A Pumping Lemma for Two-Way Finite Transducers

Tim Smith

Northeastern University
Boston, MA, USA
smithtim@ccs.neu.edu

Abstract. A two-way nondeterministic finite transducer (2-NFT) is a finite automaton with a two-way input tape and a one-way output tape. The generated language of a 2-NFT is the set of all strings it can output (across all inputs). Whereas two-way nondeterministic finite acceptors (2-NFAs) accept only regular languages, 2-NFTs can generate languages which are not even context-free, e.g. $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$. We prove a pumping lemma for 2-NFT languages which strengthens and generalizes previous results. Our pumping lemma states that every 2-NFT language L is k-iterative for some $k \geq 1$. That is, every string in L above a certain length can be expressed in the form $x_1y_1x_2y_2 \cdots x_ky_kx_{k+1}$, where the ys can be "pumped" to produce new strings in L of the form $x_1y_1^ix_2y_2^i \cdots x_ky_k^ix_{k+1}$.

1 Introduction

A pumping lemma for a language class C is a powerful tool for proving that certain languages do not belong to C, and thus for separating one language class from another. The pumping lemmas for regular and context-free languages are well-known, and pumping lemmas have been proved for other language classes as well. These lemmas differ in their specifics, but have in common that they subject elements of the language to an iterated pumping operation which yields new elements in the language. Pumping lemmas come in two strengths: universal and existential. A universal pumping lemma states that all but finitely many elements of the language can be pumped, whereas an existential pumping lemma guarantees only that some element in the language can be pumped. The broader the class of languages, the harder it is to provide it with a universal pumping lemma, and the more intricate the pumping operation becomes.

In this paper we prove a universal pumping lemma for two-way finite transducers. A two-way nondeterministic finite transducer (2-NFT) is a finite automaton with a two-way input tape and a one-way output tape. At each step, the machine can read the symbol under its input head, move its input head left or right, change state, and append a string to the output tape. The final content of the output tape is the output of the computation. Early studies of two-way finite transducers include [1] and [12]; for a survey of early results, see [2].

More recent work connects two-way finite transducers to monadic second-order (MSO) logic [3, 4] and explores the relationship between nondeterministic and sequential transducers [15] and between two-way and one-way transducers [5].

As a transducer, a 2-NFT M can be viewed as a relation of input strings to output strings, and thus as an operation which maps languages to other languages. In addition to its role as a transducer, M can also be treated as a language generator, generating a single language $L(M)$, consisting of all strings it can output (across all inputs). We call $L(M)$ the generated language or range of M. We denote the class of 2-NFT generated languages by $\mathscr{L}$(2-NFT), and the deterministic restriction by $\mathscr{L}$(2-DFT).

It is well known that for finite-state acceptors, neither nondeterminism nor two-way input allows recognition of any additional languages, so that we have:

$$\text{1-DFA} = \text{1-NFA} = \text{2-DFA} = \text{2-NFA} = \text{REG}.$$

For transducers the case is different: 2-DFTs and 2-NFTs can generate languages which are not even context-free, e.g. $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$. In fact, the class $\mathscr{L}$(2-DFT) equals APL, the languages generated by the absolutely parallel grammars of [12], while $\mathscr{L}$(2-NFT) equals 1-NCSA, the languages recognized by one-way nondeterministic checking stack automata [12]. $\mathscr{L}$(2-NFT) also equals 2-NFT(REG), the images of regular languages under 2-NFT transductions. Our pumping lemma, which we prove for $\mathscr{L}$(2-NFT), therefore holds for all of the classes $\mathscr{L}$(2-NFT) = 1-NCSA = 2-NFT(REG) and $\mathscr{L}$(2-DFT) = APL.

The pumping operation we use involves the notion of k-iterativity [6]. For $k \geq 1$, a string s is **k-iterative for a language** L if $s = x_1y_1x_2y_2 \cdots x_ky_kx_{k+1}$ for some strings $x_1, \ldots, x_{k+1}, y_1, \ldots, y_k$, and $\{x_1y_1^ix_2y_2^i \cdots x_ky_k^ix_{k+1} \mid i \geq 0\}$ is an infinite subset of L. (The condition that the subset be infinite is equivalent to requiring that $y_1 \cdots y_k$ is not empty.) Notice that if s is k-iterative for L, then s is i-iterative for L for all $i \geq k$. For $k \geq 1$, a language L is **k-iterative** if there is a $c \geq 0$ such that for every $s \in L$ where $|s| > c$, s is k-iterative for L. L is **weakly k-iterative** if either L is finite, or some string is k-iterative for L. Notice that every regular language is 1-iterative and every context-free language is 2-iterative, due to the pumping lemmas for those classes.

In this paper we show that every language in $\mathscr{L}$(2-NFT) is k-iterative for some $k \geq 1$. Our work strengthens and generalizes the following results, which were proved for checking stack automata or related models but which apply to two-way transducers through the equivalence $\mathscr{L}$(2-NFT) = 1-NCSA.

