Theory of Computing Systems (2023) 67:956–975
https://doi.org/10.1007/s00224-023-10124-0

Visit-Bounded Stack Automata

Jozef Jirásek[1] · Ian McQuillan[1]

Accepted: 8 May 2023 / Published online: 23 July 2023
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

Abstract
An automaton is *k-visit-bounded* if during any computation its work tape head visits each tape cell at most k times. In this paper we consider stack automata which are k-visit-bounded for some integer k. This restriction resets the visits when popping (unlike similarly defined Turing machine restrictions) which we show allows the model to accept a proper superset of context-free languages and also a proper superset of languages of visit-bounded Turing machines. We study two variants of visit-bounded stack automata: one where only instructions that move the stack head downwards increase the number of visits of the destination cell, and another where any transition increases the number of visits. We prove that the two types of automata recognize the same languages. We then show that all languages recognized by visit-bounded stack automata are effectively semilinear, and hence are letter-equivalent to regular languages, which can be used to show other properties.

Keywords Stack automata · Visit-bounded automata · Semilinear languages

1 Introduction

When introducing a machine model or a grammar system, one of the most useful properties is that of semilinearity. The idea of a language being semilinear is defined formally in Section 2, but equivalently, a language is semilinear if and only if it has the same Parikh image as some regular language [1]. In particular, when this property is effective for a machine model $\mathcal{M}$, there is a procedure to construct a letter-equivalent finite automaton from any such machine. It is well-known due to Parikh that the context-free languages have this property [2]. When this property is effective along

✉ Jozef Jirásek
jirasek.jozef@usask.ca

Ian McQuillan
mcquillan@cs.usask.ca

1 Department of Computer Science, University of Saskatchewan, Saskatoon SK S7N 5A9, Saskatchewan, Canada

with effective closure under homomorphism, inverse homomorphism, and intersection with regular languages (the full trio properties), it immediately implies several useful properties.[1]

1. It provides a procedure to decide emptiness, finiteness, and membership [3].
2. The class can be augmented by reversal-bounded counters and the resulting class is still semilinear [4] — more generally, the smallest full trio (or even full AFL) containing the languages accepted by $\mathcal{M}$ that is also closed under intersection with one-way nondeterministic reversal-bounded multicounter machines [3] is also semilinear. The resulting family has the positive decidable properties of (1).
3. All bounded languages accepted by $\mathcal{M}$ are so-called *bounded semilinear languages* [5], and they can all be accepted by a deterministic machine model, one-way deterministic reversal-bounded multicounter machines [5], where we can decide containment and equivalence of two machines.
4. Properties related to counting functions and slenderness (having at most k strings of each length) can be decided [6].

It is also one of the key properties of a class of grammars being mildly context-sensitive [7], which was developed to encompass the properties that are important for computational linguistics.

Stack automata are a generalization of pushdown automata with the ability to push and pop at the top of the stack, and an added ability to read the contents of the stack in a two-way read-only fashion [8]. They are quite powerful however and can accept non-semilinear languages [9, 10]. Checking stack automata are stack automata that cannot pop, and cannot push after reading from the stack. Here, we consider a restriction on stack automata. Given a subset E of the stack instructions (push, pop, stay, move left, or right), a machine is k-visit$_E$-bounded if, during any computation, its stack head visits each tape cell while performing an instruction of E at most k times; and it is visit$_E$-bounded if it is k-visit$_E$-bounded for some k. We omit E if it contains all instructions.

Importantly in this definition, when a cell is popped from the stack, the count towards this bound disappears with it, and any new symbols pushed start with a count of zero again. This makes the definition in some ways more general than had we defined Turing machines with a visit-bounded worktape. This type of model was studied by Greibach [11], who studied one-way input with a single Turing machine work tape which it can edit (precisely, Greibach defines the machines to be preloaded with a string from a language family such as the regular languages — but as we are restricting our study to regular languages, this preloading does not affect the capacity). Greibach showed that the languages accepted by finite-visit Turing machines are a semilinear subset of the checking stack languages.

Here we show that a stack language is visit-bounded if and only if it is visit$_E$-bounded where E only contains an instruction to move left. We then show that the family of languages accepted by visit-bounded stack automata only contain semilinear

[1] A preliminary version of this paper appeared in the Proceedings of the 26th International Conference on Developments in Language Theory (DLT 2022), Lecture Notes in Computer Science 13257, 189-200, 2022. This version is substantially extended from the previous version to include the complete proof of Theorem 2, and full details of all algorithms used in the proof of Theorem 4, which are central to the work.

languages, in contrast to stack automata generally. Furthermore, they form a language family properly between the context-free and stack languages. Lastly, we show that the class of languages of Turing machines with a finite-visit (or finite-crossing) restriction (and a one-way input tape) is properly contained in the class of languages of finite-visit stack automata (as the former does not contain all context-free languages), demonstrating the power of our model while still preserving semilinearity. This makes the family useful towards showing that other language families are semilinear.

2 Preliminaries

We refer to [1, 12] for an introduction to automata and formal language theory. An *alphabet* Σ is a finite set of *symbols*. A *string* over Σ is a finite sequence of symbols from Σ. The set of all strings over Σ, including the empty string λ, is denoted by Σ^*. A *language* is a subset of Σ^*.

Let w be a string over $\Sigma = \{a_1, a_2, \dots, a_n\}$. The *length* of w, denoted by $|w|$, is the number of characters in w, with $|\lambda| = 0$. For $a \in \Sigma$, the number of occurrences of the character a in the string w is denoted by $|w|_a$. The *Parikh image* of a string w, denoted $\Psi(w)$, is the vector $(|w|_{a_1}, |w|_{a_2}, \dots, |w|_{a_n})$. We note that two strings have the same Parikh image if one is a permutation of the other. For a language $L \subseteq \Sigma^*$, let $\Psi(L) = \{\Psi(w) \mid w \in L\}$. Two languages L_1 and L_2 are *letter-equivalent* if $\Psi(L_1) = \Psi(L_2)$. Equivalently, every string in L_1 is a permutation of some string in L_2, and vice versa.

A subset Q of $\mathbb{N}^m$ (m-tuples) is a *linear set* if there exist $\vec{v_0}, \vec{v_1}, \dots, \vec{v_r} \in \mathbb{N}^m$ such that $Q = \{\vec{v_0} + i_1\vec{v_1} + \cdots + i_r\vec{v_r} \mid i_1, \dots, i_r \in \mathbb{N}\}$. We call $\vec{v_0}$ the constant and $\vec{v_1}, \dots, \vec{v_r}$ the periods. A finite union of linear sets is a *semilinear set*. A language $L \subseteq \Sigma^*$ is semilinear if $\Psi(L)$ is a semilinear set. It is known that a language L is semilinear if and only if there exists a regular language L' with $\Psi(L) = \Psi(L')$ [1]. For a family of languages accepted by a class of machines $\mathcal{M}$, we say that the family is *effectively semilinear* if there is an algorithm to determine the constant and the periods for each linear set (or equivalently the letter-equivalent finite automaton). The following is a classical result in automata theory.

