{aligned} &X \longleftarrow 0;\ X \xleftarrow{f} X+1; \\ &\texttt{if } f \texttt{ then } \{Y \longleftarrow 0;\ Y \xleftarrow{\texttt{t}} Y+1;\ \texttt{repeat } Y \xleftarrow{g,X} aY \texttt{ until } \neg g\}, \end{aligned} \tag{12}$$

an operation that fails iff $n = 0$. To denote this operation, we use the notation:

$$Y \xleftarrow{f,X} a^{\lg_a n}$$

where, as already mentioned, $\lg_a n := \lfloor \log_a n \rfloor$.

If a third counter Z is present, we can modify (12) to also count (in Z) the number of iterations performed. This gives us a way to calculate $\lg_a n$:

$$Z \xleftarrow{f,X,Y} \lg_a n,$$

an operation that fails iff $n = 0$. In another variation, we can modify the multiplication in (11) so that the success of the operation depends on the *contents* of X (as opposed to its upper bound n):

$$Y \overset{f,X,Z}{\longleftarrow} aY,$$

meaning that, using Z as auxiliary and without affecting X: if $aY \leq X$, then Y is set to aY; otherwise, Y is unaffected. Specifically, to implement this, we first set Z to 0. Then, we repeatedly decrease Y, increase Z, and decrease X by a. *If X becomes 0 before Y*, then $aY > X$ and the operation should fail: we restore the original values of Y and X by repeatedly decreasing Z, increasing Y, and increasing X by a, until Z becomes 0. *Otherwise*, $aY \leq X$ and the operation will succeed: we copy the correct value to Y and restore the value of X by repeatedly decreasing Z and increasing each of Y, X by a, until Z becomes 0. Note that *if originally $Y, Z \leq X$, then at no point during the operation does any of the counters assume a value greater than the original value of X.*

Using this last operation, we can program the following variant of (12):

```
$X \overset{f}{\longleftarrow} X - 1$; if $f$ then {$X \overset{t}{\longleftarrow} X + 1$;
                    $Y \longleftarrow 0$; $Y \overset{t}{\longleftarrow} Y + 1$; repeat $Y \overset{g,X,Z}{\longleftarrow} aY$ until $\neg g$}
```

which implements the attempt to set Y to the largest power of a that is at most X, using Z as auxiliary and leaving X unaffected (and failing iff X is 0). We denote this operation by

$$Y \overset{f,Z}{\longleftarrow} a^{\lg_a X}.$$

It is important to note that, by the remark at the end of the previous paragraph, *if originally $Y, Z \leq X$, then during this operation no counter ever assumes a value greater than the original value of X.*

Hopefully, the reader is convinced of the quite significant capabilities of DCA$_k$s that have 2 or more counters. We will be using these capabilities in the next section.

5. Proof of the Main Lemma

We now prove that E is unrecognizable. We do this by a reduction from the complement of the halting problem, which is known to be unrecognizable [**21**]:

$$\overline{\text{HALTING}} \leq \text{E},$$

where $\overline{\text{HALTING}} := \{z \in \{0,1\}^* \mid z$ encodes a DTM that loops on $z\}$. That is, we give an algorithm that, on input a description z of a DTM M produces a description z' of a terminating, bounded, threshold-obeying DCA$_2$ M', such that

(13) $\qquad M$ loops on $z \implies (z') = \emptyset \qquad$ and $\qquad M$ halts on $z \implies (z') \neq \emptyset$.

In describing this algorithm, we will be calling a machine (DTM or DCA$_k$) *good*, if it is terminating, bounded (for DCA$_k$s), obeys a threshold, and its language satisfies (13) when it replaces (z'). Thus, for example, M' will be good.

On its way to z', the algorithm will construct descriptions of two other machines: a description z_A of a DTM A, and a description z_B of a DCA$_3$ B. In the sequence M, A, B, M' each machine after M will be defined in terms of the previous one and will be good. Our constructions use the ideas of [**61**] and [**38**] (also found in [**39, 21**]).

