Sampling a Two-way Finite Automaton[1]

Zhe Dang[a,b], Oscar H. Ibarra[c], Qin Li[a]

[a]*School of Computer, Anhui University of Technology, Ma'anshan, Anhui, China*
[b]*School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164, USA*
[c]*Department of Computer Science, University of California, Santa Barbara, CA 93106, USA*

Abstract

We study position sampling in a two-way nondeterministic finite automaton (2NFA) to measure information flow between state variables, based on the information-theoretic sampling technique proposed in [20]. We prove that for a 2NFA, the language generated by position sampling is regular. We also show that for a 2NFA, we can effectively find a vector of sampling positions that maximizes information flow in a run of the 2NFA. Finally, we give some language properties of sampled runs of 2NFAs augmented with restricted unbounded storage. We conduct preliminary experiments to identify information concentration in a subset of statements in a program.

Keywords: two-way automata, information rate, sampling

1. Introduction and Motivation

One way to understand the behavior of a software system is through observation. That is, when the system runs, we record a sequence of values of all or part of its state variables like PC (program counter), variable values, pointer locations, stack frames, I/O, etc. Such a sequence, called a trace, can later be used for either off-line or online analysis. A trace contains valuable information such as information flow among the state variables, where "information flow", a jargon in software security, between state variables means

[1]A preliminary version of this paper has appeared in The Emergence, Complexity and Computation (ECC) series, Volume 12: Automata, Universality, Computation (Springer), 2014.

the information, totally [14] or partially [12, 22], flows from secret variables to public ones. When confidentiality is the concern, this kind of information flow is illegal, and must be constrained.

There have been a number of white-box techniques using static analysis, such as program slicing [32, 34], type analysis [3, 27, 30], probabilistic analysis [21, 24, 28], and information-theoretic analysis [5, 12, 23], to find information flow among variables. In our recent work [20], we introduce a new information-theoretic technique that does not depend on static analysis and can be adapted to black-boxes easily as shown in [20].

There has been a fundamental notion shown below, proposed by Shannon [26] and later by Chomsky and Miller [4], that computes the information quantity in a string (word). Let L be a set of words over a given finite and non-empty alphabet Σ, and $S_n(L)$ be the number of words of length n in L. The *information rate* λ_L of L is defined as

$$\lambda_L = \lim \frac{\log S_n(L)}{n}.$$

Where the limit does not exist, we take the upper limit, which always exists for a finite alphabet. Throughout this paper, the logarithm is in base 2. Intuitively, λ_L is the average amount of information per symbol contained in a word in L. We emphasize that the information rate does not require any probabilistic nor statistical arguments. Instead, it is a characteristic of the language L itself.

Based on the Shannon information rate, we introduced information flow in [20]. We now briefly state the definitions. Consider a device M where we observe two of its finite state variables, x_1 and x_2, and record a trace, α. Herein, the trace is a sequence of the values of the two state variables on each move when the device is running. Notice that α can be considered as a word on alphabet $S_1 \times S_2$ (i.e., a symbol in the alphabet is a pair of values of the two variables) where S_1 and S_2 are the two finite sets that x_1 and x_2 are respectively drawn from. M, in general, is nondeterministic. We use $L_{x_1,x_2} \subseteq (S_1 \times S_2)^*$ to denote the language of all such traces α that can be observed and use λ_{x_1,x_2} to denote the information rate of L_{x_1,x_2}. When we replace every value of x_2 with a special symbol "$-$" in every α in the language, we obtain a new language $L_{x_1,-}$ with its information rate denoted by $\lambda_{x_1,-}$. For instance, when L_{x_1,x_2} is the language of two words $(a,c)(b,d)$ and $(a,c)(b,c)$, each of which is clearly with length two, $L_{x_1,-}$ is the language of only one word $(a,-)(b,-)$ which is also of length two. Similarly, one can

define λ_{-,x_2}. Now, information flow from x_1 to x_2 is defined as

$$F(x_1; x_2) = \lambda_{x_1,-} + \lambda_{-,x_2} - \lambda_{x_1,x_2}.$$

Notice that, as in the classic Venn diagram of Shannon information, the quantity $\lambda_{x_1,-} + \lambda_{-,x_2} - \lambda_{x_1,x_2}$ resembles the "mutual information rate" between x_1 and x_2 which characterizes the bit rate that is shared between x_1 and x_2 in a trace.

The above definition comes from our previous work [20], where information flow is studied for various one-way automata. In this paper, our focus is on two-way nondeterministic finite automata (2NFAs). Notice that, even though 2NFAs are equivalent to their deterministic version in terms of language acceptance, the runs of 2NFAs (which are not necessarily regular) are still difficult to handle when computing their information rate [6]. In this paper, instead of looking at the runs, we study the sampled executions or traces, where the idea of sampling used in [20] is generalized. We need the following fundamental result.

Theorem 1.1. *The information rate of a regular language is computable [4].*

As pointed out in [4], the information rate can actually be efficiently computed using a matrix algorithm. Recently, we have implemented the algorithm [35] and confirmed the efficiency when approximately computing the information rates of some fairly large C programs [6]. It turns out that the notion itself is useful in software analysis and testing [7, 8, 31].

2. Information Flow in 2NFAs

Let M be a 2NFA with input alphabet Σ, which is equipped with a two-way input tape (with left end marker $\rhd \notin \Sigma$ and right end marker $\lhd \notin \Sigma$). Suppose that an input word, say $w \in \Sigma^*$, is given on the input tape (so the tape content is actually $\rhd w \lhd$). The read head in M reads the input while performing a state transition drawn from a finite set T, which is called the transition table of the M, or simply, the transitions of the M. More precisely, a state transition is in the form of

$$(s, a, s', d)$$

where s, s' are states (there are only finitely many distinct states), $a \in \Sigma \cup \{\rhd, \lhd\}$ is an input symbol or an end marker, and $d \in \{+1, -1, 0\}$ is a

direction (i.e., $+1$, -1, and 0 are respectively for moving to the right, moving to the left, and staying). For instance, the transition $(s, a, s', -1)$ means that, when M is at state s while symbol a is under the read head, the head moves to the right and the state is changed to s'. Sometimes, we explicitly indicate the direction for readability; e.g., (s, a, s', left). M is said deterministic (written 2DFA) if for each pair (s, a) there is at most one (s', d) such that $(s, a, s', d) \in T$. In this case, we write $(s, a) \to (s', d)$ for the transition (s, a, s', d). M starts from its initial state and the read head is under the left end marker. M then performs a sequence of state transitions, for some n,

$$(s_0, a_0, s_1, d_0)(s_1, a_1, s_2, d_1) \cdots (s_n, a_n, s_{n+1}, d_n)$$

while moving the head two-way on the w, where s_0 is the initial state, the first transition (s_0, a_0, s_1, d_0) reads the left end marker and moves to the right. One shall notice that the symbols $a_0, a_1, \cdots, a_n$ are actually the symbols under the read head when M executes the sequence of transitions on input $\triangleright w \triangleleft$. It is also noted that there is no transition in T that moves beyond the end markers.

M accepts w when M enters an accepting state (i.e., s_{n+1} is an accepting state in the above transition sequence). The state sequence $s_0 s_1 \cdots s_{n+1}$ in the state transition sequence witnessing the acceptance is called an accepting run. Since M is nondeterministic, there could be more than one accepting run for the given w. We use $L(M)$ to denote the set of all words w accepted by M and use $L_{\text{run},M} = \{\alpha : \alpha \text{ is an accepting run of } M \text{ on } w,\ w \in L(M)\}$ to denote the set of all accepting runs of M. Clearly, $L_{\text{run},M}$ is not necessarily a regular language on alphabet S (the states in M).

We show an example below of using 2NFA to specify executions of a program.