Theorem 1 (Parikh's Theorem [2]) *Let L be a context-free language. Then $\Psi(L)$ is a semilinear set.*

Let NFA be the class of nondeterministic finite automata and NPDA be the class of nondeterministic pushdown automata. Given a class of machines $\mathcal{M}$, let $\mathcal{L}(\mathcal{M})$ be the family of languages accepted by $\mathcal{M}$.

2.1 Stack Automata

A nondeterministic one-way *stack automaton* M is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where:

- Q is the finite set of states,
- Σ and Γ are the input and work tape alphabets,

- Let $I = \{\mathtt{S}, \mathtt{L}, \mathtt{R}, \mathtt{push}(x), \mathtt{pop} \mid x \in \Gamma\}$ be the *instruction set*, then:
- $\delta \subseteq Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{\rhd\}) \times Q \times I$ is the transition relation,
- $q_0 \in Q$ is the initial state, and
- $F \subseteq Q$ is the set of final states.

The special symbol $\rhd$ denotes the left end of the work tape, which is identified with the bottom of the stack.

We define the contents of the stack slightly differently (but equivalently) from previous definitions in order to better capture the restrictions on number of visits. The work tape shall be represented as a series of pairs (x, i), denoting individual tape cells, where $x \in \Gamma \cup \{\rhd\}$ is the symbol written in this cell, and $i \in \mathbb{N}$ is the number of times the automaton has visited this cell. Note that the transition function of the automaton only has access to the symbols written on the tape, and the automaton can not inspect the visit counters of the cells.

A *configuration* of the automaton M is a triple (q, w, γ), where:

- $q \in Q$ is the current state,
- $w \in \Sigma^*$ is the input that is still to be read,
- $\gamma \in (\{\rhd\} \times \mathbb{N})(\Gamma \times \mathbb{N})^* \lrcorner (\Gamma \times \mathbb{N})^*$ is the current content of the work tape. The special symbol $\lrcorner$ denotes the position of the tape head, which is scanning the cell immediately preceding this symbol.

Now let $E \subseteq \{\mathtt{S}, \mathtt{L}, \mathtt{R}, \mathtt{push}, \mathtt{pop}\}$ be a set of *expensive instructions*. These are the instructions that are counted as visits to a tape cell. The automaton performs all instructions on the work tape as usual for stack automata. When an expensive instruction is performed, the number of visits of the tape cell under the head after the instruction is completed is increased by one.

We define the *move relation* $\vdash$ between configurations of M using a set of expensive instructions E as follows: For $\iota \in \{\mathtt{S}, \mathtt{L}, \mathtt{R}, \mathtt{push}, \mathtt{pop}\}$, let the *cost* of ι be $\mathrm{c}(\iota) = 1$ if $\iota \in E$, and $\mathrm{c}(\iota) = 0$ if $\iota \notin E$. Then:

- $(p, aw, \alpha(x, i) \lrcorner \beta) \vdash (q, w, \alpha(x, i + \mathrm{c}(\mathtt{S})) \lrcorner \beta)$
  if $(p, a, x, q, \mathtt{S}) \in \delta$,
- $(p, aw, \alpha(x, i)(y, j) \lrcorner \beta) \vdash (q, w, \alpha(x, i + \mathrm{c}(\mathtt{L})) \lrcorner (y, j)\beta)$
  if $(p, a, x, q, \mathtt{L}) \in \delta$,
- $(p, aw, \alpha(x, i) \lrcorner (y, j)\beta) \vdash (q, w, \alpha(x, i)(y, j + \mathrm{c}(\mathtt{R})) \lrcorner \beta)$
  if $(p, a, x, q, \mathtt{R}) \in \delta$,
- $(p, aw, \alpha(x, i) \lrcorner) \vdash (q, w, \alpha(x, i)(y, \mathrm{c}(\mathtt{push})) \lrcorner)$
  if $(p, a, x, q, \mathtt{push}(y)) \in \delta$, and
- $(p, aw, \alpha(x, i)(y, j) \lrcorner) \vdash (q, w, \alpha(x, i + \mathrm{c}(\mathtt{pop})) \lrcorner)$
  if $(p, a, y, q, \mathtt{pop}) \in \delta$;

where $p, q \in Q$, $a \in \Sigma \cup \{\lambda\}$, $w \in \Sigma^*$, $x \in \Gamma \cup \{\rhd\}$, $y \in \Gamma$, $i, j \in \mathbb{N}$, $\alpha \in \{\lambda\} \cup ((\rhd \times \mathbb{N})(\Gamma \times \mathbb{N})^*)$, $\beta \in (\Gamma \times \mathbb{N})^*$, and the work tape string on both sides of the relation is well-formed (in particular, $x = \rhd$ if and only if $\alpha = \lambda$). Let $\vdash^*$ denote the reflexive and transitive closure of $\vdash$.

A *computation* of a stack automaton M on a string $w \in \Sigma^*$ is a sequence of configurations $c_0 \vdash c_1 \vdash \cdots \vdash c_n$, where $c_0 = (q_0, w, (\rhd, 0) \lrcorner)$, and $c_n = (q_n, \lambda, \gamma_n)$. If $q_n \in F$, this computation is *accepting*. The automaton M *accepts* a string w if there exists an accepting computation of M on w. The *language accepted by* M, denoted by $\mathrm{L}(M)$, is the set of all strings from Σ^* that M accepts.

Let SA be the class of all stack automata. A stack automaton is called a *non-erasing stack automaton* if it uses no $\mathtt{pop}$ instructions. A non-erasing stack automaton is called a *checking stack automaton* if it cannot push more symbols on the stack after the head enters the stack using an $\mathtt{L}$ instruction. The class of non-erasing stack automata is denoted by NESA, and the class of checking stack automata by CSA.

For an integer k and a set of expensive instructions E, we say that a computation of a stack automaton M is *k-visit$_E$-bounded*, if the number of visits of every cell in every configuration in this computation is less than or equal to k. We say that M is *k-visit$_E$-bounded* if for every string $w \in \mathrm{L}(M)$ the automaton M has a k-visit$_E$-bounded accepting computation on w. Finally, M is *visit$_E$-bounded* if there is a finite $k \in \mathbb{N}$ such that M is k-visit$_E$-bounded. Let $\mathsf{VISIT}_E(k)$ be the class of k-visit$_E$-bounded stack automata, and VISIT_E of all visit$_E$-bounded stack automata. If we leave off the subscript E, it is assumed that $E = \{\mathtt{S}, \mathtt{L}, \mathtt{R}, \mathtt{push}, \mathtt{pop}\}$, this means that every instruction is considered expensive. It is immediate that $\mathcal{L}(\mathsf{SA}) = \mathcal{L}(\mathsf{VISIT}_\emptyset)$.