5.1. The first machine. A is a DTM with one tape, infinite in both directions; the tape alphabet is $\{\sqcup, 0, 1, \dot{0}, \dot{1}\}$, while the input alphabet is $\{0\}$. On input 0^n, A starts with tape contents

$$\cdots\sqcup\sqcup\sqcup\underbrace{000\cdots00}_{n \text{ times}}\sqcup\sqcup\sqcup\cdots$$

and its head on the $\sqcup$ next to the leftmost 0 (or any $\sqcup$, if $n = 0$). It then computes:

```
1. For all $w \in \{0,1\}^n$, from $0^n$ up to $1^n$:
      — if $w$ encodes a halting computation history of $M$ on $z$, accept.
2. Reject.
```

The check inside the loop presupposes some fixed reasonable encoding of sequences of configurations of M into binary strings, with the additional property that *if w encodes a computation history, then every string of the form $w0^*$ encodes the same computation history.*

Note that, using the extra dotted symbols, A can easily perform this check without ever writing a non-blank symbol on a $\sqcup$, or a $\sqcup$ on a non-blank symbol; and without ever visiting any $\sqcup$ that lies beyond the two that originally delimit the input. As a consequence, throughout its computation on 0^n, A keeps exactly n non-blank symbols on its tape, occupying the same n cells as the symbols of the input. Also note that, by the selection of the encoding scheme for M's computation histories, if A accepts an input 0^n, it necessarily accepts all longer inputs, as well.

5.2. The second machine. B is a DCA3 that, on input $n \geq 30$, simulates the behavior of A on input $0^{\lg_5 \lg_{30} n}$; on input $n < 30$, B just rejects. Note the strange length $\lg_5 \lg_{30} n$. This is chosen as a function of n that is (i) *computable* by a DCA3 and (ii) *increasing,* but also (iii) *small enough.* Goodness of B bases on (ii), whereas (iii) facilitates the simulation performed by M' in the next section.

To explain B's behavior, let J, L, R be its three counters. J is primary and helps performing operations on L and R, while L and R together encode tape configurations of A. To see the encoding, consider the following example of a configuration:

$$\begin{array}{cccccccccccccc}
\cdots & l_4 & l_3 & l_2 & l_1 & l_0 & h & r_0 & r_1 & r_2 & r_3 & r_4 & r_5 & \cdots \\
\hline
\cdots & \sqcup & \sqcup & \times & \times & \times & \times & \times & \times & \times & \times & \sqcup & \sqcup & \cdots \\
\hline
 & & & & & & \uparrow & & & & & & &
\end{array}$$

where $\times$ stands for any non-blank symbol, and $\uparrow$ shows the head position. Mapping symbols $\sqcup$, $\dot{1}$, $\dot{0}$, 1 and 0 to numbers 0, 1, 2, 3 and 4, respectively (in fact, any mapping that maps symbol $\sqcup$ to code 0 and symbol 0 to code 4 will do), we get each tape cell map to a digit of the 5-ary numbering system. Then, the head position splits the tape into three portions, which define the integers

$$l = \sum_{i=0}^{\infty} l_i \cdot 5^i \qquad \text{and} \qquad h \qquad \text{and} \qquad r = \sum_{r=0}^{\infty} r_i \cdot 5^i,$$

where the two sums are finite, exactly because $\sqcup$ maps to 0. The values l and r are kept in L and R, while h is kept in a register H in B's finite memory.

More specifically, on input n, B starts with a two-part initialization. First, it computes $\lg_{30} n$ into J, leaving 0s in L and R (this is if $n \geq 1$; if $n = 0$, B rejects):

$$R \xleftarrow{f,J,L} \lg_{30} n; \ \texttt{if} \ \neg f \ \texttt{then reject else} \ \{(J,R) \xleftarrow{\texttt{t}} (R,0);\ L \longleftarrow 0\}.$$

Then, it computes into R the value $5^m - 1$, where $m := \lg_5 \lg_{30} n$, leaving 0s in L and H (this is only if $J \geq 1$, that is if $n \geq 30$; otherwise, $n < 30$ and B rejects):

$$R \xleftarrow{f,L} 5^{\lg_5 J};\ \texttt{if}\ \neg f\ \texttt{then reject else}\ \{R \xleftarrow{\texttt{t}} R-1;\ L \longleftarrow 0;\ H \longleftarrow 0\}.$$

This completes the initialization, with $L = H = 0$ and $R = 5^m - 1$, or in 5-ary:

$$L = 0 \qquad \text{and} \qquad H = 0 \qquad \text{and} \qquad R = \underbrace{4\ 4\ 4\ \cdots\ 4}_{m \text{ times}}.$$

Hence L, H, R correctly represent A's starting tape configuration on input 0^m:

$$\begin{array}{cccccccccccc}
\cdots & l_1 & l_0 & h & r_0 & r_1 & r_2 & \cdots & r_{m-1} & r_m & r_{m+1} & \cdots \\
\hline
\cdots & \sqcup & \sqcup & \sqcup & 0 & 0 & 0 & \cdots & 0 & \sqcup & \sqcup & \cdots \\
\hline
 & & & \uparrow & & & & & & & &
\end{array}$$

since symbol 0 maps to code 4. At this point, B is ready to start a faithful step-by-step simulation of A.