(a) Greibach [8, Lemma 2.1] showed that every language in 1-NCSA over a single-letter alphabet is 1-iterative.
(b) Rodriguez [13] showed that every reversal-bounded 1-NCSA language is k-iterative for some $k \geq 1$.
(c) As observed in [14, Lemma 3], the results of Greibach [6] imply that every language in 1-NCSA is weakly k-iterative for some $k \geq 1$.

Our result generalizes (a) to alphabets with multiple letters, extends (b) to automata which are not reversal-bounded, and strengthens (c) to k-iterativity instead of weak k-iterativity.

1.1 Proof techniques

A useful tool for analyzing the behavior of an automaton on a two-way tape is the notion of a "visiting sequence". For each square of the tape, the visiting sequence at that square is the sequence of states in which the square is visited during the computation. In a pumping argument, one shows that either the machine visits the same input square twice in the same state, in which case the intervening computation can be repeated, or else two squares have the same visiting sequence, in which case the region between them can be pumped.

We extend this argument to work with two-way finite transducers. Here, it is not enough to show that the input can be pumped to yield new accepting computations; it is also necessary that the outputs of those computations exhibit a k-iterative pattern. In particular, it is necessary to deal with zigzags (repeated changes of direction) in the computation path, which tend to fragment the pumped output. We do so by finding regions of the computation with small zigzags, occurring near positions of the input string with matching "neighborhoods" of surrounding input symbols. Zigzags at these positions stay within their neighborhoods, allowing pumping to proceed with a k-iterative output pattern.

1.2 Related work

The classic paper of Rabin and Scott [10] presented a technique of zigzag elimination to show the equivalence of two-way and one-way finite acceptors. From an original two-way automaton they define a new derived automaton which performs fewer zigzags, and then repeat this derivation operation until a one-way automaton is obtained. Recent work of Filiot et al. [5] extends Rabin and Scott's proof to a subset of nondeterministic transducers called functional transducers, in order to build a one-way functional transducer from a two-way functional transducer whenever one exists. In the present work we take a different approach: instead of eliminating zigzags, we locate regions of the computation with small zigzags and identical neighborhoods of surrounding input symbols, so that zigzags can occur within these neighborhoods without disrupting our pumping operation.

Greibach [7] defines a notion of strong k-iterativity, which goes beyond k-iterativity by allowing certain positions of a string to be designated as distinguished in the pumping operation. Greibach shows that a certain language L in 2-DFT(REG) is not strongly k-iterative for any $k \geq 1$ (Lemma 5.4 of [7] and its corollary). It is not difficult to show that this particular language L is nonetheless k-iterative for some $k \geq 1$, in accordance with our main result.

The class of languages MCFL generated by multiple context-free grammars, a generalization of context-free grammars, has been studied in connection with k-iterativity. As with $\mathscr{L}$(2-NFT), it was known that every MCFL is weakly k-iterative for some $k \geq 1$, but whether k-iterativity held was not known. Recent work resolves this question, showing that in fact there is an MCFL which is not k-iterative for any k [9].

1.3 Outline of paper

The paper is organized as follows. Section 2 gives preliminary definitions concerning two-way finite transducers. Section 3 gives a framework for a pumping argument and explains the challenges to be overcome in applying it to transducers. Section 4 proves our pumping lemma for $\mathscr{L}$(2-NFT). Section 5 provides an application of the pumping lemma. Section 6 gives our conclusions.

2 Preliminaries

An **alphabet** A is a finite set of symbols. A **string** x is an element of A^*. The length of x is denoted by $|x|$. We denote the empty string by λ. For $1 \leq i \leq |x|$, $x[i]$ denotes the ith symbol of x. A **language** is a subset of A^*.

A **two-way nondeterministic finite transducer (2-NFT)** is a tuple $M = (Q, A, B, P, q_{in}, q_{out})$ where Q is a finite set of states, A is the input alphabet, B is the output alphabet, $q_{in}, q_{out} \in Q$ are the initial and final states, respectively, and P is a finite subset of $(Q - \{q_{out}\}) \times (A \cup \{\triangleright, \triangleleft\}) \times B^* \times Q \times \{-1, 0, 1\}$. The symbols $\triangleright$ and $\triangleleft$ are the left and right endmarkers, respectively.

A **step** I of M is a tuple $(q, \triangleright x \triangleleft, y, i)$ for $q \in Q$, $x \in A^*$, $y \in B^*$, and i an integer. We call I a **visit** of i. We write $(q, \triangleright x \triangleleft, y, i) \vdash (q', \triangleright x \triangleleft, ys, i+j)$ if $1 \leq i \leq |\triangleright x \triangleleft|$ and $(q, (\triangleright x \triangleleft)[i], s, q', j)$ is in P. For $n \geq 1$, an **accepting computation** C with **input** x and **output** y is a sequence of steps $I_1 \vdash I_2 \vdash \cdots \vdash I_n$ where $I_1 = (q_{in}, \triangleright x \triangleleft, \lambda, 1)$ and $I_n = (q_{out}, \triangleright x \triangleleft, y, i)$ for some i. By $|C|$ we mean the number of steps in C and by $C[i]$ we mean the ith step of C.