Example 2.1. *Consider a version of the Euclidean algorithm to compute the greatest common divisor (GCD) of two nonnegative numbers a and b:*

```
    int GCD(unsigned int a, unsigned int b){
1:    int x=a;
2:    int y=b;
3:    while  not (x == y){
4:        if  (x > y)
5:          x = x-y
```

```
6:        else
7:          y = y-x;
8:    }
9:    return x;}
```

When the program runs, the PC (treated as a variable on line numbers) starts with line 1 and moves to line 2. Depending on the input values of a and b, it may loop on the block from line 3 to line 8. Then, finally, it ends at line 9. For instance, when $a = 2$ and $b = 1$, the following sequence of line numbers will be exercised by the program: 1, 2, 3, 4, 5, 8, 3, 9. We now understand the PC as the position of a read head that reads the input word aaaaaaaaa (of nine symbols) and the aforementioned sequence is the result of the head that moves from the first symbol of the input all the way to the eighth symbol while the sixth and seventh are skipped and then moves back to the third symbol (while the seventh, the sixth, the fifth and the fourth are skipped) and then moves to the ninth symbol (while skipping all the symbols in between). This is a two-way read head. In fact, such a head is controlled by finite state transitions when each state (the quadruple of a value PC in the set $\{1, 2, \cdots, 9\}$, a value of z in the set $\{+, -, \texttt{eq}\}$, and values for local variables x and y) of the program is assumed to be drawn from a finite space (assuming that unsigned integers are of 32-bits). Notice that the z is to indicate, when the program enters the while-loop, whether $x > y$ (i.e., $z = +$), $x < y$ (i.e., $z = -$), or $x == y$ (i.e., $z = \texttt{eq}$). For instance, the following is a (finite) state transition:

> *when the state is $(PC = \texttt{4}, z = +, x = 2, y = 1)$ and the head reads an a (i.e., line 4 is now executed since $PC = \texttt{4}$), the head then moves to the right (and now, the PC points to line 5) and the rest of the state keeps the same (i.e., line 4 in the program does not change the values of z, x and y).*

Notice that a skip can also be carefully encoded as a state transition. For instance, the following state transition encodes the skip of line 7 while the head moves from line 5 to line 8 in the aforementioned sequence:

> *when the state is $(PC = 7, z = +, x = 1, y = 1)$ and the head reads an a, the head then moves to the right (and now, the PC points to line 8) and the rest of the state keeps the same (i.e., the content of line 7 has no effect since it is skipped).*

In this way, the program is essentially translated into a 2NFA M that reads the input word aaaaaaaaa only (i.e., it will crash without accepting on other input words). One shall notice that the M has many state transitions when the local variables are assumed to be drawn from a large finite range (e.g., using 32 bits to encode integers). The 2NFA accepts the input after it reads the last a (i.e., $PC = 9$). Observe that $L_{\text{run},M}$ contains many runs (that are state sequences). This is because each pair of unsigned integers x and y, as arguments to the program, produces a run. However, $L_{\text{run},M}$ is still a finite set under the aforementioned assumption.

Clearly, not every program can be modeled as a 2NFA. For instance, a program that inherently uses unbounded storage (e.g., a smartphone app that queries a cloud storage) would be such an example. On the other hand, there are certainly many different ways to model the same program as a 2NFA, as shown in the following example.

Example 2.2. *Consider now the following three threads T_1, T_2 and T_3 that run concurrently where x (which initially is 0) is the only (shared) variable and k is a constant:*

T_i (with $i = 1, 2$):

```
atomic block{
  line i.1: x = 0;
  ⋮
  line i.k: x = 0; }
line i.k + 1: while (true) x = i;
```

and T_3 :

line 3.1: if $(x == 2)$ halt.

Notice that both T_1 and T_2 start with an atomic block of k statements of $x = 0$. Following the encoding in the previous example, we can simulate the interleaved runs of the three threads by a 2NFA M that runs on the following input word w of length $(k + 1) + (k + 1) + 1$: $a \cdots a\hat{a}b \cdots b\hat{b}c$ where k symbol a's correspond to the k lines in the atomic block in T_1, symbol $\hat{a}$ corresponds to line $1.k + 1$ in T_1, k symbol b's correspond to the k lines in the atomic block in T_2, symbol $\hat{b}$ corresponds to line $1.k + 1$ in T_2, and finally symbol c corresponds to line 3.1 in thread T_3.

The M simulates the threads by, on input word w, first nondeterministically choosing to read through the a-block or the b-block. It then keeps doing the following: nondeterministically choosing to read $\hat{a}$ (if the a-block has been read through) or $\hat{b}$ (if the b-block has been read through). Hence the head now moves two-way while, for instance, skipping the b-block in between. Right after it reads $\hat{b}$, it may also nondeterministically choose to move to the right to read c and accepts. The states of M have many ways to encode, though they are finitely many. For instance, two Boolean variables can be used to memorize whether the two blocks have been read through, respectively. Notice that, in our modeling of M, we choose not to explicitly encode the values of the shared variable x in M. In contrast to the previous example, $L_{\text{run},M}$ now is an infinite set. This is because the while-loops may loop for unbounded number of times.

In practice, the finite state space S is often expressed as the Cartesian product of the state spaces of a number of finite state variables $x_1, \cdots, x_k$, for some k. In this case, every state s in S is a k-ary vector of values. For instance, in specifying a traffic light system with a sensor as an automaton, two variables `color` and `sensor` may be used to specify the states, where the `color` takes one of three values $red, yellow, green$ while the `sensor` takes one of two values no_car, car. Herein, the state space S is therefore the set of all pairs of a `color`-value and a `sensor`-value. The size of S is 6.

Let $A \subseteq \{x_1, \cdots, x_k\}$ be a set of variables. Let $s \downarrow_A$ be the result of dropping the components in vector s that correspond to variables not in A. Therefore, $s \downarrow_A$ is an $|A|$-ary vector. For instance, in the aforementioned example, when $s = (yellow, car)$, we have $s \downarrow_{\{\texttt{color}\}} = yellow$ and $s \downarrow_{\{\texttt{color},\texttt{sensor}\}} = (yellow, car)$.

For a sequence $\alpha = s_0 \cdots s_n$, we denote the sequence $s_0 \downarrow_A \cdots s_n \downarrow_A$ as $\alpha \downarrow_A$. When α is an accepting run, $\alpha \downarrow_A$ is a *trace* wrt A. Clearly, all such traces form a language $L^A_{\text{run},M} = \{\alpha \downarrow_A : \alpha \in L_{\text{run},M}\}$. Notice that $L^A_{\text{run},M}$ is a language on alphabet $S \downarrow_A = \{s \downarrow_A : s \in S\}$.

We now give a more general definition of information flow [20]. Let L be a set of state sequences and $A, B \subseteq \{x_1, \cdots, x_k\}$ be two sets of variables. We use λ_{L^A}, λ_{L^B} and $\lambda_{L^{A\cup B}}$ to denote the information rates of languages L^A $(= \{\alpha \downarrow_A : \alpha \in L\})$, L^B, and $L^{A\cup B}$, respectively. The *information flow* from A to B is defined as

$$F(A; B|L) = \lambda_{L^A} + \lambda_{L^B} - \lambda_{L^{A\cup B}}. \tag{1}$$

In the definition of a trace wrt A, we require that a trace must record the values of the variables in A at *every* step of the run. This is not always possible considering the fact that the system that M specifies may run at a speed much faster than the values can be recorded. In practice, it often suffices to sample a trace (e.g., record the values once every 100 steps), and as a result, a subsequence of a trace is obtained. We call such a subsequence as a sampled trace. Depending on the scheme used in sampling, the information flow may or may not be computable. Notice that the information rate on a sampled trace may not be even a faithful approximation to the information rate on an un-sampled trace. Therefore, a decision problem arises, when given a sampling scheme, whether the scheme would roughly maintain the bit rate of the traces of the system.

Example 2.3. *We continue our discussions on the 2NFA M in Example 2.1. Clearly, the state space of M is the Cartesian product of the ranges of PC, z, x and y. If we are only interested in variables x and y (i.e., now A is the set of (the names of) these two variables), $L^A_{\text{run},M}$ will give us all the traces of the M wrt the A (i.e., when the M runs, we record only the values of x and y in each state in a run while discarding the values for PC and z). The information flow $F(\{x\};\{y\}|L_{\text{run},M})$ intuitively characterizes the average amount of information leaked (per move of M) from variable x to variable y when M runs. Clearly, we might not be interested in all moves of M. Instead, for instance, we might only focus on line 3 of the program in Example 2.1 in the sense that an intruder can only observe executions of line 3 and in the line, can only observe the variable x's values. We would like to know how much information will be leaked about y that the intruder can not observe at line 3. That is, we shall define a way to sample a trace when M's head reads symbols on some specific positions (now the positions are only one: the third symbol on the input in Example 2.1). Then, we define a similar notion of information flow on those sampled positions, shown below.*

Recall that M works on a two-way input. As pointed out in [6], it is currently an open problem whether the information rate of the set of all accepting runs of a 2NFA is computable or not, even when the input is unary. The difficulty is that all such runs form a non-regular language whose information rate is known computable only for limited cases.

We now consider a way to sample a run of a 2NFA M. In the literature, a crossing sequence is often used for a 2NFA; i.e., to record the state of M whenever the head crosses the boundary between two input cells (i.e., two

input positions). To slightly modify the concept by recording the state of M whenever the head is under a given position of the input, we call the resulting sequence a visiting sequence. For instance, consider an input word abc and suppose that the M performs the following transition sequence on the input tape (whose content now is $\triangleright abc \triangleleft$ with the left end marker $\triangleright$ and the right end marker $\triangleleft$):

$$(q_0, \triangleright, q_1, +1)(q_1, a, q_2, 0)(q_2, a, q_3, +1)(q_3, b, q_4, -1)(q_4, a, q_5, +1)(q_5, b, q_6, +1).$$

The visiting sequence at the first position of the input word (i.e., the position of the symbol a) is $q_1 q_2 q_4$.