The automaton remembers in its finite memory the current state of A as well as the code of the currently read symbol (in H). If s is the code of the new symbol to be written on the tape, B computes

$$L \xleftarrow{\texttt{t},J} 5L;\ \texttt{repeat}\ s\ \texttt{times:}\ L \xleftarrow{\texttt{t}} L+1;\ R \xleftarrow{\texttt{t},r_0,J} \lfloor \tfrac{R}{5} \rfloor;\ H \longleftarrow r_0$$

to simulate writing this symbol and moving to the right; similarly, it computes

$$R \xleftarrow{\texttt{t},J} 5R;\ \texttt{repeat}\ s\ \texttt{times:}\ R \xleftarrow{\texttt{t}} R+1;\ L \xleftarrow{\texttt{t},l_0,J} \lfloor \tfrac{L}{5} \rfloor;\ H \longleftarrow l_0$$

to simulate writing this symbol and moving to the left.

It is important to note the range of the values assumed by the counters. By the design of its main operation, the second part of the initialization phase never assigns to a counter a value greater than the original value of J, which is $\lg_{30} n$. Then, in the simulation phase, the behavior of A (the tape starts with m 0s and always contains exactly m non-blank symbols) and the selection of the symbol codes (0 gets the largest code) are such that the initial value $5^m - 1$ of R upper bounds all possible values that may appear in B's counters. One consequence of this is that all operations in the previous paragraph are guaranteed to be successful (hence the t reminder). Another consequence is that, since $5^m - 1 < \lg_{30} n$, *the entire computation of B after the first part of its initialization phase keeps all values of all counters at or below* $\lg_{30} n$. This will prove crucial in the next section.

5.3. The final machine. M' is a DCA$_2$ that simulates the behavior of B. If U, V are its two counters, then U is primary and helps performing operations on V, while V encodes the contents of the counters of B: whenever J, L, R, contain j, l, r respectively, V contains $2^j 3^l 5^r$.

The automaton starts by computing into V the product $30^t = 2^t 3^t 5^t$, where $t := \lg_{30} n$ (this is only if $n \geq 1$; if $n = 0$, M' rejects, exactly as B would do):

$$V \xleftarrow{f,U} 30^{\lg_{30} n};\ \texttt{if}\ \neg f\ \texttt{then reject.}$$

It then removes all 3s and 5s from this product, so that V becomes $2^{\lg_{30} n} 3^0 5^0$. Specifically, in order to remove all 3s, M' divides V by 3 repeatedly:

$$V \xleftarrow{\texttt{t},r,U} \lfloor \tfrac{V}{3} \rfloor,$$

until a non-zero remainder r is returned, which implies there were no 3s in V before the last division. Then the correction

$$V \xleftarrow{\mathtt{t},U} 3V;\ \mathtt{repeat}\ r\ \mathtt{times:}\ V \xleftarrow{\mathtt{t}} V+1$$

undoes the damage caused by the last division. After this, M' performs a similar computation to remove from V all 5s. At this point, the value $2^{\lg_{30} n}3^0 5^0$ correctly encodes the values of the counters of B *right after the first part of its initialization phase* and M' is ready to start a faithful step-by-step simulation of B.

The current state of B is stored in M''s finite memory. Whenever B tries to decrease J,

$$J \xleftarrow{f} J-1,$$

M' divides V by 2. If this division returns no remainder, then it has simulated a successful decrement; otherwise, the simulated attempt has failed, and M' restores the initial value of V:

$$V \xleftarrow{\mathtt{t},r,U} \left\lfloor \tfrac{V}{2} \right\rfloor;\ \mathtt{if}\ r=0\ \mathtt{then}\ f \longleftarrow \mathtt{true}\ \mathtt{else}$$
$$\{f \longleftarrow \mathtt{false};\ V \xleftarrow{\mathtt{t},U} 2V;\ \mathtt{repeat}\ r\ \mathtt{times:}\ V \xleftarrow{\mathtt{t}} V+1\}.$$

The attempts to decrease L or R are handled similarly, with 3 or 5 instead of 2.