Note the important distinguishing feature of the stack automaton model which sets it apart from visit-bounded Turing machine models previously considered in literature: whenever a tape cell is popped from the top of the stack, the number of visits of that cell is reset. Whenever a new cell is pushed to the top of the stack, this new cell begins with a visit count of 0 (or 1, if $\mathtt{push}$ is an expensive instruction). This allows a visit-bounded stack automaton to perform some computations that an analogous visit-bounded Turing machine could not.

3 Visit-Bounded Automata

As we have seen in definitions in Section 2, the notion of a visit-bounded stack automaton is dependent on the choice of the set of expensive instructions E which increase the visit counters of tape cells. To begin, we consider two expensive instruction sets: $E_1 = \{\mathtt{L}\}$, and $E_2 = \{\mathtt{S}, \mathtt{L}, \mathtt{R}, \mathtt{push}, \mathtt{pop}\}$. In the first case, only $\mathtt{L}$ instructions increase the visit counters. In the second case, all instructions increase the visit counters.

Example 1 *Let $M = (\{q_0\}, \{a\}, \{\}, \{(q_0, a, \rhd, q_0, \mathtt{S})\}, q_0, \{q_0\})$ be a stack automaton. This simple automaton scans its input consisting of a number of symbols a, while the work tape head rests on the bottom of the stack marker.*

Observe that M is visit$_{\{\mathtt{L}\}}$-bounded, as it never performs an $\mathtt{L}$ instruction, and thus the number of visits of the only used tape cell never increases above 0. On the other hand, M is not visit-bounded, as the $\mathtt{S}$ instructions in the only computation of M on string a^k increase the visit counter of the tape cell to k.

Every visit-bounded automaton is also visit$_{\{\mathtt{L}\}}$-bounded. Indeed, the number of visits to a cell can not increase if we only consider a limited subset of expensive instructions. Perhaps surprisingly, as we will show in Theorem 2, the converse is also true if we only consider languages accepted by the automaton. For any visit$_{\{\mathtt{L}\}}$-bounded automaton A, we can construct a visit-bounded automaton B with $\mathrm{L}(B) = \mathrm{L}(A)$. Therefore, limiting the usage of any instruction other than L does not reduce the descriptive power of the automaton model.

Theorem 2 *Let $A = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a visit$_{\{\mathtt{L}\}}$-bounded stack automaton. Then there exists a visit-bounded stack automaton B such that $\mathrm{L}(B) = \mathrm{L}(A)$. Hence, $\mathcal{L}(\mathsf{VISIT}) = \mathcal{L}(\mathsf{VISIT}_{\{\mathtt{L}\}})$.*

Proof Let A be visit$_{\{\mathtt{L}\}}$-bounded, i.e., A visits every tape cell using the L instruction at most k times. We prove the theorem by describing a construction of the automaton B. The basic idea of the construction is that B emulates a computation of A, but every symbol on the work tape of A shall be represented by multiple copies of the same symbol on the work tape of B. Instructions of A operating on a specific tape cell will be distributed among the copies of this cell by B in such a way that every copy is only visited a fixed number of times. By careful counting we show that any computation of A can be emulated by B in such a way that the number of visits to every cell of B on any instruction can be bounded as a function of k. This means that there is a constant ℓ which depends on k such that B is ℓ-visit-bounded, i.e., B is visit-bounded. The detailed construction of B follows.

To represent a configuration of A, the machine B will use several copies of every symbol on A's work tape. The last (right-most) copy will be denoted by a special symbol $\bar{x}$, to allow B to distinguish between several consecutive instances of the same symbol on the stack. This includes the left end marker $\triangleright$, whose copies will be denoted by a new symbol $\blacktriangleright$. In a majority of cases, to represent a configuration of A where the head is scanning a work tape symbol x, the head of B will be scanning the last (barred) copy of this symbol. Thus a stack string $\triangleright xyy \dashv z$ of A can be represented in B by the string $\triangleright \blacktriangleright\blacktriangleright \bar{\blacktriangleright} x\bar{x}yyy\bar{y}y\bar{y} \dashv zz\bar{z}$. The number of copies of each symbol will be nondeterministically chosen such that there are enough copies for all the operations of B to be performed correctly, see details below.

Next, for every instruction of A we describe a sequence of operations by which B will emulate this instruction. For every sequence we carefully list all work tape cells that B visits using any instruction. The goal is to show that the number of visits to every cell of B can be bounded by a fixed function of k, the visit bound for A.

To initialize the computation, B pushes a nondeterministically chosen number of symbols $\blacktriangleright$ on the stack, followed by a single symbol $\bar{\blacktriangleright}$. This adds one visit to each of the newly created cells at the beginning of the computation.

To simulate a push(x) **instruction of** A, the machine B pushes a nondeterministically chosen number of symbols x on the stack, followed by a single symbol $\bar{x}$. This adds one visit to each of the newly created cells.

To simulate an L **instruction of** A, the machine B moves its head left until it reaches the next barred symbol. This adds a visit to cells of the tape of B corresponding to two different cells of A. Call the cell of A the head was scanning before the transition the *top* cell, and the cell the head is scanning after the transition the *bottom* cell. The sequence of transitions of B adds one visit to each cell with a non-barred symbol corresponding to the top cell, and one visit to the cell with the barred symbol corresponding to the bottom cell.

How many times can such a sequence of transitions visit any one cell of B? Recall that A is k-visit$_{\{\mathtt{L}\}}$-bounded, therefore an `L` transition can only visit the bottom cell at most k times. And since the bottom cell can not be removed from the stack before the top cell is, this also means that an `L` transition can only originate in the top cell at most k times. Therefore, over the entire computation, the number of visits of any single cell of B can only increase by at most k during all of these sequences combined.

To simulate an `R` **instruction of** A, the machine B moves its head right until it reaches the next symbol marked with a bar. Call the cell that the head of A was scanning before this instruction was executed the *bottom* cell, and the cell that it is scanning after performing this instruction the *top cell*. The sequence of transitions of B adds one visit to every cell that corresponds to the top cell in A.

How many of these sequences can be performed targeting the same top cell? Since the head of a stack automaton can only move one cell at a time, this `R` instruction had to be preceded by an `L` instruction moving the head from the top cell to the bottom cell earlier in the computation. Moreover, every `R` instruction from the bottom cell to the top cell can be uniquely paired with a preceding `L` instruction from the top cell to the bottom cell. Since the bottom cell can not be removed from the stack before the top cell is, the number of `R` instructions ending in the top cell is bounded by the number of `L` instructions ending in the bottom cell, which is in turn bounded by k as A is k-visit$_{\{\mathtt{L}\}}$-bounded. Therefore any cell of A can only be the destination of an `R` instruction at most k times, and the resulting sequences of instructions in B can only add at most k visits to the corresponding cells of B over the entire computation.