We now generalize the concept to multiple positions. Let $\mathbf{c} = (c_1, \cdots, c_k)$, for some k, be a monotonic sequence of rational numbers with each $0 \leq c_i \leq 1$. Let w be an input word to M. The numbers $p_i = \lfloor c_i(|w| + 1) \rfloor$ are the *sampling positions* for the w. Notice that, when $c_i = 0$ (resp. $c_i = 1$), it is the left (resp. right) end marker position of the input tape (whose content is $\triangleright w \triangleleft$). When M runs, whenever the head is at a sampling position, the state of M is recorded. Therefore, when a run is an accepting run, we may record a sequence of states, called a $\mathbf{c}$-sampling accepting run, which is a subsequence of the accepting run. For instance, suppose that the aforementioned transition sequence on input word $w = abc$ is accepting (the last state q_6 is an accepting state) and that $k = 2$, $c_1 = 0.25$ and $c_2 = 0.75$. Clearly, $p_1 = 1$ (the first position of w) and $p_2 = 3$ (the third position of w). In this case, $q_0 q_1 q_2 q_3 q_4 q_5 q_6$ is an accepting run and the subsequence $q_1 q_2 q_4 q_6$ is a $\mathbf{c}$-sampling accepting run. We use $L_{\mathrm{run},\mathbf{c}(M)}$ to denote the set of all $\mathbf{c}$-sampling accepting runs for all input words.

We first consider the case when $k = 1$ and $c_1 = 0$ (hence, only the left end (i.e., the left end marker) of an input is sampled). In this case, $L_{\mathrm{run},\mathbf{c}(M)}$ is written as $L_{\mathrm{run},\triangleright(M)}$.

Lemma 2.4. *Let M be a 2NFA. Then, $L_{\mathrm{run},\triangleright(M)}$ is a regular language.*

Proof. Without loss of generality, we assume that M enters an accepting state and returns to the left end of an input word w when it accepts w. On the input word w, we define T_w to be the set of all pairs (s, s') satisfying the following: M starts with state s at the left end of the input word w (recalling that the input tape content is actually $\triangleright w \triangleleft$), runs on the w while the head of M is not under the left end of the input, and eventually returns to the left end with state s'. Hence, during the process, the head is at the left end

only twice. Since the state set S of M is finite, there are only finitely many distinct T_w's for all $w \in L(M)$. We use $\mathcal{T}$ to denote all of them, which can be constructed effectively. This is because, as an exercise, one can show the following: For each $T \in \mathcal{T}$, the set $\{w : T_w = T\}$ is a regular language. Now, every such T defines a graph with directed edges $(s, s') \in T$. Clearly, a state sequence is in $L_{\text{run},\triangleright(M)}$ iff, for some $T \in \mathcal{T}$, the state sequence is a walk on the graph T from the initial state of M to an accepting state of M. Again, all such walks form a regular language and the result follows. $\square$

Lemma 2.4 can be generalized. We first need some definitions.

A counter is a nonnegative integer variable that can be incremented by 1, decremented by 1, or stay unchanged. In addition, a counter can be tested against 0. Let k be a nonnegative integer. A *nondeterministic k-counter machine (NCM)* is a one-way nondeterministic finite automaton, with input alphabet Σ, augmented with k counters. For a nonnegative integer r, we use NCM(k,r) to denote the class of k-counter machines where each counter is r-*reversal-bounded*; i.e., it makes at most r alternations between non-decreasing and non-increasing modes in any computation; e.g., the following counter value sequence

$$0\ 0\ 1\ 2\ 2\ 3\ \underline{3\ 2}\ 1\ 0\ \underline{0\ 1}\ 1$$

is of 2-reversal, where the reversals are underlined. For convenience, we sometimes refer to a machine M in the class as an NCM(k,r). In particular, when k and r are implicitly given, we call M as a *reversal-bounded NCM*. When M is deterministic, we use 'D' in place of 'N'; e.g., DCM. As usual, $L(M)$ denotes the language that M accepts.

Reversal-bounded NCMs have been extensively studied since their introduction in 1978 [16]; many generalizations are identified, e.g., ones equipped with multiple tapes, with two-way tapes, with a stack, etc. In particular, reversal-bounded NCMs have found applications in areas like Alur and Dill's [1] time-automata [9, 10], Paun's [29] membrane computing systems [18], and Diophantine equations [33].

Let N be the set of nonnegative integers. $\mathbf{V} \subseteq N^k$ is a *linear set* if $\mathbf{V} = \{\mathbf{v} : \mathbf{v} = \mathbf{v}_0 + t_1\mathbf{v}_1 + \cdots + t_i\mathbf{v}_m, t_1, \cdots, t_m \geq 0\}$ where $\mathbf{v}_0, \cdots, \mathbf{v}_m$ are constant k-ary vectors in N^k, for some $m \geq 0$. $\mathbf{V}$ is a *semilinear set* if it is the union of finitely many linear sets. Every finite subset of N^k is semilinear – it is a finite union of linear sets whose generators are constant vectors. Clearly, semilinear sets are closed under union and projection. It is also known that semilinear sets are closed under intersection and complementation.

Let Y be a finite set of integer variables. An atomic Presburger formula on Y is either a linear constraint $\sum_{y \in Y} a_y y < b$, or a mod constraint $x \equiv_d c$, where a_y, b, c and d are integers with $0 \leq c < d$. A Presburger formula can always be constructed from atomic Presburger formulas using $\neg$ and $\wedge$. Presburger formulas are closed under quantification. Let S be a set of k-tuples in N^k. S is Presburger definable if there is a Presburger formula $P(y_1, \cdots, y_k)$ such that the set of non-negative integer solutions is exactly S. It is well-known that S is a semilinear set iff S is Presburger definable.

Let $\Sigma = \{a_1, \cdots, a_k\}$ be an alphabet. For each word $\alpha \in \Sigma^*$, define the Parikh map of α to be the vector $\#(\alpha) = (\#_{a_1}(\alpha), \cdots, \#_{a_k}(\alpha))$, where each symbol count $\#_{a_i}(\alpha)$ denotes the number of symbol a_i's in α. For a language $L \subseteq \Sigma^*$, the Parikh map of L is the set $\#(L) = \{\#(\alpha) : \alpha \in L\}$. The language L is semilinear if $\#(L)$ is a semilinear set. Example semilinear languages are regular languages, context-free languages and the languages accepted by reversal-bounded NCMs.

Theorem 2.5. *Let M be a 2NFA and $\mathbf{c}$ be an array of rational numbers between 0 and 1 (inclusive). Then, $L_{\mathrm{run},\mathbf{c}(M)}$ is (effectively) a regular language.*

Proof. Without loss of generality, we assume that M enters an accepting state and returns to the right end (i.e., the right end marker) of an input word when it accepts the word. Also without loss of generality, we assume that $\mathbf{c}$ contains the two end points 0 and 1, and there are k distinct numbers in $\mathbf{c}$ with $0 = c_1 < \cdots < c_k = 1$. Implicitly, they represent k positions on the input. Clearly, if the input word w is long enough, the k sampling positions given by $\lfloor c_i(|w|+1) \rfloor$ must be distinct. Since there are only finitely many "short" w's, it suffices for us to assume that the k sampling positions are indeed distinct. To ease our presentation, we further assume that these positions are marked; i.e., when originally the symbol at such a position is a, in the marked input, the symbol is a marked new symbol $\dot{a}$. By assumption, the two ends of the input are also marked (for convenience, we assume now that the input already includes the two end markers). On a marked input, a *block* is a subword between two neighboring marked symbols where the two marked symbols are included; e.g., when the input is $\dot{a}bcb\dot{d}ca\dot{c}$, we have two blocks: $\dot{a}bcb\dot{d}$ and $\dot{d}ca\dot{c}$. When M works on a marked input, it works as if the marked symbols are unmarked. Hence, marked symbols do not alter the behaviors of M.

In particular, for a marked input in the form of

$$\dot{b}_1 u_1 \cdots \dot{b}_{k-1} u_{k-1} \dot{b}_k, \tag{2}$$

we assume that each marked symbol $\dot{b}_i$ implicitly contains its index i in the mark. Hence, even if b_i and b_j, $i \neq j$, are the same symbol, $\dot{b}_i$ and $\dot{b}_j$ are different.

We now consider a block, say $\dot{b}_i u_i \dot{b}_{i+1}$, where $1 \leq i < k$. Similar to the proof of Lemma 2.4, we define the following sets, shown in Fig. 1 and described below.