Attempts of B to increase its counters are of course simulated by appropriate multiplications of V. The only subtlety involves failure during increment attempts. To be faithful, the simulation must ensure that

an attempt of B to increase a counter fails	iff	*the corresponding attempt of M' to multiply V fails.*

How is this condition satisfied? If it is, this does not happen in some obvious way. In B the upper bound for J is always the same (the input of B), whereas in M' the upper bound for its representation is the base 2 logarithm of a value that depends (on the input of M' and) on the values of the other two counters. Similarly, in B counter L is unrestricted, whereas in M' its representation is bounded by a value that depends on the other two counters—and the same is true for R.

The crucial observation (from the previous section) is that, since we are after the first part of B's initialization phase, *no counter of B ever assumes a value greater than $t = \lg_{30} n$.* This immediately implies that, after the initialization phase of M', *no counter of M' ever assumes a value greater than $2^t3^t5^t$.* Now, since $t < n$ and $2^t3^t5^t \le n$, we conclude that both

- all increment attempts of B are successful, and
- all corresponding multiplication attempts of M' are successful, as well.

Hence, the equivalence above is satisfied vacuously. Put another way, when M' multiplies V to simulate a counter increment in B, it knows in advance that this increment does not fail and therefore that the multiplication will not fail, either. Overall, B's atomic operation

$$J \xleftarrow{\mathtt{t}} J+1 \qquad \text{is simulated by} \qquad V \xleftarrow{\mathtt{t},U} 2V,$$

and similarly for L and R.

As a final remark, we note the immediate by-product of our last argument: Since V clearly never exceeds n during the initialization phase of M' and it also never exceeds n during the simulation of B, it follows that M' is bounded.

This concludes the definitions of all three machines in our reduction. It should be clear that M' is good and that a description z' of it can be computed out of z.

6. Conclusion

Using old ideas [**61, 38**], we showed the unrecognizability of the emptiness problem for DCA$_2$s that are promised to be bounded, always terminate, and obey a threshold. We then combined this with the idea of [**19**] to show that, if machines A have the resources to simulate DCA$_2$s of the particular kind and can also solve problems that machines B cannot, then typically the trade-off from A to B is non-recursive. Applying the theorem, we derived such trade-offs in many conversions.

We do not know if the emptiness problem of Section 2.2 remains unrecognizable even when the underlying machine is a *2-register automaton* [**38**] (that is, a DCA$_2$ that starts with n in its primary counter and where increments of that counter never fail). If it is, then our main theorem can be made slightly stronger.

A preliminary and more concrete version of the contents of this chapter can be found in [**26**]. An improved but more abstract version appeared in [**28**].

End Note

I would like to thank my research advisor, Michael Sipser, for suggesting to me the 2D vs. 2N problem and for being a constant source of encouragement and ideas as I was working on it during the past five years. I learned a lot from him about computation, and more generally about how to think and how to explain. I enjoyed the kindness and humanity of his personality and I particularly admire his ability to just take a few silent seconds and then return with the most valuable advice for whatever problem needs to be solved.

I would also like to thank Albert Meyer, from whom I also learned a lot over the past years as a teaching assistant for his class on discrete mathematics. I really enjoyed the honesty of his character and I cannot but admire his eagerness and ability to talk and think efficiently through almost any kind of problem.

This is probably a good place to also thank the people of MPLA in Greece, where my graduate studies actually began. When I moved to Athens in 1997, it was not certain that the universities was indeed going to be the place where I would be burning my energy. If it turned out this way, it is mainly due to the quality of the people and the academic program at MPLA. I am particularly grateful to Yiannis Moschovakis for encouraging me to continue my studies abroad.

On my way out of Greece, I also had the honor to meet General Leftheris and Mrs. Roula Kanellakis. Like so many other students, I am proud to have been a Paris Kanellakis Fellow and I have often drawn inspiration from Paris's academic conduct and achievements.

Finally, many thanks are due to my uncle Demos Fokas. When I first came to Cambridge, he had already been around for a while and he helped me a lot with the transition. Most importantly, having been through graduate school himself, Demos knew very well what this process is all about and in many occasions used his experience to provide me with valuable advice and inspiration. He is now in the 'southern provinces' already—and this is exactly where I am also heading for.

Bibliography

[1] Bruce H. Barnes. A two-way automaton with fewer states than any equivalent one-way automaton. *IEEE Transactions on Computers*, C-20(4):474–475, 1971.

[2] Piotr Berman. A note on sweeping automata. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, pages 91–97, 1980.