To simulate a `pop` **instruction of** A, denote the cell being popped as the *top cell*, and the cell the head of A scans after the transition (the new top of the stack) as the *bottom cell*. The machine B first removes all symbols corresponding to the top cell from its stack, then removes the two right-most symbols corresponding to the bottom cell (one barred and one not), and finally adds a new symbol with a bar corresponding to the bottom cell to the stack. For example, if B starts with a stack $\triangleright \blacktriangleright \bar{\blacktriangleright} xxx\bar{x}yy\bar{y} \dashv$, after this sequence the stack will be $\triangleright \blacktriangleright \bar{\blacktriangleright} xx\bar{x} \dashv$. If there are not enough symbols corresponding to the bottom cell on the stack of B, due to the computation not adding enough of them during the sequence that simulated pushing the bottom cell on the stack, then B immediately halts and rejects.

Why do we need to modify the content of the stack of B corresponding to the bottom cell? Observe that the sequence of `pop` instructions of B visits each symbol corresponding to the top cell once, but it also visits the cell containing the barred symbol corresponding to the bottom cell. If B only removed the symbols corresponding to the top cell, the cell with the barred symbol would get visited during every `pop` instruction of A ending in the same bottom cell. A sequence of repeated `push` and `pop` instructions could therefore add an unbounded number of visits to this cell, since neither of these instructions counts as a visit in A. Adding the extra instructions to remove two symbols and add a new one causes any symbol corresponding to the bottom cell to be visited at most two times by this sequence of instructions before it is removed from the stack.

Simulating `S` instructions of A will be done in two different ways, depending on whether the cell on which this instruction is performed is currently on top of the stack or not. Additionally, B will always simulate an entire contiguous sequence of `S` instructions of A at the same time. Therefore, assume that the instructions immediately preceding and following this sequence are one of `L`, `R`, `push`, or `pop`. (Except for the cases when this sequence is at the very beginning or end of the computation.)

To simulate a sequence of `S` instructions which operate on the cell on top of the stack of A, the machine B shall remove one symbol corresponding to this cell with each `S` instruction executed; i.e., `S` instructions of A shall be replaced by `pop` instructions in B. At the end of this sequence B shall push a new barred symbol to preserve the proper form of the stack word. As during the simulation of the `pop` instruction, if there are not enough symbols corresponding to the cell of A on the stack of B, the machine halts and rejects. Since cells visited by this sequence are removed from the stack, any cell of B is visited at most once by this sequence of operations.

To simulate a sequence of `S` instructions which operate on a cell below the top of the stack of A, the machine B replaces each `S` instruction with an `L` instruction. This means that to simulate a sequence of n `S` instructions of A, B visits the top n copies of the affected cell on its stack. At the end of simulating this sequence, B moves its head right back to the barred symbol representing the current cell to continue its computation. Once again, if there are not enough copies available and B hits the barred symbol representing the next cell, B halts and rejects.

During each sequence of consecutive `S` instructions of A, the machine B visits every cell corresponding to the affected cell of A at most two times. We ask how many times does A access the same cell of its stack in this way. Recall that we simulate the full contiguous sequence of `S` instructions of A in one run. Therefore, before this sequence starts, A has to enter the affected cell, and after the sequence ends, it leaves this cell. To execute another such sequence targeting the same cell, A has to return to the cell using another instruction.

We can classify the sequences of `S` instructions of A according to which instruction immediately precedes this sequence. Sequences which follow a `push` or `pop` instruction operate on the top of the stack, and thus are dealt with in the section above instead. Since A is k-visit-bounded, there are at most k `L` instructions ending in this cell, and therefore at most k distinct segments of `S` instructions following an `L` instruction. Similarly to the argument in the section simulating an `R` instruction above, the number of `R` instructions that visit a specific cell of A is bounded by the number of `L` instructions visiting the cell immediately below it, which in turn is bounded by k. Thus there can be at most k sequences of `S` instructions following an `R` instruction that operate on this cell.

The number of visits of every cell of B visited by the simulations of every instruction of A is summarized in Table 1. From this summary we can see that every cell of B is visited at most $7k + 6$ times by any instruction of B. Therefore B is visit-bounded as required. □

As a consequence of Theorem 2, the classes of languages accepted by visit$_{\{\mathtt{L}\}}$-bounded and visit-bounded automata are identical. We can also observe the following result for context-free languages:

Corollary 1 $\mathcal{L}(\mathsf{NPDA}) \subsetneq \mathcal{L}(\mathsf{VISIT})$.

Proof A pushdown automaton can be seen as a stack automaton which never uses the `L` and `R` instructions. This automaton is trivially visit$_{\{\mathtt{L}\}}$-bounded, and by Theorem 2 its language can be accepted by some visit-bounded stack automaton. Strictness can be seen using the language $\{a^n b^n c^n \mid n > 0\}$, which is not context-free, but it can be accepted by a 3-visit-bounded stack automaton. □

We conclude this section with a comparison to Turing machines. Consider nondeterministic Turing machines with a one-way read-only input and a single work tape. If there is a bound on the number of changes of direction on the work tape (finite-reversal), we denote these machines by TMFR; if there is a bound on the number of

Table 1 Number of visits of each cell of B during the simulation of A

Operation simulated	Cells affected	Number of visits per operation	Maximum number of operations affecting one cell
Initialization	Copies of the ▶ symbol	1	1
`push`	All copies of the pushed cell	1	1
`pop`	All copies of the popped cell	1	1
	Three topmost copies of the cell below the popped cell	1	2 before removing the affected cell of B
`L`	All copies of the top cell	1	k
	The topmost copy of the bottom cell	1	k
`R`	All copies of the top cell	1	k
n consecutive `S` instructions on top of the stack	Top n symbols of the stack of B	1	1
n consecutive `S` instructions following an `L` instruction	Top n symbols corresponding to the affected cell	2	k
n consecutive `S` instructions following an `R` instruction	Top n symbols corresponding to the affected cell	2	k

times the boundary of each pair of adjacent cells is crossed (finite-crossing), we denote these machines by TMFC; and if there is a bound on the number of visits to each cell (finite-visit), we denoted these by TMFV. Greibach studies these machines in [11], where the work tape is preloaded with regular languages (or other families which we do not consider here), and the work tape is confined to the preloaded space. This preloading however does not impact the languages accepted, as shown in the proof of the following.

Proposition 3 $\mathcal{L}(\mathsf{TMFR}) \subsetneq \mathcal{L}(\mathsf{TMFC}) = \mathcal{L}(\mathsf{TMFV}) \subsetneq \mathcal{L}(\mathsf{VISIT})$.

Proof First we will argue that preloading these Turing machines with regular languages does not affect the languages accepted. Indeed, preloading can be simulated by nondeterministically guessing and writing the preloaded string before the start of the computation. In the other direction, a new dummy symbol B can be introduced, and the machine can be preloaded with B^*. The machine then guesses some starting position and simulates using B as the blank symbol. It only accepts if it is preloaded with a string that is longer than the number of cells visited and it guesses a valid starting position where the head never runs out of available tape cells in either direction.