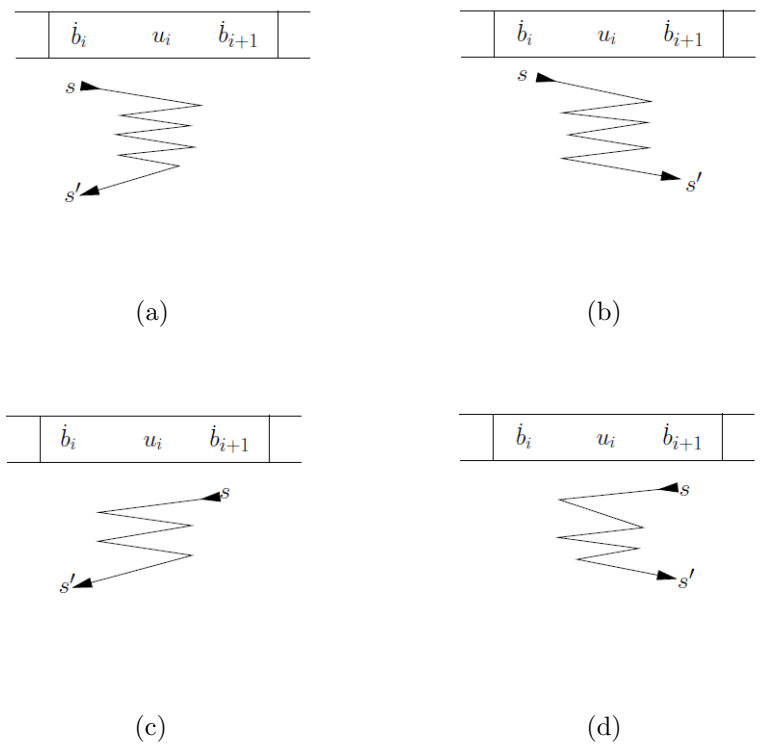

Figure 1: Generating regular sets when reading the block $\dot{b}_i u_i \dot{b}_{i+1}$.

(a) $[s, \dot{b}_i, s', \text{rstay}]$ is the set of u_i such that there is a marked input in (2) on which there is an accepting run of M satisfying the following. There is a time when M is reading $\dot{b}_i$ at state s, M will eventually come back to this $\dot{b}_i$. During this period, M's read head is under $\dot{b}_i$ only twice and, in between, the head never moves out of the right of u_i and never moves out of the left of $\dot{b}_i$.

(b) $[s, \dot{b}_i, s', \text{right}]$ is the set of u_i such that there is a marked input in (2) on which there is an accepting run of M satisfying the following. There is a time when M is reading $\dot{b}_i$ at state s, M will eventually go to the next marked symbol $\dot{b}_{i+1}$. During this period, M's read head is under $\dot{b}_i$ and under $\dot{b}_{i+1}$ exactly once, respectively, and, in between, the head never moves out of the right of $\dot{b}_{i+1}$ and never moves out of the left of $\dot{b}_i$.

(c) $[s, \dot{b}_{i+1}, s', \text{left}]$ is the set of u_i such that there is a marked input in (2) on which there is an accepting run of M satisfying the following. There is a time when M is reading $\dot{b}_{i+1}$ at state s, M will eventually go to the marked symbol $\dot{b}_i$. During this period, M's read head is under $\dot{b}_i$ and under $\dot{b}_{i+1}$ exactly once, respectively, and, in between, the head never moves out of the right of $\dot{b}_{i+1}$ and never moves out of the left of $\dot{b}_i$.

(d) $[s, \dot{b}_{i+1}, s', \text{lstay}]$ is the set of u_i such that there is a marked input in (2) on which there is an accepting run of M satisfying the following. There is a time when M is reading $\dot{b}_{i+1}$ at state s, M will eventually come back to this $\dot{b}_{i+1}$. During this period, M's read head is under $\dot{b}_{i+1}$ only twice and, in between, the head never moves out of the left of u_i and never moves out of the right of $\dot{b}_{i+1}$.

Again, it is an exercise to show that every set (which could be empty) defined above is regular. We now consider a sequence of k marked symbols, say $B = \dot{b}_1 \cdots \dot{b}_k$ and a marked word $w = \dot{b}_1 u_1 \cdots u_{k-1} \dot{b}_k$. We construct a table T_w which consists of all tuples, for all states s, s', and for all $1 \leq i < k$,

- $(s, \dot{b}_i, s', \text{right})$ if $u_i \in [s, \dot{b}_i, s', \text{right}]$;
- $(s, \dot{b}_i, s', \text{rstay})$ if $u_i \in [s, \dot{b}_i, s', \text{rstay}]$;
- $(s, \dot{b}_{i+1}, s', \text{left})$ if $u_i \in [s, \dot{b}_{i+1}, s', \text{left}]$;
- $(s, \dot{b}_{i+1}, s', \text{lstay})$ if $u_i \in [s, \dot{b}_{i+1}, s', \text{lstay}]$.

Notice that T_w defines the transition table for a two-way finite automaton (Both rstay- and lstay- transitions are staying transitions where the read head does not move. We artificially "distinguish" two kinds of staying transitions in order to ease the subsequent proof), also denoted by T_w, working on the input B (of fixed length k). The initial and accepting states of T_w are the same as those in M. Note that, by the construction of T_w,

(*) the accepting runs of M on w, sampled at positions corresponding to the marked symbols $\dot{b}_1, \cdots, \dot{b}_k$, are exactly the accepting runs, denoted by $\mathrm{run}(T_w)$, of T_w on word B.

Also note that there are only finitely many B's (since its length is a constant k) and, by definition of T_w, there are only finitely many distinct T_w's. Using the statement in (*), we have the following brute-force procedure to compute $L_{\mathrm{run},\mathbf{c}(M)}$.

We fix a $B = \dot{b}_1 \cdots \dot{b}_k$ and consider a Boolean function $\mathcal{B}$ which gives the following Boolean values $\mathcal{B}(s, \dot{b}_i, s', \mathrm{right})$, $\mathcal{B}(s, \dot{b}_i, s', \mathrm{rstay})$, $\mathcal{B}(s, \dot{b}_{i+1}, s', \mathrm{left})$, and $\mathcal{B}(s, \dot{b}_{i+1}, s', \mathrm{lstay})$, for all $1 \leq i < k$ and all states s and s'. There are only finitely many such distinct Boolean functions. This is because the domain of the function is finite for the given state set and input alphabet of the 2NFA M as well as the constant k specified in the given $\mathbf{c}$. Each such Boolean function $\mathcal{B}$ defines a two-way finite automaton $T_{B,\mathcal{B}}$ working on B, whose transitions are exactly those t that make $\mathcal{B}(t)$ true. Clearly, not every $\mathcal{B}$ is valid. To clarify this, we first define the *consistent constraint* $Q_{B,\mathcal{B}}$. More precisely, $Q_{B,\mathcal{B}}(n, p_1, \cdots, p_k)$ holds if and only if there is a marked word $w = \dot{b}_1 u_1 \cdots \dot{b}_{k-1} u_{k-1} \dot{b}_k$ such that:

- The marked symbols in w are the sampling positions defined by the p_i's. That is, $n = |w|$ and, for all $1 \leq i \leq k$,

$$p_i = \Sigma_{j=1}^{i-1}(|u_j| + 1). \tag{3}$$

  Observe that $p_1 = 0, p_k = n - 1$, which are consistent with our earlier assumption of $c_1 = 0, c_k = 1$.

- w is *consistent* with the $\mathcal{B}$; i.e., for all states s and s', and for $1 \leq i < k$,
  - $\mathcal{B}(s, \dot{b}_i, \mathrm{right}, s')$ iff $u_i \in [s, \dot{b}_i, \mathrm{right}, s']$;
  - $\mathcal{B}(s, \dot{b}_i, \mathrm{rstay}, s')$ iff $u_i \in [s, \dot{b}_i, \mathrm{rstay}, s']$;
  - $\mathcal{B}(s, \dot{b}_{i+1}, \mathrm{left}, s')$ iff $u_i \in [s, \dot{b}_{i+1}, \mathrm{left}, s']$;
  - $\mathcal{B}(s, \dot{b}_{i+1}, \mathrm{lstay}, s')$ iff $u_i \in [s, \dot{b}_{i+1}, \mathrm{lstay}, s']$.

Now, $\mathcal{B}$ is *valid* wrt B if there are $n, p_1, \cdots, p_k$ satisfying

$$\bigwedge_{1 \leq i \leq k} \lfloor c_i((n-2)+1) \rfloor = p_i \wedge Q_{B,\mathcal{B}}(n, p_1, \cdots, p_k), \tag{4}$$

where the equations $\lfloor c_i((n-2)+1) \rfloor = p_i$ say that the p_i's are exactly the sampling positions defined by the given vector $\mathbf{c}$; noticing that the $\dot{b}_1$ is the left end marker and the $\dot{b}_k$ is the right end marker. Hence, the length of the original input word that excludes the two end markers equals $n-2$.

Using the aforementioned statement (*), we have

$$L_{\mathrm{run},\mathbf{c}(M)} = \bigcup_{B} \bigcup_{\substack{\mathcal{B} \text{ is valid} \\ \text{wrt } B}} \mathrm{run}(T_{B,\mathcal{B}}). \tag{5}$$

Since each $\mathrm{run}(T_{B,\mathcal{B}})$ is regular, the result follows after we show that it is decidable (i.e., there is an algorithm to test) whether a given $\mathcal{B}$ is valid wrt a given B (under the 2NFA M and the $\mathbf{c}$ given in the theorem).