[3] Piotr Berman and Andrzej Lingas. On complexity of regular languages in terms of finite automata. Report 304, Institute of Computer Science, Polish Academy of Sciences, Warsaw, 1977.

[4] Jean-Camille Birget. Two-way automata and length-preserving homomorphisms. Report 109, Department of Computer Science, University of Nebraska, 1990.

[5] Jean-Camille Birget. Positional simulation of two-way automata: proof of a conjecture of R. Kannan and generalizations. *Journal of Computer and System Sciences*, 45:154–179, 1992.

[6] Jean-Camille Birget. State-complexity of finite-state devices, state compressibility and incompressibility. *Mathematical Systems Theory*, 26:237–269, 1993.

[7] Marek Chrobak. Finite automata and unary languages. *Theoretical Computer Science*, 47:149–158, 1986.

[8] David Damanik. *Finite automata with restricted two-way motion.* Master's thesis, J. W. Goethe-Universität Frankfurt, 1996. In german.

[9] Pavol Ďuriš and Zvi Galil. Fooling a two-way automaton *or* one pushdown store is better than one counter for two-way machines. *Theoretical Computer Science*, 21:39–53, 1982.

[10] Roger B. Eggleton and Richard K. Guy. Catalan strikes again! How likely is a function to be convex? *Mathematics Magazine*, 61(4):211–219, 1988.

[11] Michael J. Fischer, Albert R. Meyer, and Arnold L. Rosenberg. Counter machines and counter languages. *Mathematical Systems Theory*, 3:265–283, 1968.

[12] Martin Gardner. Catalan numbers: an integer sequence that materializes in unexpected places. *Scientific American*, 234(6):120–125, June 1976.

[13] Viliam Geffert, Carlo Mereghetti, and Giovanni Pighizzini. Converting two-way nondeterministic unary automata into simpler automata. *Theoretical Computer Science*, 295:189–203, 2003.

[14] Viliam Geffert, Carlo Mereghetti, and Giovanni Pighizzini. Complementing two-way finite automata. In *Proceedings of the International Conference on Developments in Language Theory*, pages 260–271, 2005.

[15] Jonathan Goldstine, Martin Kappes, Chandra M. R. Kintala, Hing Leung, Andreas Malcher, and Detlef Wotschke. Descriptional complexity of machines with limited resources. *Journal of Universal Computer Science*, 8(2):193–234, 2002.

[16] Juris Hartmanis. On the succinctness of different representations of languages. *SIAM Journal of Computing*, 9(1):114–120, 1980.

[17] Juris Hartmanis. On Gödel speed-up and succinctness of language representations. *Theoretical Computer Science*, 26:335–342, 1983.

[18] Juris Hartmanis. On the importance of being Π_2-hard. *Bulletin of the EATCS*, 37:117–127, 1989.

[19] Juris Hartmanis and Theodore P. Baker. Relative succinctness of representations of languages and separation of complexity classes. In *Proceedings of the International Symposium on Mathematical Foundations of Computer Science*, pages 70–88, 1979.

[20] John E. Hopcroft and Jeffrey D. Ullman. *Formal languages and their relation to automata.* Addison-Wesley, Reading, MA, 1969.

[21] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, Reading, MA, 1979.

[22] Juraj Hromkovič and Georg Schnitger. Nondeterminism versus determinism for two-way finite automata: generalizations of Sipser's separation. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, pages 439–451, 2003.

[23] Oscar H. Ibarra. On two-way multihead automata. *Journal of Computer and System Sciences*, 7:28–36, 1973.

[24] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal of Computing*, 17(5):935–938, 1988.

[25] Ravi Kannan. Alternation and the power of nondeterminism. In *Proceedings of the Symposium on the Theory of Computing*, pages 344–346, 1983.

[26] Christos Kapoutsis. From $k+1$ to k heads the descriptive trade-off is non-recursive. In *Proceedings of the Workshop on Descriptional Complexity of Formal Systems*, pages 213–224, 2004.

[27] Christos Kapoutsis. Deterministic moles cannot solve liveness. In *Proceedings of the Workshop on Descriptional Complexity of Formal Systems*, pages 194–205, 2005.

[28] Christos Kapoutsis. Non-recursive trade-offs for two-way machines. *International Journal of Foundations of Computer Science*, 16:943–956, 2005.

[29] Christos Kapoutsis. Removing bidirectionality from nondeterministic finite automata. In *Proceedings of the International Symposium on Mathematical Foundations of Computer Science*, pages 544–555, 2005.