Greibach shows that $\mathcal{L}(\mathsf{TMFR}) \subsetneq \mathcal{L}(\mathsf{TMFC}) = \mathcal{L}(\mathsf{TMFV})$ in Theorems 2.15 and 3.12 of [11]. To show that $\mathcal{L}(\mathsf{TMFV}) \subseteq \mathcal{L}(\mathsf{VISIT})$, we use Lemma 4.21, where Greibach shows that every language in $\mathcal{L}(\mathsf{TMFV})$ can be accepted by a Turing machine preset with a regular language where the machine does not ever change the work tape contents, and every accepting computation is k-visit-bounded. Such a machine can be simulated by a visit-bounded stack automaton by first guessing the work tape contents, and then replicating the computation. The inclusion is strict following Greibach's proof of Theorem 4.26, as the context-free Dyck language cannot be accepted by a TMFV. □

4 Semilinearity

The main result of this section is to prove that the language accepted by any visit-bounded stack automaton is semilinear. To prove this, we give a procedure that, given a visit-bounded stack automaton M, constructs a pushdown automaton P, such that $\mathrm{L}(P)$ and $\mathrm{L}(M)$ are letter-equivalent. Specifically, we show that the automaton P can accept some permutation of any string in $\mathrm{L}(M)$, and vice versa. It is known that languages of pushdown automata are semilinear, and that semilinearity is preserved under letter-equivalence, hence this proves the main result.

Theorem 4 *Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a visit-bounded stack automaton. Then the language accepted by M is effectively semilinear.*

Proof Let M be k-visit-bounded for an integer k. Further, assume that the automaton ends its computation with an empty stack. If it does not, this can be achieved by deleting the entire content of the stack before accepting, which adds at most one visit to every tape cell.

The central concept used for the proof is the *visit history* of a tape cell. Using the analogy of a physical work tape with paper cells, every time the automaton makes a move, it records the transition it has just used (a 5-tuple (p, a, x, q, ι)) on both the cell it left and the cell it entered. If the transition used an S instruction, those two refer to the same cell. In this way, since every cell is visited at most k times before it is destroyed, the visit history of every cell contains at most $2k$ entries: k for transitions which were used to enter the cell, and another k for transitions which were used to leave. We shall refer to the i-th entering transition as $t_{\text{in}}[i]$ and the i-th leaving transition as $t_{\text{out}}[i]$. Also note that throughout the computation of the machine every transition used is recorded exactly twice: once in the cell it begins in, and once in the cell it ends in. This connection links the visit histories of all cells into a linked list-like structure which records the entire computation of M. Since every transition record contains the input symbol being read (if any), following these links allows us to reconstruct the string that is being read in the computation.

Our goal is to construct a pushdown automaton P, which emulates the push and pop instructions in some computation of M, while nondeterministically guessing the entire history of every cell pushed on the stack. As long as P can ensure the integrity of links between every pair of adjacent cells, the entire linked list can be followed to reconstruct a computation of M, including the L, R, and S instructions. Then if P also reads all input symbols corresponding to every t_{in} transition in all histories, it accepts a permutation of the string accepted by M in this computation.

An important fact affecting the construction of P is that cells on the work tape of M can be erased and replaced by another cell. Therefore, not all of the transitions in the history of one cell always correspond to transitions in the history of one adjacent cell. Some transitions could connect to a cell that had been in that place but was previously erased, and some transitions might connect to a cell that will be in that place in the future, after the currently adjacent cell is erased. Therefore, the representation of every cell in P will additionally carry a *completed transition counter*, an index ctc in the range $1 \leq ctc \leq k$, which indicates how many transitions in the history of the current cell have already been matched with corresponding transitions in the histories of adjacent cells. Outgoing transitions from one cell are matched with incoming transitions in adjacent cells in the same order as the machine M executes these transitions, with the ctc acting as an index to the transition being currently processed. This ensures that every outgoing transition in the history of the the current cell has been matched to one and exactly one incoming transition in the history of some adjacent cell.

When the pop transition which removes the current cell from the stack is encountered, and this transition can be linked to an incoming pop transition in the cell directly below the current cell, processing of the current cell is finished. The machine P pops the record representing the current cell from its stack, and the algorithm can continue working on the cell below. There, the ctc of the bottom cell indicates where the algorithm should resume matching further transitions from that cell.

We can now describe the construction of the pushdown automaton P in detail. □

Definition 1 *A* $(2k + 2)$-tuple $(x, ctc, t_{\text{in}}[1], \ldots, t_{\text{in}}[k], t_{\text{out}}[1], \ldots, t_{\text{out}}[k])$ *is called a history card, where:*

- $x \in (\Gamma \cup \{\triangleright\})$ is the stack symbol written on the tape cell,

- ctc, with $1 \leq ctc \leq k$, is the completed transition counter,
- $t_{\text{in}}[i] \in (\delta \cup \{\emptyset\})$, for $1 \leq i \leq k$, are the transitions ending in this cell, and
- $t_{\text{out}}[j] \in (\delta \cup \{\emptyset\})$, for $1 \leq j \leq k$, are the transitions originating in this cell.

Not all possible history cards can appear in some computation of M. We impose several consistency constraints on the history cards that P can use, to ensure that the information on each card is filled in properly and does not contradict itself.

Definition 2 *A history card is internally consistent, if all the following hold:*

- $t_{\text{in}}[i] \neq \emptyset \iff t_{\text{out}}[i] \neq \emptyset$ for all $1 \leq i \leq k$. If there is an incoming transition, there has to be a corresponding outgoing transition.
- If $t_{\text{in}}[i] = \emptyset$, then also $t_{\text{in}}[i+1] = \emptyset$. Similarly if $t_{\text{out}}[i] = \emptyset$, then also $t_{\text{out}}[i+1] = \emptyset$. This holds for all $1 \leq i < k$. Transitions are always stored in a contiguous block of indices starting from the beginning of the card.
- The transition $t_{\text{in}}[1]$ performs the $\texttt{push}(x)$ instruction, where x is the symbol stored on this card. The last non-empty $t_{\text{out}}[i]$ performs the $\texttt{pop}$ instruction. No other t_{in} transitions are $\texttt{push}$ and no other t_{out} transitions are $\texttt{pop}$ instructions. The history of a cell begins when it is pushed and ends when it is popped from the stack. Each of these events can only happen once in the lifetime of the cell.
- The exception to the three above rules is a card with $x = \triangleright$. This card represents the bottom of the stack of M, and here the computation of M begins and ends. Therefore $t_{\text{in}}[1] = \emptyset$, there is exactly one i such that $t_{\text{in}}[i] \neq \emptyset$ and $t_{\text{out}}[i] = \emptyset$, no t_{in} is a $\texttt{push}$ or $\texttt{R}$ instruction, and no t_{out} is a $\texttt{pop}$ or $\texttt{L}$ instruction.
- The work tape symbol read in every t_{out} transition is the symbol x on this card.