We call M **returning** if for every accepting computation C, the last step of C has the form $(q_{out}, \triangleright x \triangleleft, y, 1)$. Clearly for every 2-NFT M, there is a returning 2-NFT M' such that $L(M) = L(M')$. (Whenever M would enter q_{out}, M' first moves to the left endmarker, and then enters q_{out}.)

We define 2-NFT transductions over strings, languages, and families of languages. For a string x, let $M(x) = \{y \mid M \text{ has an accepting computation with input } x \text{ and output } y\}$. For a language L, let $M(L) = \{y \in M(x) \mid x \text{ is in } L\}$. For a family of languages $\mathscr{L}$, let 2-NFT($\mathscr{L}$) $= \{M(L) \mid M \text{ is a 2-NFT and } L \text{ is in } \mathscr{L}\}$.

2-NFTs can also be viewed as language generators. Let $L(M) = M(A^*)$. $L(M)$ is called the "generated language", or "range" of M. Let $\mathscr{L}$(2-NFT) $= \{L(M) \mid M \text{ is a 2-NFT}\}$. Clearly $\mathscr{L}$(2-NFT) = 2-NFT(REG), since the finite-state control of a 2-NFT can be used to check whether or not an input word in A^* is in some particular regular language L.

3 Pumping framework

In this section we give a framework for a pumping argument on a two-way tape and explain the challenges to be overcome in applying it to transducers. We keep the discussion at a high level, giving a more formal treatment in Section 4.

A useful tool for analyzing the behavior of an automaton on a two-way tape is the notion of a visiting sequence. For each square of the tape, the visiting sequence at that square is the sequence of states in which the square is visited during the computation. In a pumping argument, one shows that either the machine visits the same input square twice in the same state, in which case the intervening computation can be repeated, or else two squares have the same visiting sequence, in which case the region between them can be pumped. By choosing the original computation to be a shortest computation for its output string, we ensure that the pumped portion of the path has non-empty output, and therefore that the pumping operation produces an infinity of new strings.

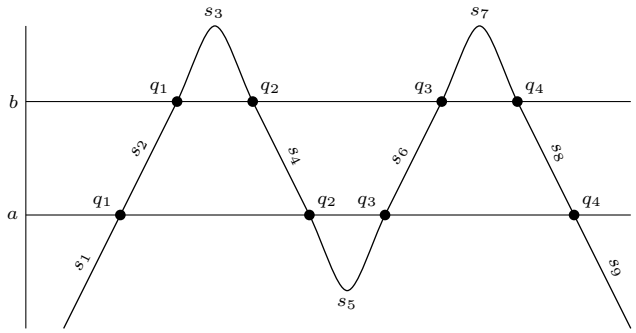

Fig. 1.

Let us apply this basic idea to a 2-NFT M. Consider a shortest accepting computation C (i.e., one with fewest steps, and in the case of a tie, with shortest input) for an output string s. Suppose C has the form shown in Figure 1. The input tape is represented as a vertical line whose bottom corresponds to the left end of the tape and whose top corresponds to the right end. The steps of the computation are depicted as a path winding back and forth along the input tape. Each s_i designates the output of the machine during some portion of the computation, and each q_i designates the state of the machine at a particular point. Thus the output of this computation is the string $s = s_1s_2s_3s_4s_5s_6s_7s_8s_9$. The two input positions a and b can be seen to have the same visiting sequence (q_1, q_2, q_3, q_4). This means that the input region r which separates a and b can be removed, yielding a computation C_0 with output $s_1s_3s_5s_7s_9$. Alternatively, r can be duplicated, yielding a computation C_2 with output $s_1s_2^2s_3s_4^2s_5s_6^2s_7s_8^2s_9$. In general, with $i \geq 0$ copies of region r, we can obtain a computation C_i with output $s_1s_2^is_3s_4^is_5s_6^is_7s_8^is_9$. Suppose $s_2s_4s_6s_8 = \lambda$. Then $s = s_1s_3s_5s_7s_9$. But then C_0 is a shorter computation for s than C, a contradiction. Therefore $|s_2s_4s_6s_8| \geq 1$, making s 4-iterative for $L(M)$.

Problems arise, however, if C instead has a form which zigzags through crossings of r, as in Figure 2. Here, a and b still have the same visiting sequence, so we can still remove or duplicate r and complete the computation, but the new output strings will not have the form needed to make the original string s k-iterative for $L(M)$. For example, if we remove r, we can complete the computation by

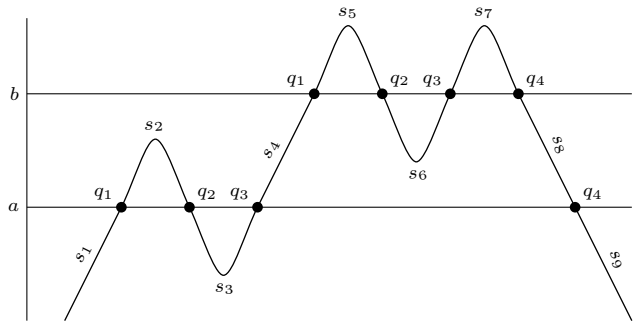

Fig. 2.

skipping from q_1 at a to q_1 at b, from q_2 at b back to q_2 at a, from q_3 at a to q_3 at b, and finally from q_4 at b to q_4 at a. We thereby obtain the output string $s_1s_5s_3s_7s_9$. But in this string, s_5 precedes s_3, whereas in s, s_3 precedes s_5. The new string therefore does not have the right form for showing that s is k-iterative for $L(M)$. Thus the pumping argument does not go through as it stands.