We now prove that the decidability indeed holds. We now claim that

> (**) The consistent constraint $Q_{B,\mathcal{B}}(n, p_1, \cdots, p_k)$ is a Presburger formula.

Notice that once the claim is established, the formula in (4) is also Presburger (the rational numbers in $\mathbf{c}$ are constants such as $\frac{3}{4}$) and hence the decidability follows.

To show the claim, we first notice that the set, denoted by $L_{B,\mathcal{B}}$, of w's being consistent with $\mathcal{B}$ is a regular language. This is because, as we have mentioned earlier, sets like $[s, \dot{b}_i, \mathrm{right}, s']$ in defining the language are regular. Suppose that $L_{B,\mathcal{B}}$ is accepted by a (one-way) finite automaton M'. Recall that each w in $L_{B,\mathcal{B}}$ is in the form of $w = \dot{b}_1 u_1 \cdots \dot{b}_{k-1} u_{k-1} \dot{b}_k$. We now construct a reversal-bounded NCM M'' that has counters $x_1, \cdots x_k$ which are initially zero, and works on an input word with k unary blocks in the form of $e_1^{p_1} \cdots e_k^{p_k}$, for some $p_1, \cdots, p_k$, where $e_1, \cdots, e_k$ are new symbols. M'' first guesses a word $w = \dot{b}_1 u_1 \cdots \dot{b}_{k-1} u_{k-1} \dot{b}_k$, one symbol by one symbol, and simulates M' on the guessed word. During the simulation, M'' increments each x_i by one on every symbol guesses. However, for each i, after the i-th marked symbol $\dot{b}_i$ is guessed, M'' nolonger increments x_i anymore. (Hence now, x_i stores the RHS of (3).) Nondeterministically, M'' stops guessing. At this time, M'' makes sure that M' accepts the guessed word. Now, M'' starts to read its own input $e_1^{p_1} \cdots e_k^{p_k}$. For each i, whenever an e_i is read, counter x_i is decremented by one. At the end of the input, M'' accepts when all the counters are zero. Observe that the counters in M'' are indeed reversal-bounded. Clearly, M'' accepts $e_1^{p_1} \cdots e_k^{p_k}$ iff there is a $w = \dot{b}_1 u_1 \cdots \dot{b}_{k-1} u_{k-1} \dot{b}_k$ consistent with $\mathcal{B}$ such that the equation in (3) holds for all $1 \leq i \leq k$. All the

tuples $(p_1, \cdots, p_k)$ with $e_1^{p_1} \cdots e_k^{p_k} \in L(M'')$ are then definable by a Presburger formula Q since the language $L(M'')$, being accepted by a reversal-bounded NCM is known semilinear [16]. Now, it is clear that the consistent constraint $Q_{B,\mathcal{B}}(n, p_1, \cdots, p_k)$ can be expressed as $n - 1 = p_k \wedge Q(p_1, \cdots, p_k)$ and hence the claim in (**) follows. □

Now, we turn back to the traces of a 2NFA M. In M, the $\mathbf{c}$-*sampling information flow* from A to B is defined as $F(A; B|L_{\text{run},\mathbf{c}(M)})$. Clearly, from Theorem 2.5 and Theorem 1.1, $F(A; B|L_{\text{run},\mathbf{c}(M)})$ is computable from the given $\mathbf{c}$ and M.

We now consider the following *maximal sampling problem* using position sampling:

Given: a 2NFA M, a variable set A in M, and a number $k \geq 1$.

Goal: Find a k-ary vector $\mathbf{c}$ such that the information rate of $L^A_{\text{run},\mathbf{c}(M)}$ is maximal (among all choices of $\mathbf{c}$).

The maximal sampling problem is to find a best set of k positions in order to maximally maintain the bit rate for the given set of (observable) state variables. Similarly, the *information flow problem using position sampling* is as follows:

Given: a 2NFA M, two disjoint variable sets A and B in M, and a number $k \geq 1$.

Goal: Find a k-ary vector $\mathbf{c}$ such that $F(A; B|L_{\text{run},\mathbf{c}(M)})$ is maximal (among all choices of $\mathbf{c}$).

Intuitively, the goal here is to find best input positions (for the given k) in order to observe the maximal information flow.

Theorem 2.6. *For 2NFAs, maximal sampling and maximal information flow problems using position sampling are solvable.*

Proof. Let M be a 2NFA. Without loss of generality, similar to the proof Theorem 2.5, we assume that in a $\mathbf{c}$, $c_1 = 0$ and $c_k = 1$. Closely inspecting the proof of Theorem 2.5, there are only finitely many distinct regular languages $L_{\text{run},\mathbf{c}(M)}$, as shown in (5), whereas there are infinitely many choices of $\mathbf{c}$. This is because that there are only finitely many choices for B and $\mathcal{B}$ and hence $T_{B,\mathcal{B}}$ (for the given k and M). Each such choice does not depend on $\mathbf{c}$. However, whether $\mathcal{B}$ is valid wrt B for a given pair of B and $\mathcal{B}$ does depend on the choice of $\mathbf{c}$. More precisely, we say that $V_{B,\mathcal{B}}(\mathbf{c})$ holds if, under the $\mathbf{c}$, $\mathcal{B}$ is valid wrt B, as defined in (4). Consequently to emphasize this fact,

formula (5) is re-written into

$$L_{\mathrm{run},\mathbf{c}(M)} = \bigcup_{B} \bigcup_{\mathcal{B}:\ V_{B,\mathcal{B}}(\mathbf{c})} \mathrm{run}(T_{B,\mathcal{B}}). \tag{6}$$

Recall the definition of $V_{B,\mathcal{B}}(\mathbf{c})$ in (4) which is now shown again below:

$$\exists n, p_1, \cdots, p_k. \quad \bigwedge_{1 \le i \le k} \lfloor c_i(n-1) \rfloor = p_i \wedge Q_{B,\mathcal{B}}(n, p_1, \cdots, p_k). \tag{7}$$

Hence, each $\mathbf{c}$ with $V_{B,\mathcal{B}}(\mathbf{c})$ can be "represented" as a tuple $(n, p_1, \cdots, p_k)$ that satisfies(7).

Let D be a total function that assigns a Boolean value to each pair of B and $\mathcal{B}$. There are also finitely many distinct D's. Notice that the D does not depend on $\mathbf{c}$. To relate D with $\mathbf{c}$, we re-write formula (6) into

$$L_{\mathrm{run},\mathbf{c}(M)} = \bigcup_{B} \bigcup_{\mathcal{B}:D(B,\mathcal{B})} \mathrm{run}(T_{B,\mathcal{B}}), \tag{8}$$

where the D in the RHS of (8) satisfies the following:

$$\text{for all } B \text{ and } \mathcal{B}, \quad V_{B,\mathcal{B}}(\mathbf{c}) = D(B, \mathcal{B}). \tag{9}$$

Hence, for each given D, all the $\mathbf{c}$'s that make the statement in (9) true will result in the same language $L_{\mathrm{run},\mathbf{c}(M)}$ defined in the RHS of (8). That is, there are only finitely many distinct regular languages $L_{\mathrm{run},\mathbf{c}(M)}$ for all choices of $\mathbf{c}$.

To complete the proof, we need only show that, given a D, we can effectively compute a $\mathbf{c}_D$ (if exists) that satisfies the statement in (9). Once this is done, to solve the maximal sampling problem, we need only enumerate all the D's and for each D, compute the $\mathbf{c}_D$, the regular language $L_{\mathrm{run},\mathbf{c}_D(M)}$ (if $\mathbf{c}_D$ does not exists, this language is simply empty), and the regular language $L^A_{\mathrm{run},\mathbf{c}_D(M)}$. The information rate for $L^A_{\mathrm{run},\mathbf{c}_D(M)}$ can then be computed and denoted by $\lambda_{D,\mathbf{c}_D}$. The maximal sampling problem is solved by returning a $\mathbf{c}_D$ that makes $\lambda_{D,\mathbf{c}_D}$ maximal among all the D's. Notice that if $D(B, \mathcal{B}) = false$ for all B and $\mathcal{B}$, then the RHS of (9) and hence $\lambda_{D,\mathbf{c}_D}$ is always $\emptyset$ no matter what the $\mathbf{c}_D$ is. This makes the information rate of $L^A_{\mathrm{run},\mathbf{c}_D(M)}$ (the information flow as well) being zero. Therefore, below, we only enumerate the D's satisfying that there are $B^0, \mathcal{B}^0$ satisfying $D(B^0, \mathcal{B}^0) = true$.

The maximal information flow problem can be solved similarly.