Denote by H the set of all internally consistent history cards. Note that $|H| \leq (|\Gamma| + 1)k(|\delta| + 1)^{2k}$. The set H shall be the working alphabet of the pushdown automaton P. An example of history cards and links between them corresponding to a computation of M is shown in Fig. 1. The links are not explicitly stored but will be implied.

Now we describe an algorithm used by P to simulate a computation of M. This algorithm employs two subprocedures. The first one advances the completed transition counter on a card step by step, verifying that the transitions on the card can link together to form a continuous computation, until either a $\texttt{push}$ or a $\texttt{pop}$ transition, or the end of the computation is reached. The facts that need to be verified are that $\texttt{S}$ instructions on this card link to each other, and that every outgoing $\texttt{L}$ instruction is followed by an incoming $\texttt{R}$ instruction.

The second procedure takes two history cards as input and attempts to link together transitions between them. An outgoing $\texttt{push}$ instruction on the bottom card has to link to the first incoming instruction on the top card. Every outgoing $\texttt{R}$ instruction on the bottom card has to link to an incoming $\texttt{R}$ instruction on the top card, and every outgoing $\texttt{L}$ instruction on the top card has to link to an incoming $\texttt{L}$ instruction on the bottom card. Finally, the last transition of the top card, performing a $\texttt{pop}$ instruction, has to link to an incoming $\texttt{pop}$ transition on the bottom card.

The complete description of both procedures can be found in the Appendix. The important fact is that since there are only finitely many different history cards, the

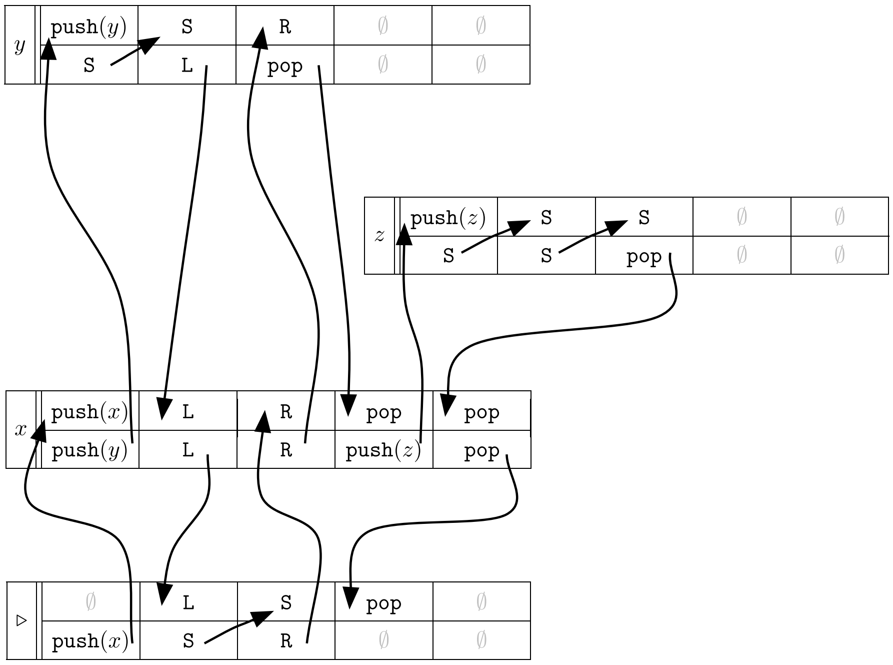

Fig. 1 Histories of tape cells after executing the following sequence of instructions: $\mathtt{push}(x), \mathtt{push}(y), \mathtt{S}, \mathtt{L}, \mathtt{L}, \mathtt{S}, \mathtt{R}, \mathtt{R}, \mathtt{pop}, \mathtt{push}(z), \mathtt{S}, \mathtt{S}, \mathtt{pop}, \mathtt{pop}$. Only the instructions used in the transitions are shown, states and symbols read are omitted. Transitions t_{in} are written in the top row of a card, and t_{out} in the bottom row. Arrows show links between history cards formed by pairs of identical transitions

pushdown automaton itself does not have to perform either of these procedures. The results for all possible inputs can be encoded into its transition function.

A detailed description of the algorithm performed by P is in Algorithm 1.

If the computation of P succeeds, all transitions in all the history cards used can be linked together to form one possible contiguous computation of M. Further, P reads every symbol that is read by every instruction in this computation, just not necessarily in the same order as M. This means that the string read by P is a permutation of the string that is read by the corresponding computation of M. Therefore, the language of P is letter-equivalent to the language of M. Finally, since the languages of pushdown automata are semilinear, and semilinearity is preserved under letter-equivalence, this means that the language of M is semilinear as well.

5 Other Expensive Instruction Sets

So far we have considered automata models with expensive instruction sets $E = \{\mathtt{L}\}$ and $E = \{\mathtt{S}, \mathtt{L}, \mathtt{R}, \mathtt{push}, \mathtt{pop}\}$. We can ask whether models with other expensive instruction sets also describe the same class of languages.

Algorithm 1 The algorithm performed by the pushdown automaton P emulating a computation of a visit-bounded stack automaton.

```
1: Nondeterministically choose a history card containing the symbol ▷.
2: Push this card on the stack.
3: Read all input symbols that are read in any incoming transition on this card.
4: while There is a history card on the stack do
5:     Advance the $ctc$ of the card on top of the stack,
        verifying the consistency of instruction links,
        until either a push or a pop instruction,
        or the end of the computation is encountered.[1]
6:     if The transition encountered performs a push instruction then
7:         Nondeterministically choose a new history card
            containing the symbol being pushed.
8:         Verify that the chosen card can be matched
            to the card currently on top of the stack.[2]
9:         if The cards can be matched together then
10:            Move the $ctc$ of the card on top of the stack
               to the incoming pop instruction corresponding
               to the removal of the cell represented by the new card.
11:            Push the newly chosen card on top of the stack,
               initializing its $ctc$ to 1.
12:            Read all input symbols that are read
               in any incoming transition on the new card.
13:        else
14:            Halt the computation and reject.
15:        end if
16:    else if The transition encountered performs the pop instruction then
17:        Erase the top card from the stack.
18:    else if The end of the computation of $M$ is encountered then
19:        Halt the computation and accept.
20:    else if A transition on the card can not be linked properly then
21:        Halt the computation and reject.
22:    end if
23: end while
```

[1] See Algorithm 2.

[2] See Algorithm 3.

It is possible to show that visit$_{\{\mathtt{R}\}}$-bounded automata accept the same class of languages as visit$_{\{\mathtt{L}\}}$-bounded automata. The proof uses similar ideas as in the construction in the proof of Theorem 2, though we do not include it here. Adding the S instruction to a set of expensive instructions does not change the class of languages accepted, as every S instruction can be replaced by a pair of R and L instructions, or push and pop instructions when operating on top of the stack. Therefore it is always possible to construct an equivalent automaton which never uses the S instruction. Hence, if we consider any expensive instruction set E containing either L or R, any visit-bounded automaton is also visit$_E$ bounded for such E. Therefore all models with such an expensive instruction set accept the same class of languages.