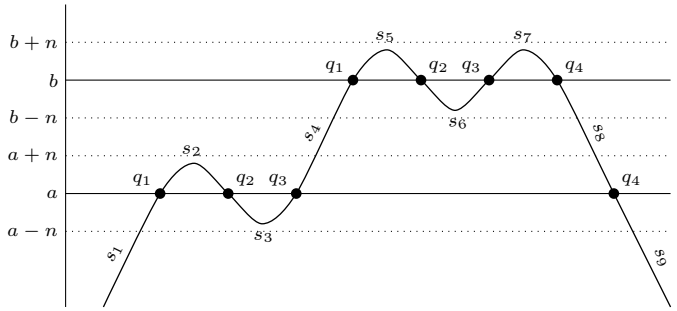

Fig. 3.

To resolve this issue, we will find input positions a and b which not only have appropriate visiting sequences, but also have matching "neighborhoods" of surrounding input symbols large enough to encompass any problematic zigzags. The zigzags in these neighborhoods can then be retained for the pumping and shrinking operations. For example, in Figure 3, suppose the input region from $a-n$ to $a+n$ is identical to the input region from $b-n$ to $b+n$; that is, a and b have the same neighborhood out to n symbols above and below. Now when we remove r, we do not have to skip from q_2 at b back to q_2 at a as before, but can proceed from q_2 at b to q_3 at b, outputting s_6. This is possible because the s_6 portion of the computation never leaves the lower neighborhood of b, so with r removed, it will never leave the lower neighborhood of a, and these two neighborhoods are the same. We thus obtain a computation C_0 with output $s_1s_5s_6s_7s_9$. If we duplicate r, then since the s_3 portion of the computation never leaves the lower neighborhood of a, we can repeat the segments from q_1 at a to q_1 at b and from q_4 at b to q_4 at a, obtaining the output $s_1(s_2s_3s_4)^2s_5s_6s_7s_8^2s_9$. In general, with

$i \geq 0$ copies of region r, we can obtain the output $s_1(s_2s_3s_4)^i s_5s_6s_7s_8^i s_9$. Now suppose $s_2s_3s_4s_8 = \lambda$. Then $s = s_1s_5s_6s_7s_9$ and C_0 is a shorter computation for s than C, a contradiction. Hence $|s_2s_3s_4s_8| \geq 1$, making s 2-iterative for $L(M)$.

In Section 4 we give the details of this argument, showing that if the original output string is sufficiently long, then there is always a region with visiting sequences and surrounding neighborhoods fit for shrinking and pumping in a form which yields k-iterativity. This will allow us to prove our pumping lemma for $\mathscr{L}$(2-NFT).

4 Pumping lemma

In this section we prove our pumping lemma for $\mathscr{L}$(2-NFT), formalizing the high-level approach outlined in Section 3.

Theorem 1. *Suppose L is in $\mathscr{L}$(2-NFT). Then L is k-iterative for some $k \geq 1$.*

Proof. We begin with some definitions. Take $M = (Q, A, B, P, q_{in}, q_{out})$ to be a returning 2-NFT such that $L(M) = L$. We define a function f over the integers from 1 to $\lfloor \frac{|Q|}{2} \rfloor + 1$, as follows.

$$f(i) = \begin{cases} 1 & : i = \lfloor \frac{|Q|}{2} \rfloor + 1 \\ |A|^{2f(i+1)+1} \cdot |Q|^{2i} + 2f(i+1) + 2 & : 1 \leq i \leq \lfloor \frac{|Q|}{2} \rfloor \end{cases}$$

Notice that for $1 \leq i \leq \lfloor \frac{|Q|}{2} \rfloor$, $f(i) > f(i+1)$. Let $k = |Q|$. If L is finite, then trivially L is k-iterative. So say L is infinite. Let r be the highest i such that for some $(q, d, s, q', j) \in P$, $|s| = i$. This is the length of the longest string that M can add to its output in a single step. Let $c = (f(1) + 2) \cdot |Q| \cdot r + r$. Take any $y \in L$ such that $|y| > c$. We will show that y is k-iterative for L.

Let a shortest computation for y be an accepting computation C with output y such that for every accepting computation C' with output y, $|C| \leq |C'|$ and if $|C| = |C'|$, then $|x| \leq |x'|$, where x is the input of C and x' is the input of C'. Take any shortest computation C for y. Let x be the input of C.