We now show how to compute a $\mathbf{c}_D$ that satisfies the statement in (9), for a given D. Combining (9) and (7), to compute the $\mathbf{c}_D$, we need only compute a $\mathbf{c}$ satisfying, for the given D,

$$\forall B, \mathcal{B}. \qquad D(B, \mathcal{B}) \text{ is true iff } \exists n, p_1, \cdots, p_k.$$

$$\bigwedge_{1 \leq i \leq k} \lfloor c_i(n-1) \rfloor = p_i \wedge Q_{B,\mathcal{B}}(n, p_1, \cdots, p_k), \tag{10}$$

recalling that $Q_{B,\mathcal{B}}(n, p_1, \cdots, p_k)$ is a Presburger formula. The computation is straightforward as follows. First, we observe that, according to the definition of $Q_{B,\mathcal{B}}$, $Q_{B,\mathcal{B}}(n, p_1, \cdots, p_k)$ already implies that $0 = p_1 < p_2 < \cdots < p_k = n-1$. Second, once $0 = p_1 < p_2 < \cdots < p_k = n-1$, we can always take

$$c_i = \frac{p_i}{n-1} \tag{11}$$

to make $\lfloor c_i(n-1) \rfloor = p_i$ true in (10). This really says that we need only find values $p_1^0, \cdots, p_k^0, n^0$ satisfying

$$\bigwedge_{B,\mathcal{B}} \quad Q_{B,\mathcal{B}}(n^0, p_1^0, \cdots, p_k^0) \text{ iff } D(B, \mathcal{B}). \tag{12}$$

This is because, since, as we assumed, there are $B^0, \mathcal{B}^0$ making $D(B^0, \mathcal{B}^0)$ true. Hence, the $\mathbf{c}_D$ can be computed using (11).

For the given D, the formula in (12) is a Presburger formula in $p_1^0, \cdots, p_k^0, n^0$ since each $D(B, \mathcal{B})$ is a Boolean value. The Presburger formula in (12) is then denoted as $Q_D(p_1^0, \cdots, p_k^0, n^0)$. It is well-known that solving a Presburger formula like $Q_D(p_1^0, \cdots, p_k^0, n^0)$ for nonnegative integer solutions can be done algorithmically (i.e., using tools for integer linear programming after eliminating quantifiers in the formulas). The result follows. □

Observe that the set of sampling positions specified in $\mathbf{c}$ is a linear function of the input length, while there are no nontrivial relationships between these positions. We now consider cases where the k sampling positions are constrained. For an input of length n, let $0 = p_1 < \cdots < p_k = n-1$ be the k sampling positions on the input. Let $\mathbf{C}(p_1, \cdots, p_k, n)$ be a Presburger formula to specify a *sampling position constraint* satisfying the following condition that $(p_1, \cdots, p_k)$ is a (not necessarily total) function of n:

For all n, there are no $(p_1, \cdots, p_k) \neq (p'_1, \cdots, p'_k)$ such that $\mathbf{C}(p_1, \cdots, p_k, n)$ and $\mathbf{C}(p'_1, \cdots, p'_k, n)$ hold.

(The condition can be verified algorithmically for this $\mathbf{C}$ since the condition itself is a closed Presburger formula.) Hence, once the input length is given, the sampling positions, if they exist, are unique, hence they depend only on the length. For instance, such a $\mathbf{C}$ can be the following constraint

$$p_2 - p_1 = \cdots = p_k - p_{k-1}$$

indicating that all sampling positions are distributed roughly evenly on the input. For notational convenience, most of the previous definitions involving $\mathbf{c}$ are reused for $\mathbf{C}$. For instance, for the 2NFA M, $L_{\mathrm{run},\mathbf{C}(M)}$ still denotes the set of all $\mathbf{C}$-sampling (at positions $p_1, \cdots, p_k$ satisfying $\mathbf{C}$) accepting runs for all inputs. We can show Theorem 2.5 can be generalized.

Theorem 2.7. *Let M be a 2NFA and $\mathbf{C}$ be a Presburger sampling position constraint. Then, $L_{\mathrm{run},\mathbf{C}(M)}$ is a regular language.*

Proof. Let M be a 2NFA. Again, we look at the proof of Theorem 2.5, where the first and the only place that we really use $\mathbf{c}$ (to constrain the sampling positions) is in (4). Certainly, after we replace the constraint $\bigwedge_{1 \leq i \leq k} \lfloor c_i((n-2)+1) \rfloor = p_i$ in (4) by the Presburger formula $\mathbf{C}(p_1, \cdots, p_k, n)$ and replace $\mathbf{c}$ by $\mathbf{C}$ in (5), the rest of the proof still goes through. □

To formulate the maximal sampling problem, let $\mathbf{C}(p_1, \cdots, p_k, n; t_1, \cdots, t_m)$ be a Presburger formula, where $p_1, \cdots, p_k$ are sampling positions, and $t_1, \cdots, t_m$ are parameters. This $\mathbf{C}$ further satisfies a condition such that, for any given $t_1, \cdots, t_m$, $(p_1, \cdots, p_k)$ is a (not necessarily total) function of n. Again, such a condition can be verified algorithmically for this $\mathbf{C}$. Notice that $L_{\mathrm{run},\mathbf{C}(M)}$ depends on the values of the parameters. Now, we define the *maximal sampling problem* using Presburger position sampling:

Given: a 2NFA M, a variable set A in M, and a Presburger sampling constraint $\mathbf{C}$ with parameters $t_1, \cdots, t_m$.

Goal: Find values for parameters $t_1, \cdots, t_m$ such that the information rate of $L^A_{\mathrm{run},\mathbf{C}(M)}$ is maximal (among all choices of the values of the parameters).

Similarly, the *information flow problem using Presburger position sampling* is as follows:

Given: a 2NFA M, two disjoint variable sets A and B in M, and a Presburger sampling constraint $\mathbf{C}$ with parameters $t_1, \cdots, t_m$.

Goal: Find values for parameters $t_1, \cdots, t_m$ such that $F(A; B|L_{\text{run},\mathbf{C}(M)})$ is maximal (among all choices of the values of the parameters).

Theorem 2.6 can be generalized.

Theorem 2.8. *For 2NFAs, maximal sampling and maximal information flow problems using Presburger position sampling are solvable.*

Proof. Let M be a 2NFA. Similar to the proof of Theorem 2.7, again, we look at the proof of Theorem 2.5. Let us first fix values for the parameters $t_1, \cdots, t_m$. Then, $\mathbf{C}$ is a Presburger constraint over the sampling positions $p_1, \cdots, p_k$ and input length n. As we have mentioned earlier, in the proof of Theorem 2.5, the first and the only place that we really use $\mathbf{c}$ (to constrain the sampling positions) is in (4). We replace the constraint with $\mathbf{C}$ and modify (4) into

$$\mathbf{C}(p_1, \cdots, p_k, n; t_1, \cdots, t_m) \wedge Q_{B,\mathcal{B}}(n, p_1, \cdots, p_k). \tag{13}$$

Now, we can re-write (5) into

$$L_{\text{run},\mathbf{C}(M)} = \bigcup_{B} \bigcup_{\substack{\mathcal{B} \text{ is valid} \\ \text{wrt } B}} \text{run}(T_{B,\mathcal{B}}). \tag{14}$$

Recall that B and $\mathcal{B}$ (which are only finitely many), from the proof, only depends on M (not on the parameters). However, the condition in (13) that $\mathcal{B}$ is valid wrt B does depend on the choice of the parameters. Therefore, depending on the truth of the conditions, $L_{\text{run},\mathbf{C}(M)}$ can only be one of finitely many different languages, among all parameter values. That is, each such language can be decided by a function D that maps $D(B, \mathcal{B})$ to true or false, for each B and $\mathcal{B}$. Herein, $D(B, \mathcal{B})$ equals the truth of the condition, denoted by V, in (13). Therefore,

$$\bigwedge_{B,\mathcal{B}} V(p_1, \cdots, p_k, n; t_1, \cdots, t_m) \text{ iff } D(B, \mathcal{B}) \tag{15}$$

defines a Presburger formula $P_D(p_1, \cdots, p_k, n; t_1, \cdots, t_m)$ for a fixed D. All the parameter values $(t_1, \cdots, t_m)$ satisfying

$$\exists p_1, \cdots, p_k, n: \; P_D(p_1, \cdots, p_k, n; t_1, \cdots, t_m) \tag{16}$$

will result in the same language $L_{\text{run},\mathbf{C}(M)}$. Therefore, the result is immediate since one can enumerate all the D's and for each such D, solving the Presburger formula in (16) for parameter values $t_1, \cdots, t_m$, and then compute the

regular language $L_{\text{run},\mathbf{C}(M)}$ under the parameter values using (14). Finally, we compute the information rate of $L^A_{\text{run},\mathbf{C}(M)}$. The computed parameter values that achieve the maximal information rate for all the enumerated D's can then be found.

The maximal information flow problem can be solved similarly. □

3. Decision Questions Concerning Sampled Runs for Some Two-way Infinite-state Automata

In this section, we show that position sampling may result in quite complex languages when the 2NFA is augmented with unbounded storage. We start with finite-crossing finite automata augmented with reversal-bounded counters [16]. Finite-crossing here means that there is some fixed k such that in every accepting computation on any input, the number of times the input head crosses the boundary between any two adjacent symbols of the input is at most k. Note that the number of turns (i.e., changes in direction from left-to-right and right-to-left and vice-versa) the input head makes on the input may be unbounded. We assume that when we are given a finite-crossing machine, the integer k for which the machine is k-crossing is also specified.