Making expensive instructions exactly the push instructions has no effect on the languages accepted (i.e. this model accepts all stack languages), as any cell can only be pushed on the stack once. We only include the push instruction as a possible expensive instruction for completeness.

Finally, we shall see that a model with $E = \{\texttt{pop}\}$ also accepts all stack automaton languages. Using a procedure similar to the one in the construction of automaton B in Theorem 2 we can clone symbols on the stack and replace every pop transition by a sequence pop − pop − push, such that every cell is visited at most twice by pop instructions.

These results can be summarized in a hierarchy depicted in Fig. 2. Recall that NESA denotes the class of non-erasing stack automata, and CSA the class of checking stack automata.

Theorem 5 *The hierarchy shown in Fig. 2 is correct.*

Proof The fact that $\mathcal{L}(\mathsf{TMFR}) \subsetneq \mathcal{L}(\mathsf{TMFC}) = \mathcal{L}(\mathsf{TMFV}) \subsetneq \mathcal{L}(\mathsf{VISIT})$ is shown in Proposition 3. That $\mathcal{L}(\mathsf{TMFV}) \subsetneq \mathcal{L}(\mathsf{CSA})$ is shown in [11]. That $\mathcal{L}(\mathsf{CSA}) \subsetneq \mathcal{L}(\mathsf{NESA}) \subsetneq \mathcal{L}(\mathsf{SA})$ is well known [10]. That $\mathcal{L}(\mathsf{NPDA}) \subsetneq \mathcal{L}(\mathsf{VISIT})$ is shown in Corollary 1. That $\mathcal{L}(\mathsf{VISIT}) = \mathcal{L}(\mathsf{VISIT}_{\{\mathtt{L}\}})$ is from Theorem 2, and the equality with $\mathcal{L}(\mathsf{VISIT}_{\{\mathtt{R}\}})$ is discussed above. The equality of $\mathcal{L}(\mathsf{SA})$ with $\mathcal{L}(\mathsf{VISIT}_{\{\mathtt{pop}\}})$ and $\mathcal{L}(\mathsf{VISIT}_{\{\mathtt{push}\}})$ is also mentioned above. Further, $\mathcal{L}(\mathsf{CSA})$ contains languages not accepted by $\mathcal{L}(\mathsf{VISIT})$ by Theorem 4, since $\mathcal{L}(\mathsf{CSA})$ contains non-semilinear languages [10]. By transitivity this proves that the inclusion of $\mathcal{L}(\mathsf{VISIT})$ in $\mathcal{L}(\mathsf{SA})$ is proper. Finally, it is known that $\mathcal{L}(\mathsf{NESA})$ does not contain all context-free languages [11]. □

Hence, all the families above that are semilinear are contained in $\mathcal{L}(\mathsf{VISIT})$, making it the most powerful such family.

6 Conclusions and Future Directions

We have introduced a new restriction of stack automata — k-visit-bounded automata — which are only allowed to visit each cell of the stack tape at most k times. In contrast to similar restrictions previously considered, e.g. [11], popping a cell from the stack resets its visit count to zero. More generally, we also study k-visit$_E$-bounded automata, where the limit on number of visits only applies to certain instructions performed by the automaton. Depending on the set E, we show a hierarchy of language clases

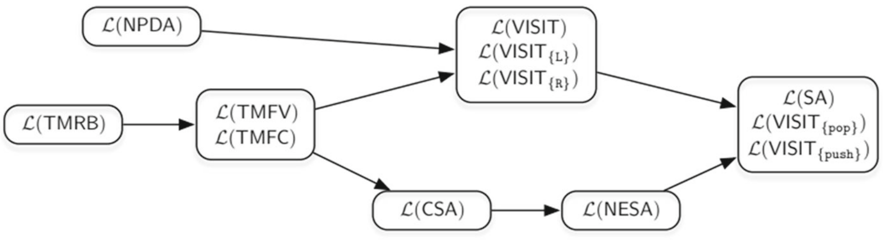

Fig. 2 Inclusion relationships between various language families described in this paper. Families in each box are equal. Proper inclusions are shown with an arrow. Families with no lines connecting them are incomparable

depicted in Fig. 2. In particular, restricting only the use of the “move left” instruction L allows the machines to recognize the same class of languages as restricting every possible instruction. We call the resulting family of languages $\mathcal{L}(\mathsf{VISIT})$. The class $\mathcal{L}(\mathsf{VISIT})$ properly contains all context-free languages, but also properly contains languages accepted by finite-visit Turing machines. Moreover, the family only contains effectively semilinear languages. An algorithm is provided that takes a visit-bounded stack automaton as input, and constructs a letter-equivalent pushdown automaton.

Several interesting questions remain unanswered. What classes of languages result from applying the visit-bounded restriction to nonerasing, or checking stack automata (see for example [10, 13])? These families will be a subset of languages of visit-bounded stack automata, since the models are more restrictive; and a proper subset of $\mathcal{L}(\mathsf{NESA})$, resp. $\mathcal{L}(\mathsf{CSA})$, since these models without restrictions can accept non-semilinear languages. Can these classes be compared to any other well-known language classes? Does an analogy of Theorem 2, that the class of visit$_{\{\mathtt{L}\}}$-bounded languages coincides with visit-bounded languages, also hold for NESA, resp. CSA models?

As we have proven that visit-bounded stack automata languages are effectively semilinear, for every such automaton we can construct a regular language that is letter-equivalent. Directly following the steps of the construction in the proof of Theorem 4 to build a pushdown automaton, and then a construction from [2] to obtain the necessary constant and periods, yields a fairly trivial upper bound on the state complexity of this regular language. Can a tighter upper bound, or some nontrivial lower bound, be obtained? Semilinearity also implies that various decision problems for visit-bounded stack automata are decidable, including emptiness, finiteness, or membership, see [3]. Can anything be said about the computational complexity of these problems?

The construction in the proof of Theorem 2, which builds a visit-bounded stack automaton accepting the same language as a given visit$_{\{\mathtt{L}\}}$-bounded stack automaton, results in a machine which potentially uses significantly more stack space than the original machine. In particular, every S and pop instruction of the original machine “consumes” one extra cell from the tape of the output machine. Is it possible to perform this construction with only some fixed blowup in the amount of stack space required? Space complexity of stack automata has been recently studied by a team including the authors of this paper in [14]. Additionally, the k-visit-bounded limitation implies that the time complexity (number of instructions needed for a computation) is at most k times the space complexity (maximum size of the stack during the computation). This fact could be used to obtain new results about complexity aspects of visit-bounded modifications of various models, including Turing machines.

Acknowledgements Supported, in part, by a grant from Natural Sciences and Engineering Research Council of Canada 2022-05092 (Ian McQuillan).