Each step i of the computation C has the form $(q_i, \triangleright x \triangleleft, s_i, v_i)$. We will view C as a path and each step of C as a **node** on the path. We call each position on the input tape from 1 to $|\triangleright x \triangleleft|$ a **level** and we call v_i the level at node i. We have $v_1 = v_{|C|} = 1$; that is, the path starts at level 1 (the left endmarker) and also ends at level 1 (since M is returning). The level of each node differs from that of its predecessor by at most 1.

For $1 \leq i \leq j \leq |C|$, we call the sequence $C[i], \ldots, C[j]$ a subpath from i to j. A **hill** h **at level** l is a subpath from i to j of length > 2 such that $v_i = v_j = l$ and for all m such that $i < m < j$, $v_m > l$. The **top** of h is $\max(v_i, \ldots, v_j)$ and the **height** of h is $\max(v_i, \ldots, v_j) - l$. A **valley** v **at level** l is a subpath from i to j of length > 2 such that $v_i = v_j = l$ and for all m such that $i < m < j$, $v_m < l$. The **bottom** of v is $\min(v_i, .., v_j)$ and the **depth** of v is $l - \min(v_i, .., v_j)$.

With these definitions in place, the proof idea is as follows. If some level has more than $|Q|$ visits, we will see that y is 1-iterative for L. Otherwise, the input

string x is long enough that it contains a "smooth" region (l to $l + f(n)$ below), one with small hills and valleys. Within this region we find two levels a and b with similar neighborhoods and visiting sequences. The smoothness of the region then permits a pumping argument applied to the area between a and b to show that y is k-iterative for L.

So suppose some level has more than $|Q|$ visits. Then M visits the same input position twice in the same state. Suppose M produces no output between these two visits. Then C could be shortened, a contradiction. So between the two visits M produces some non-empty output w. Then we can "pump" (repeat) the intervening computation. Hence $y = uwz$ for some strings u and z, and $\{uw^iz \mid i \geq 0\}$ is an infinite subset of L. Then y is 1-iterative for L, hence k-iterative for L. So say no level has more than $|Q|$ visits. Notice that we now have $|Q| \geq 2$, since if $|Q| = 1$, then M can only visit the left endmarker once, so since C must begin and end at the left endmarker, we have $|C| = 1$ and $y = \lambda$, which contradicts the fact that $|y| > c$.

Since no level has more than $|Q|$ visits, $|\triangleright x \triangleleft| \geq \frac{|y|}{|Q|\cdot r}$. Hence $|x| \geq \frac{|y|}{|Q|\cdot r} - 2 > \frac{c}{|Q|\cdot r} - 2 = \frac{(f(1)+2)\cdot|Q|\cdot r + r}{|Q|\cdot r} - 2 > f(1)$. Every position of x is visited at least once, otherwise x could be shortened and C would still output y, a contradiction. Therefore level 1 has a hill of height $> f(1)$. So take the highest $n \leq \lfloor \frac{|Q|}{2} \rfloor$ such that some level has $\geq n$ hills of height $> f(n)$. We have $1 \leq n \leq \lfloor \frac{|Q|}{2} \rfloor$ and $f(n) > f(n+1) \geq 1$. Notice that no level i has $\geq n+1$ hills of height $> f(n+1)$, since if $n < \lfloor \frac{|Q|}{2} \rfloor$, then this would contradict the construction of n, and if $n = \lfloor \frac{|Q|}{2} \rfloor$, then since level $i+1$ has at least two visits for each hill of height > 1 at level i, level $i+1$ would have more than $|Q|$ visits, a contradiction. So take any level l with $\geq n$ hills of height $> f(n)$. Then since $f(n) > f(n+1)$, l must have exactly n hills of height $> f(n)$. For the same reason, level l cannot have a hill whose height is $> f(n+1)$ but $\leq f(n)$.

Further, suppose some level $i \leq l + f(n)$ has a valley v of depth $> f(n+1)$ whose bottom is above l. Consider the n hills of height $> f(n)$ at level l. Since the bottom of v is above l, either v is contained completely by one of these n hills, or it is not part of any of them. If it is not part of any of them, then it is part of another hill at level l, but then level l has $\geq n+1$ hills of height $> f(n+1)$, a contradiction. If v is contained completely by one of the n hills of height $> f(n)$ at level l, then this hill contains two hills of height $> f(n+1)$ at level $i - f(n+1)$ (one on each side of v). But then level $i - f(n+1)$ contains $\geq n+1$ hills of height $> f(n+1)$, a contradiction. So there is no such level i.

We will refer to the n hills at level l which are of height $> f(n)$ as hill 1, ..., hill n. For any level i from $l + f(n+1)$ to $l + f(n) - f(n+1)$, call $x[i - f(n+1)] \cdots x[i + f(n+1)]$ the **neighborhood** of i. For j from 1 to n, let $\text{in}(i, j)$ be the first node at level i in hill j, and let $\text{out}(i, j)$ be the last node at level i in hill j. Recall that q_m is the state of M at step m of C. Let the **pair list** of i be a list of n pairs of states such that for $1 \leq j \leq n$, the jth pair is $(q_{\text{in}(i,j)}, q_{\text{out}(i,j)})$.