Let M be a two-way acceptor, whose input, as we have mentioned earlier, is with end markers; i.e., in the form of $w = \triangleright a_1 \cdots a_n \triangleleft$ for some n, where $a_1 \cdots a_n$ is an input word. As usual, the left and right end markers will be at position 0 and $n+1$, respectively. For $0 \leq c \leq n+1$, let $\text{cr}(M, c) = \{s_1 \cdots s_m \mid w$ is an input (includes the end markers), M halts and accepts w in some run where the visiting sequence of states at position c is $s_1, \ldots, s_m\}$. We now assume that the symbol at position c is marked and it is the only marked symbol on the input. That is, the machine M "knows" the sampling position c (since it is marked). In this way, $\text{cr}(M, c)$ is simply written $\text{cr}(M)$.

Theorem 3.1.

1. *It is decidable, given a 2DFA (i.e., a two-way DFA) with one reversal-bounded counter (no restriction on the two-way input) M and a regular language L, whether $\text{cr}(M) \cap L$ is empty;*
2. *It is undecidable, given a 2DFA with two reversal-bounded counters (no restriction on the two-way input) M and and a singleton (hence, regular) language L, whether $\text{cr}(M) \cap L$ is empty;*
3. *It is decidable, given a finite-crossing reversal-bounded 2NCM (i.e., a finite-crossing 2NFA with multiple reversal-bounded counters) M and a regular language L, whether $\text{cr}(M) \cap L$ is empty.*

Proof. For the first part, given a 2DFA with one reversal-bounded counter M and a DFA A accepting L, we construct a 2DFA with one reversal-bounded counter M' which on input w, simulates M. Whenever M is at the "marked position" under which the input symbol is marked (there is exactly one such position, as we have mentioned earlier) in state, say s, it simulates the DFA A's transition on s. M' accepts w if M accepts w and the visiting sequence at the marked position is accepted by A. The definition of M' directly gives that

$$L(M') = \emptyset \text{ iff } \mathrm{cr}(M) \cap L = \emptyset. \tag{17}$$

The result follows, since $L(M') = \emptyset$ is known decidable (the emptiness problem for 2DFAs with one reversal-bounded counter is decidable [17]).

Part 2 follows from the fact that the emptiness problem for 2DFAs with two reversal-bounded counters is undecidable [16]. Let M_0 be any 2DFA with two reversal-bounded counters and input alphabet Σ. Construct a 2DFA M with two reversal-bounded counters and input alphabet $\Sigma \cup \{a_1, a_2\}$, where a_1 and a_2 are new symbols. M when given input $\triangleright w \triangleleft$ first checks that a prefix of w is of the form xa_1 for some $x \in \Sigma^*$. Then M simulates the computation of M_0 on x treating a_1 as the right end marker in the simulation of M_0. When M_0 accepts x, M moves to the right of a_1 in a unique state f and accepts if the next two symbols to the right of a_1 are a_2 and $\triangleleft$; i.e., the input is $\triangleright x a_1 a_2 \triangleleft$. Now let a_2 be the "marked position" and let $L = \{f\}$. Then $\mathrm{cr}(M) \cap L$ is empty if and only if $L(M_0)$ is empty, which is undecidable.

The proof for Part 3 is similar to Part 1. That is, now, M is a finite-crossing reversal-bounded 2NCM, and the constructed M' is also a finite-crossing reversal-bounded 2NCM. From (17), the result follows, since the emptiness of $L(M')$ is known decidable (the emptiness problem for finite-crossing reversal-bounded 2NCMs is decidable [16]). □

Remark: Theorem 3.1, part 3 can be strengthened. The result holds also when L is accepted by a one-way NFA M_L with multiple reversal-bounded counters. As described in the proof of part 1, the finite-crossing M' now simulates M_L's transitions on s.

So far in this section, we have only considered the case when there is only one marked symbol in an input (i.e., there is only one sampling position). We now consider the case where the marked symbols on an input are unbounded. Let Σ_1 be an alphabet and $\Sigma_2 = \{a' \mid a \in \Sigma_1\}$. Thus, the symbols in Σ_2 are the "marked" symbols in Σ_1. Let $\Sigma = \Sigma_1 \cup \Sigma_2$.

Let M be a two-way acceptor (with end markers) possibly with infinite storage over the input alphabet Σ. We assume that M on input w in Σ "ignores" the marks (treating symbols a and a' in the same way).

Let M_{CR} be another acceptor over Σ (one-way or two-way, possibly with infinite storage) over Σ. This time in its computation, M_{CR} distinguishes the marked symbols from the unmarked symbols. Thus, e.g., on input w, M_{CR} accepts if every other symbol is marked (or the number of marked symbols is even; etc.). Intuitively, this M_{CR} specifies where the marked symbols are on an input to M.

Let L be a language over sequences of states of M (specified by another acceptor M_L). Let $\mathrm{CR}(M)$ be the set of visiting sequences (of states) for all marked input symbols (i.e., such a visiting sequence is the result of sampling all the input positions where the input symbols are marked) in an accepting run of M.

Theorem 3.2. *It is decidable, given a finite-crossing reversal-bounded 2NCM M, a finite-crossing reversal-bounded 2NCM M_{CR}, and a reversal-bounded NCM M_L accepting a language L, whether* $\mathrm{CR}(M) \cap L$ *is empty.*

Proof. The proof is a generalization of the proof of Part 1 of Theorem 3.1. Given M, M_{CR}, and M_L, we construct a finite-crossing reversal-bounded 2NCM M' that operates as follows, when given input w. M' first simulates M_{CR} and checks that w is accepted by M_{CR}. Then M' simulates M. Whenever M is at a marked input symbol, in state, say s, M' simulates a transition of M_L on s. M' accepts if M and M_L accept. □

Similarly, we have:

Theorem 3.3. *It is decidable, given a 2DFA with one reversal-counter M, a 2DFA with one reversal-bounded counter M_{CR}, and a DFA M_L accepting a language L, whether* $\mathrm{CR}(M) \cap L$ *is empty.*

Proof. We construct a 2DFA with one reversal-bounded counter M' which, on input w, simulates M_{CR} and checks that w is accepted by M_{CR}. Then M' simulates M and checks that M accepts w and the visiting sequence is accepted by M_L, The result follows since the emptiness of $L(M')$ (being a 2DFA with one reversal-bounded counter) is decidable [17]. □

A special case of the above is the following, which can be simply observed. If M is a one-way NFA and M_{CR} (which specifies the "marked" positions) is

an NFA, then CR(M) can be accepted by a one-way NFA (and hence it is regular).

4. More on $L_{\text{run},M}$ for 2DFAs M

Recall that for a 2NFA M, $L_{\text{run},M}$ is not necessarily regular. In fact, consider a 1-turn 2DFA M that accepts an input if it falls off the left end marker in an accepting state. Thus, M makes a left-to-right sweep of the input followed by a right-to-left sweep, and it has no stationary moves. We have:

Proposition 4.1. *Let $\Sigma = \{a, b\}$. Σ^+ can be accepted by a 1-turn 2DFA M such that $L_{\text{run},M}$ cannot be accepted by any finite-crossing reversal-bounded 2NCM.*

Proof. The 1-turn 2DFA M has states $s, q_a, q_b, r, p_a, p_b, f$, with s the starting state and f the only accepting state. On an input $\triangleright w \triangleleft$, M uses the following transitions, where +1 (resp., -1) denotes right (resp., left) move:

$$(s, \triangleright) \to (s, +1);\ (s, a) \to (q_a, +1);\ (s, b) \to (q_b, +1);$$

$$(q_a, a) \to (q_a, +1);\ (q_a, b) \to (q_b, +1);$$
$$(q_b, a) \to (q_a, +1);\ (q_b, b) \to (q_b, +1);$$

$$(q_a, \triangleleft) \to (r, -1);\ (q_b, \triangleleft) \to (r, -1);$$

$$(r, a) \to (p_a, -1);\ (r, b) \to (p_b, -1);$$

$$(p_a, a) \to (p_a, -1);\ (p_a, b) \to (p_b, -1);$$
$$(p_b, a) \to (p_a, -1);\ (p_b, b) \to (p_b, -1);$$

$$(p_a, \triangleright) \to (f, -1);\ (p_b, \triangleright) \to (f, -1).$$

Intuitively, M, on a non-null input word with end markers, makes a left-to-right sweep (from $\triangleright$ all the way to $\triangleleft$) and then a right-to-left sweep. It accepts when, on the end of the second sweep, it falls off (the last two transitions) the left end marker. On each sweep, a state "memorizes" the input symbol that has just been read as well as the direction of the sweep. For instance, on input $\triangleright aab \triangleleft$, the resulting accepting run is the following state sequence: $ssq_aq_aq_brp_bp_ap_af$.