Declarations

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A: Detailed Algorithm Listings

This section contains the description of two algorithms: one to advance the current transition counter of a history card and verify that all encountered transitions can be properly linked, and the other to match two history cards together. Since there are only finitely many possible history cards, the pushdown automaton P from the proof of Theorem 4 does not need to actually perform these procedures, but their result for every possible input card can be encoded in its transition function.

Algorithm 2 Advancing the completed transition counter of a history card to the next `push` or `pop` transition, or to the end of the computation. At the same time, it is verified that every transition in this card can be properly linked to some other transition.

```
Require: An internally consistent history card.
Ensure: The $ctc$ of the card is advanced to the next push or pop transition.
Ensure: If the transitions on this card can not be properly linked to some other card, the input is rejected.
repeat
    if $t_{\text{out}}[ctc]$ performs the S instruction then
        if $t_{\text{in}}[ctc + 1] = t_{\text{out}}[ctc]$ then
            $ctc := ctc + 1$
        else Halt computation and reject input.
        end if
    else if $t_{\text{out}}[ctc]$ performs the L instruction then
        if $t_{\text{in}}[ctc + 1]$ performs the R instruction then
            $ctc := ctc + 1$
        else Halt computation and reject input.
        end if
    else if $t_{\text{out}}[ctc]$ performs the R instruction then
        Moving right before a new cell has been pushed on top.
        Halt computation and reject input.
    else if $t_{\text{out}}[ctc]$ performs a push or pop instruction then
        A new cell has been pushed, or the current cell popped.
        break main loop
    else if $t_{\text{out}}[ctc] = \emptyset$ and $x = \triangleright$ then
        End of computation.
        break main loop
    end if
until finished
```

Algorithm 3 Matching two history cards. Returns **true** if the transitions on the cards can be matched together, **false** if not.

```
Require: Two internally consistent history cards, labeled $B$ (bottom) and $T$ (top).
Require: The $ctc$ of the bottom card points to an outgoing push instruction which pushes the cell represented by the top card.
Ensure: The $ctc$ of the bottom card is advanced to the incoming pop instruction corresponding to the popping of the top card.
1: $i := 1$
2: $j := B.ctc$
   The variables i and j track the transition in T and B respectively that is currently being matched. The first instruction to match must be a push instruction pushing the top card on the stack:
3: if $B.t_{\mathrm{out}}[j]$ performs the push($T.x$) instruction,
       and $T.t_{\mathrm{in}}[1] = B.t_{\mathrm{out}}[j]$ then
4:     goto match_top
5: else return false
6: end if

7: label match_top:
   Match the next transition from the top card.
8:     if $T.t_{\mathrm{out}}[i]$ performs the S instruction then
9:         if $T.t_{\mathrm{in}}[i+1] = T.t_{\mathrm{out}}[i]$ then
10:            $i := i + 1$
11:            goto match_top
12:        else return false
13:        end if
14:    else if $T.t_{\mathrm{out}}[i]$ performs the L instruction then
15:        if $B.t_{\mathrm{in}}[j+1] = T.t_{\mathrm{out}}[i]$ then
16:            $j := j + 1$
17:            goto match_bottom
18:        else return false
19:        end if
20:    else if $T.t_{\mathrm{out}}[i]$ performs the R or push instruction then
           Whether these transitions can be linked will be checked once the next card is pushed on the stack. For now we only need to know if the computation can return to the top card.
21:        if $T.t_{\mathrm{in}}[i+1]$ performs the L or pop instruction then
22:            $i := i + 1$
23:            goto match_top
24:        else return false
25:        end if
26:    else if $T.t_{\mathrm{out}}[i]$ performs the pop instruction then
27:        if $B.t_{\mathrm{in}}[j+1] = T.t_{\mathrm{out}}[i]$ then
28:            $B.ctc := j + 1$
29:            return true
30:        else return false
31:        end if
32:    end if
```

```
33: label match_bottom:
    Match the next transition from the bottom card.
34:   if $B.t_{\text{out}}[j]$ performs the S instruction then
35:     if $B.t_{\text{in}}[j+1] = B.t_{\text{out}}[j]$ then
36:       $j := j + 1$
37:       goto match_bottom
38:     else return false
39:     end if
40:   else if $B.t_{\text{out}}[j]$ performs the L instruction then
41:     if $B.t_{\text{in}}[j+1]$ performs the R instruction then
          The fact that the computation can continue from the outgoing transition to the incoming transition has already been verified when the bottom card was pushed on the stack.
42:       $j := j + 1$
43:       goto match_bottom
44:     else return false
45:     end if
46:   else if $B.t_{\text{out}}[j]$ performs the R instruction then
47:     if $T.t_{\text{in}}[i+1] = B.t_{\text{out}}[j]$ then
48:       $i := i + 1$
49:       goto match_top
50:     else return false
51:     end if
52:   else if $B.t_{\text{out}}[j]$ performs the push or pop instruction,
      or $B.t_{\text{out}}[j] = \emptyset$ then
        Trying to push, pop, or end computation from the bottom card while the top card is still on the stack.
53:     return false
54:   end if
```

References

1. Harrison, M.: Introduction to Formal Language Theory. Addison-Wesley, Reading, Ma (1978)
2. Parikh, R.: On context-free languages. J. ACM **13**(4), 570–581 (1966)
3. Ibarra, O., McQuillan, I.: Semilinearity of families of languages. International. J. Found. Comput. Sci. **31**(8), 1179–1198 (2020)
4. Harju, T., Ibarra, O., Karhumäki, J., Salomaa, A.: Some decision problems concerning semilinearity and commutation. J. Comput. Syst. Sci. **65**(2), 278–294 (2002)
5. Ibarra, O.H., McQuillan, I.: On families of full trios containing counter machine languages. Theor. Comput. Sci. **799**, 71–93 (2019)
6. Ibarra, O.H., McQuillan, I., Ravikumar, B.: On counting functions and slenderness of languages. Theor. Comput. Sci. **777**, 356–378 (2019)
7. Joshi, A.K.: Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions?, pp. 206–250. Cambridge University Press (1985)
8. Ginsburg, S., Greibach, S., Harrison, M.: Stack automata and compiling. J. ACM **14**(1), 172–201 (1967)
9. Ginsburg, S., Greibach, S., Harrison, M.: One-way stack automata. J. ACM **14**(2), 389–418 (1967)
10. Greibach, S.: Checking automata and one-way stack languages. J. Comput. Syst. Sci. **3**(2), 196–217 (1969)
11. Greibach, S.A.: One way finite visit automata. Theor. Comput. Sci. **6**, 175–221 (1978)
12. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading, MA (1979)
13. Harrison, M.A., Schkolnick, M.: A grammatical characterization of one-way nondeterministic stack languages. J. ACM **18**(2), 148–172 (1971)

14. Ibarra, O., Jirásek, J., McQuillan, I., Prigioniero, L.: Space complexity of stack automata models. Int. J. Found. Comput. Sci. **32**(06), 801–823 (2021)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.