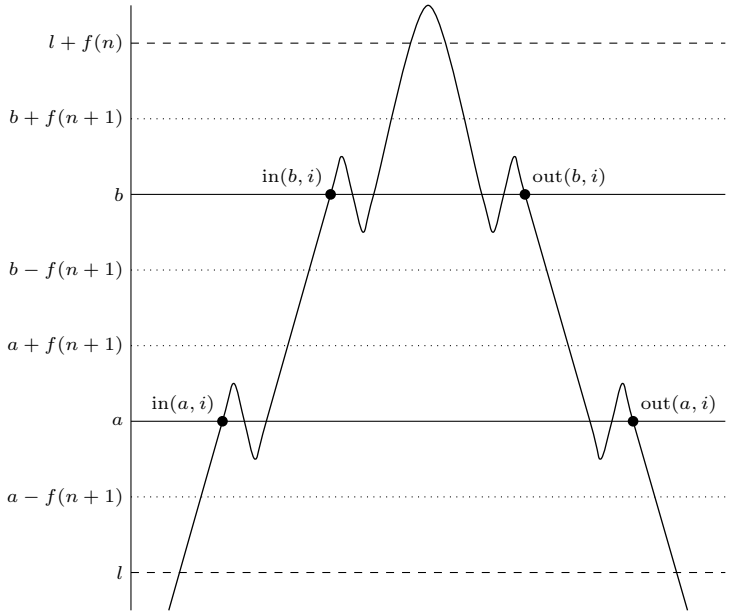

Fig. 4. An example hill i at level l.

There are at most $p_1 = |A|^{2f(n+1)+1}$ distinct neighborhoods, and at most $p_2 = |Q|^{2n}$ distinct pair lists. From $l + f(n+1) + 1$ to $l + f(n) - f(n+1) - 1$ there are $f(n) - 2f(n+1) - 1$ levels. Then since $f(n) - 2f(n+1) - 1 > p_1 p_2$, there are levels a, b such that $l + f(n+1) < a < b < l + f(n) - f(n+1)$ and a and b have the same neighborhood and pair list. See Figure 4 for an example hill i passing through levels a and b. We make some observations O1, O2, O3 for use below.

O1. For any hill i such that $1 \leq i < n$, the portion of C between $\text{out}(a, i)$ and $\text{in}(a, i+1)$ never reaches level a, since if it did so as part of hill i, then $\text{out}(a, i)$ would not be the last node at level a in hill i, if it did so as part of hill $i+1$, then $\text{in}(a, i+1)$ would not be the first node at level a in hill $i+1$, and if it did so between hills i and $i+1$, then level l would have a hill whose height is $> f(n+1)$ but $\leq f(n)$, which we observed to be impossible. Similarly, the portion of C before $\text{in}(a, 1)$ never reaches level a, and the portion of C after $\text{out}(a, n)$ never reaches level a.

O2. For any hill i, the portion of C from $\text{in}(a, i)$ to $\text{out}(a, i)$ never goes below the neighborhood of a, since if it did, then a would have a valley of depth $> f(n+1)$ whose bottom is above l (since this portion of the path is in a hill of l), which we observed to be impossible.

O3. Similarly, for any hill i, the portion of C from $\text{in}(b, i)$ to $\text{out}(b, i)$ never goes below the neighborhood of b.

We now break up y into substrings, where each substring designates the output produced during a range of steps of C. Recall that s_i is the output produced

from step i of C. Let $h_0 = s_1 \cdots s_{\text{in}(a,1)-1}$ and for i from 1 to n, let:

$$e_i = s_{\text{in}(a,i)} \cdots s_{\text{in}(b,i)-1}$$
$$f_i = s_{\text{in}(b,i)} \cdots s_{\text{out}(b,i)-1}$$
$$g_i = s_{\text{out}(b,i)} \cdots s_{\text{out}(a,i)-1}$$
$$h_i = \begin{cases} s_{\text{out}(a,i)} \cdots s_{\text{in}(a,i+1)-1} & \text{if } 1 \leq i < n \\ s_{\text{out}(a,i)} \cdots s_{|C|} & \text{if } i = n \end{cases}$$

We have $y = h_0\ e_1 f_1 g_1 h_1 \cdots e_n f_n g_n h_n$. For $i \geq 0$, let $y_i = h_0\ e_1^i f_1 g_1^i h_1 \cdots e_n^i f_n g_n^i h_n$. We will show that $\{y_i \mid i \geq 0\}$ is an infinite subset of L, and thus that $y\ (= y_1)$ is $2n$-iterative for L.