[30] Christos Kapoutsis. Small sweeping 2NFAs are not closed under complement. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, 2006.

[31] Martin Kutrib. On the descriptional power of heads, counters, and pebbles. In *Proceedings of the Workshop on Descriptional Complexity of Formal Systems*, pages 138–149, 2003.

[32] Martin Kutrib. The phenomenon of non-recursive trade-offs. In *Proceedings of the Workshop on Descriptional Complexity of Formal Systems*, pages 83–97, 2004.

[33] Hing Leung. Separating exponentially ambiguous finite automata from polynomially ambiguous finite automata. *SIAM Journal of Computing*, 27(4):1073–1082, 1998.

[34] Hing Leung. Tight lower bounds on the size of sweeping automata. *Journal of Computer and System Sciences*, 63(3):384–393, 2001.

[35] Albert R. Meyer and Michael J. Fischer. Economy of description by automata, grammars, and formal systems. In *Proceedings of the Symposium on Switching and Automata Theory*, pages 188–191, 1971.

[36] Silvio Micali. Two-way deterministic finite automata are exponentially more succinct than sweeping automata. *Information Processing Letters*, 12(2):103–105, 1981.

[37] Pascal Michel. An NP-complete language accepted in linear time by a one-tape Turing machine. *Theoretical Computer Science*, 85(1):205–212, 1991.

[38] Marvin L. Minsky. Recursive unsolvability of Post's problem of "tag" and other topics in theory of Turing machines. *Annals of Mathematics*, 74(3):437–455, 1961.

[39] Marvin L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, Englewood Cliffs, NJ, 1967.

[40] Burkhard Monien. Transformational methods and their application to complexity problems. *Acta Informatica*, 6:95–108, 1976.

[41] Burkhard Monien. Corrigenda: Transformational methods and their application to complexity problems. *Acta Informatica*, 8:383–384, 1977.

[42] Burkhard Monien. Two-way multihead automata over a one-letter alphabet. *RAIRO Informatique Théorique/Theoretical Informatics*, 14(1):67–82, 1980.

[43] Frank R. Moore. On the bounds for state-set size in the proofs of equivalence between deterministic, nondeterministic, and two-way finite automata. *IEEE Transactions on Computers*, 20(10):1211–1214, 1971.

[44] G. Ott. On multipath automata I. Research report 69, SRRC, 1964.

[45] Michael O. Rabin. Two-way finite automata. In *Proceedings of the Summer Institute of Symbolic Logic*, pages 366–369, Cornell, 1957.

[46] Michael O. Rabin and Dana Scott. Remarks on finite automata. In *Proceedings of the Summer Institute of Symbolic Logic*, pages 106–112, Cornell, 1957.

[47] Michael O. Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:114–125, 1959.

[48] William J. Sakoda and Michael Sipser. Nondeterminism and the size of two-way finite automata. In *Proceedings of the Symposium on the Theory of Computing*, pages 275–286, 1978.

[49] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4:177–192, 1970.

[50] Erik M. Schmidt and Thomas G. Szymanski. Succinctness of descriptions of unambiguous context-free languages. *SIAM Journal of Computing*, 6(3):547–553, 1977.

[51] Joel I. Seiferas. Manuscript communicated to Michael Sipser. October 1973.

[52] Joel I. Seiferas. Relating refined space complexity classes. *Journal of Computer and System Sciences*, 14(1):100–129, 1977.

[53] John C. Shepherdson. The reduction of two-way automata to one-way automata. *IBM Journal of Research and Development*, 3:198–200, 1959.

[54] Michael Sipser. Halting space-bounded computations. *Theoretical Computer Science*, 10:335–338, 1980.

[55] Michael Sipser. Lower bounds on the size of sweeping automata. *Journal of Computer and System Sciences*, 21(2):195–202, 1980.

[56] Richard E. Stearns. A regularity test for pushdown machines. *Information and Control*, 11:323–340, 1967.

[57] Ivan H. Sudborough. On tape-bounded complexity classes and multihead finite automata. *Journal of Computer and System Sciences*, 10(1):62–76, 1975.

[58] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[59] Leslie G. Valiant. A note on the succinctness of descriptions of deterministic languages. *Information and Control*, 32:139–145, 1976.

[60] Moshe Y. Vardi. A note on the reduction of two-way automata to one-way automata. *Information Processing Letters*, 30:261–264, 1989.

[61] Hao Wang. A variant of Turing's theory of computing machines. *Journal of the ACM*, 4(1):63–92, 1957.