First we show that y can be "shrunk", i.e. that y_0 is in L. We construct a computation C_0 with input $x[1] \cdots x[a-1]\ x[b] \cdots x[|x|]$ which will follow C, but skip some steps. C_0 proceeds as follows. Follow C until $\text{in}(a,1)$, outputting h_0. This is possible due to O1. For i from 1 to n, continue as follows. Skip to $\text{in}(b,i)$, which is possible because $q_{\text{in}(a,i)} = q_{\text{in}(b,i)}$. Proceed from $\text{in}(b,i)$ to $\text{out}(b,i)$, outputting f_i. This is possible because due to O3, in C this portion never went below the neighborhood of b, so in C_0 it will never go below the neighborhood of a, and these two neighborhoods are equal. Skip from $\text{out}(b,i)$ to $\text{out}(a,i)$. This is possible because $q_{\text{out}(b,i)} = q_{\text{out}(a,i)}$. If $i < n$, proceed from $\text{out}(a,i)$ to $\text{in}(a,i+1)$, outputting h_i (possible due to O1). If $i = n$, proceed from $\text{out}(a,n)$ to the end of C, outputting h_n and finishing in the final state q_{out} (possible due to O1). C_0 is now an accepting computation with output y_0, which is therefore in L.

Next we show that y can be "pumped", i.e. that y_m is in L for each $m \geq 2$. So take any $m \geq 2$. We show how to construct a computation C_m with input $x[1] \cdots x[a-1]\ (x[a] \cdots x[b-1])^m\ x[b] \cdots x[|x|]$ which will follow C, but repeat some steps. C_m proceeds as follows. Follow C until $\text{in}(a,1)$, outputting h_0. This is possible due to O1. For i from 1 to n, continue as follows. Perform the portion of C from $\text{in}(a,i)$ to $\text{in}(b,i)$ m times, each time outputting e_i. This is possible because $q_{\text{in}(a,i)} = q_{\text{in}(b,i)}$, and because due to O2, in C this portion never goes below the neighborhood of a (or above level b, since $\text{in}(b,i)$ is the first node at level b in hill i), and in C_m, for j from 1 to m, the neighborhood of $a + j(b-a)$ equals the neighborhood of a. Now C_m is at level $a + m(b-a)$. Proceed from $\text{in}(b,i)$ to $\text{out}(b,i)$, outputting f_i. This is possible due to O3. Next, perform the portion of C from $\text{out}(b,i)$ to $\text{out}(a,i)$ m times, each time outputting g_i. This is possible because $q_{\text{out}(b,i)} = q_{\text{out}(a,i)}$, and because in C this portion never goes above level b (since $\text{out}(b,i)$ is the last node at level b in hill i) or below the neighborhood of a (due to O2). Now if $i < n$, proceed from $\text{out}(a,i)$ to $\text{in}(a,i+1)$, outputting h_i (possible due to O1). If $i = n$, proceed from $\text{out}(a,n)$ to the end of C, outputting h_n and finishing in the final state q_{out} (possible due to O1). C_m is now an accepting computation with output y_m, which is therefore in L. Hence $\{y_i \mid i \geq 2\}$ is a subset of L.

Finally, suppose $e_1 g_1 \cdots e_n g_n = \lambda$. Then $y_0 = y$ and C_0 is an accepting computation with output y, a contradiction, since $|C_0| < |C|$ and C is a shortest computation for y. So $e_1 g_1 \cdots e_n g_n \neq \lambda$. Then $\{y_i \mid i \geq 0\}$ is an infinite language. Hence $\{y_i \mid i \geq 0\}$ is an infinite subset of L. Therefore y is $2n$-iterative for L. Then since $n \leq \lfloor \frac{|Q|}{2} \rfloor$ and $k = |Q|$, we have $2n \leq k$, so y is k-iterative for L. So

for any $y \in L$ such that $|y| > c$, y is k-iterative for L. Hence L is k-iterative, which was to be shown. □

5 Application

In this section we apply our pumping lemma to show that a particular language of interest does not belong to $\mathscr{L}$(2-NFT). Addressing the question of whether a certain type of "mildly context-sensitive" grammars can generate the class of natural languages, Radzinksi [11] considers the system of Chinese number-names. In particular, he examines the set L consisting of number-names composed only of instances of *wu* (five) and *zhao* (trillion):

$$L = \{\texttt{wu (zhao)}^{k_1} \texttt{ wu (zhao)}^{k_2} \cdots \texttt{ wu (zhao)}^{k_n} \mid k_1 > k_2 > \cdots k_n > 0\}$$

Radzinski shows that L cannot be generated by the class TAG of tree adjoining grammars; we will show that it also cannot be generated by a 2-NFT. The string `wu zhao zhao` is 1-iterative for L, since $\{\texttt{wu zhao (zhao)}^i \mid i \geq 0\}$ is an infinite subset of L. The language L is therefore weakly 1-iterative. At first glance it might seem that L is also 1-iterative, since we can pump the first `zhao` in any string. For example, for the string `wu zhao zhao wu zhao` we have $\{\texttt{wu (zhao)}^i \texttt{ zhao wu zhao} \mid i \geq 1\}$, which is an infinite subset of L. But recall that k-iterativity requires the pumping index i to start at 0, not 1, and $\{\texttt{wu (zhao)}^i \texttt{ zhao wu zhao} \mid i \geq 0\}$ is not a subset of L. In fact, Radzinski shows that a related language (K below) is not k-iterative for any $k \geq 1$. Our pumping lemma then gives the following result.

Theorem 2. *L is not in $\mathscr{L}$(2-NFT).*

Proof. Let K be the language $\{\texttt{a b}^{k_1} \texttt{ a b}^{k_2} \cdots \texttt{ a b}^{k_n} \mid k_1 > k_2 > \cdots k_n > 0\}$. The proof of Lemma 2 of Radzinski [11] shows that K is not k-iterative for any $k \geq 1$. Then by our Theorem 1, K is not in $\mathscr{L}$(2-NFT). Suppose L is in $\mathscr{L}$(2-NFT). Let h be a homomorphism from $\{\texttt{w}, \texttt{u}, \texttt{z}, \texttt{h}, \texttt{a}, \texttt{o}\}^*$ to $\{\texttt{a}, \texttt{b}\}^*$ such that $h(\texttt{wu}) = \texttt{a}$ and $h(\texttt{zhao}) = \texttt{b}$. Then K is the image of L under h. By Lemma 1.1 of [8], $\mathscr{L}$(2-NFT) is closed under substitution. Then K is in $\mathscr{L}$(2-NFT), a contradiction. So L is not in $\mathscr{L}$(2-NFT). □

6 Conclusion

In this paper we have proved a pumping lemma for the class $\mathscr{L}$(2-NFT) of languages generated by two-way nondeterministic finite transducers. Our pumping lemma strengthens and generalizes previous results for this class. We gave an example of a language of interest which can be shown using our pumping lemma not to belong to $\mathscr{L}$(2-NFT), and we hope that our lemma will help to obtain similar results in other cases. One direction for further research would be to generalize our lemma to broader classes of languages. For example, recall that $\mathscr{L}$(2-NFT) = 2-NFT(REG). Where CFL denotes the context-free languages, the

class 2-NFT(CFL) properly contains 2-NFT(REG), since 2-NFT(REG) does not contain CFL [6, Theorem 4.26]. We can then ask whether our pumping lemma can be generalized to apply to 2-NFT(CFL). More broadly, it would be interesting to know whether such a lemma holds for all 2-NFT($\mathscr{L}$) where $\mathscr{L}$ is a language class such that every language in $\mathscr{L}$ is k-iterative.

Acknowledgments. I want to thank my advisor, Rajmohan Rajaraman, for supporting this work, encouraging me, and offering many helpful comments and suggestions.

References

1. Ehrich, R., Yau, S.: Two-way sequential transductions and stack automata. Information and Control 18(5), 404 – 446 (1971)
2. Engelfriet, J.: Two-way automata and checking automata. In: de Bakker, J.W., van Leeuwen, J. (eds.) Foundations of Computer Science III Part 1, Mathematical Centre Tracts, vol. 108, pp. 1–69. Mathematisch Centrum, Amsterdam (1979)
3. Engelfriet, J., Hoogeboom, H.J.: MSO definable string transductions and two-way finite-state transducers. ACM Trans. Comput. Logic 2(2), 216–254 (Apr 2001)
4. Engelfriet, J., Hoogeboom, H.J.: Finitary compositions of two-way finite-state transductions. Fundam. Inf. 80(1-3), 111–123 (Jan 2007)
5. Filiot, E., Gauwin, O., Reynier, P.A., Servais, F.: From two-way to one-way finite state transducers. In: LICS 2013. pp. 468–477. IEEE Computer Society (2013)
6. Greibach, S.A.: One way finite visit automata. Theoretical Computer Science 6(2), 175 – 221 (1978)
7. Greibach, S.A.: The strong independence of substitution and homomorphic replication. RAIRO - Theoretical Informatics and Applications 12(3), 213–234 (1978)
8. Greibach, S.: Checking automata and one-way stack languages. J. Comput. Syst. Sci. 3(2), 196–217 (May 1969)
9. Kanazawa, M., Kobele, G., Michaelis, J., Salvati, S., Yoshinaka, R.: The failure of the strong pumping lemma for multiple context-free languages. Theory of Computing Systems pp. 1–29 (2014)
10. Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM J. Res. Dev. 3(2), 114–125 (Apr 1959)
11. Radzinski, D.: Chinese number-names, tree adjoining languages, and mild context-sensitivity. Comput. Linguist. 17(3), 277–299 (Sep 1991)
12. Rajlich, V.: Absolutely parallel grammars and two-way finite-state transducers. J. Comput. Syst. Sci. 6(4), 324–342 (Aug 1972)
13. Rodriguez, F.: Une double hiérarchie infinie de langages vérifiables. RAIRO - Theoretical Informatics and Applications 9(R1), 5–19 (1975)
14. Smith, T.: On Infinite Words Determined by Stack Automata. In: FSTTCS 2013. Leibniz International Proceedings in Informatics (LIPIcs), vol. 24, pp. 413–424. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2013)
15. Souza, R.: Uniformisation of two-way transducers. In: LATA 2013, Lecture Notes in Computer Science, vol. 7810, pp. 547–558. Springer Berlin Heidelberg (2013)