Clearly, M accepts Σ^+, and it makes only one turn on the input. We claim that $L_{\text{run},M}$ cannot be accepted by a finite-crossing reversal-bounded 2NCM. Otherwise, we can convert this finite-crossing reversal-bounded 2NCM to an equivalent (one-way) reversal-bounded NCM [15]. Then we apply a homomorphism to $L_{\text{run},M}$ that maps q_a and p_a to a, q_b and p_b to b, and all other states to ϵ. The resulting image language is $L = \{xx^R \mid x \in \Sigma^+\}$, and clearly, it should be accepted by a reversal-bounded NCM M_L (x^R is the reverse of word x). Then there is a constant d such that every string of length n accepted by a reversal-bounded NCM (such as M_L) can be accepted in a computation within $d \cdot n$ steps [2]. We now consider an accepting computation (with at most $2d \cdot |x|$ steps) of M_L on input xx^R. Notice that the values of the counters in M_L can not exceed $2d \cdot |x|$ in the computation. In particular, we use α to denote the configuration (state and the values of its counters) in the accepting computation of M_L right after it reads x, the first half of the input. Let $k = |x|$. Clearly, the total number of distinct x with length k is exponential in k, while the total number of distinct α's for all accepting computations with at most $2d \cdot k$ steps of M_L on all inputs xx^R with $|x| = k$ is polynomial in k. This immediately gives that there are $x \neq y$ (both with length k, for some k) such that there are two accepting computations (with at most $2d \cdot k$ steps) for xx^R and yy^R respectively, such that both computations end up with the same configuration α right after reading the first half of the inputs. As a result, $yx^R \notin L$ will be accepted by M_L, a contradiction, □

Proposition 4.2. Σ^+ *can be accepted by a 2-turn 2DFA M (i.e., makes a left-to-right sweep of the input, then a right-to-left sweep, and finally a left-to-right sweep and accepts when it falls off the right end marker in an accepting state) such that $L_{\text{run},M}$ cannot be accepted by any NPDA (nondeterministic pushdown automaton).*

Proof. The 2-turn DFA M can be constructed exactly as the 1-turn DFA constructed in the proof of Proposition 4.1, except that the last two transitions replaced by the following transitions, where t, u_a, u_b are additional states for the third sweep:

$$(p_a, \triangleright) \to (t, +1);\ (p_b, \triangleright) \to (t, +1);$$

$$(t, a) \to (u_a + 1);\ (t, b) \to (u_a, +1);$$

$$(u_a, a) \to (u_a, +1);\ (u_a, b) \to (u_b, +1);$$
$$(u_b, a) \to (u_a, +1);\ (u_b, b) \to (u_b, +1);$$
$$(u_a, \lhd) \to (f, +1);\ (u_b, \lhd) \to (f, +1).$$

For instance, on input $\rhd aab\lhd$, the resulting accepting run of the 2-turn M is the following state sequence: $ssq_aq_aq_brp_bp_ap_atu_au_au_bf$.

Suppose $L_{run,M}$ can be accepted by an NPDA. Then, applying a homomorphism which maps q_a and u_a to a and q_b and u_b to b and all other states to ϵ yields the language $L = \{xx \mid x \in \Sigma^+\}$, which is not a context-free language. This is a contradiction, since context-free languages are closed under homomorphism. □

A 1-reversal NPDA is an NPDA with the property that once the (pushdown) stack pops, it can no longer push.

Proposition 4.3. *If M is 1-turn 2NFA, then $L_{\text{run},M}$ can be accepted by an unambiguous 1-reversal NPDA M'.*

Proof. Let $x = s_1 \cdots s_n$ (sequence of states of M) be an input to M'. We may assume that n is even, since M has no stationary moves. Thus $x = s_1 \cdots s_m s_{m+1} \cdots s_{2m}$ for some $m \geq 1$. Furthermore, without loss of generality, we assume that the position m, which is guessed by M', is the cell containing the right end marker $\lhd$ and where M' makes a left turn to make a right-to-left sweep.

The NPDA M' when given input x operates as follows: For each $1 \leq i < m$, M' reads s_i and s_{i+1} and pushes $A_i = \{a \mid (s_i, a) \to (s_{i+1}, +1)$ is a transition in $M\}$. Hence, the stack alphabet of M' is the power set of the input alphabet of M. At some point, M' guesses and verifies that M is on $\lhd$ (i.e., at position m) and makes a left move from state s_m to state s_{m+1}. Then M' checks that the remaining input sequence $s_{m+1} \cdots s_{2m}$ is the state sequence that M goes through in a right-to-left sweep of its input. It does this by popping the stack and confirming that for $1 \leq j < m$, $(s_{m+j}, a) \to (s_{m+j+1}, -1)$ is a transition of M for some a in A_{m-j}, and s_{2m} is an accepting state of M.

To see that $L(M')$ is indeed the same as $L_{\text{run},M}$, we need establish two arguments. Firstly, $L_{\text{run},M} \subseteq L(M')$, which is obvious. Secondly, $L(M') \subseteq L_{\text{run},M}$, which needs some effort. From the construction of M', it shall be clear that, once $s_1 \cdots s_m s_{m+1} \cdots s_{2m}$ is accepted, the state sequence is the

accepting run of the 1-turn 2NFA M that works on input $a_1 \cdots a_{m-1}$ on the left-to-right sweep while works on (possibly different) input $b_1 \cdots b_{m-1}$ on the way back (the right-to-left sweep) and accepts, satisfying that both b_i and a_i are in the set A_i, for all $1 \leq i \leq m-1$. When, however, on the left-to-right sweep, M works on $b_1 \cdots b_{m-1}$ (instead of $a_1 \cdots a_{m-1}$), the state sequence is still an accepting run, which is an accepting run of M on input $b_1 \cdots b_{m-1}$. The second argument is therefore established.

M' is deterministic, except when it guesses the position m. Note also that if $x = s_1 \cdots s_m s_{m+1} \cdots s_{2m}$ is accepted by M', m is unique (since different m's will yield different lengths of x's because M has no stationary moves.). It follows that M' is an unambiguous 1-reversal NPDA. □

Corollary 4.4. *The information rate of $L_{\mathrm{run},M}$ is computable for any 1-turn 2NFA M.*

Proof. This follows from Proposition 4.3 and the fact that the information rate of an unambiguous linear context-free language (which is equivalent to an unambiguous 1-reversal NPDA language) is computable [19]. □

A 1-turn 2NFA M is *special* if it has the following property: For any states q and r, if $(q, \lhd) \to (r, -1)$ is a transition in M, then there is no transition $(r, a) \to (p, +1)$ for any $a \in \Sigma \cup \{\rhd, \lhd\}$ and state p. (Intuitively, this means that M moving left on $\lhd$ into state r in an accepting computation uniquely determines the location where the head moves left of the right end marker.) Clearly, for this case, in the proof of Proposition 4.3, M' need not guess the position m. Hence:

Corollary 4.5. *If M is a special 1-turn 2NFA, then $L_{\mathrm{run},M}$ can be accepted by a 1-reversal DPDA M'.*

Example 4.6. *The 1-turn 2DFA M in the proof of Proposition 4.1 (with states $\{s, q_a, q_b, r, p_a, p_b, f\}$) which accepts $\Sigma^+ = \{a, b\}^+$ is special; hence, $L_{\mathrm{run},M} = \{ssyrz^R f \mid y \in \{q_a, q_b\}^+, z \in \{p_a, p_b\}^+, y$ becomes z after we replace q_a by p_a and q_b by $p_b\}$ can be accepted by a 1-reversal DPDA by Corollary 4.5.*

Corollary 4.5 does not hold if M is not special. In fact, we can prove:

Proposition 4.7. *There is a 1-turn 2DFA M such that $L_{\mathrm{run},M}$ cannot be accepted by any DPDA, even when the DPDA is allowed ϵ-moves and there is no bound on stack reversal.*

Proof. We construct a 1-turn 2DFA M whose $L_{\mathrm{run},M}$ contains many accepting runs (state sequences) with common prefixes so that no DPDA is able to distinguish between these runs.

Without loss of generality, we assume that a 1-turn DFA crashes (without accepting the input) whenever it tries to make the second turn. This assumption will make the following construction more succinct. M has input alphabet $\{a, b\}$ and state set $\{s, q_a, p_a, p_b, f\}$ with the following transitions:

$$(s, \rhd) \to (s, +1);\ (s, a) \to (q_a, +1);$$

$$(q_a, a) \to (q_a, +1);\ (q_a, \lhd) \to (p_a, -1);$$

$$(p_a, a) \to (p_a, -1);\ (p_a, \rhd) \to (f, -1);$$

$$(q_a, b) \to (p_a, +1);\ (p_a, b) \to (p_a, +1);$$

$$(p_a, \lhd) \to (p_b, -1);\ (p_b, b) \to (p_b, -1);\ (p_b, a) \to (p_a, -1).$$

Intuitively, M works as follows. After reading the left end marker, it is going to read an a, and keeps doing so while remaining in state q_a. When the right end marker is read, it makes a turn and makes a right-to-left sweep while remaining in state p_a. However, if it reads a b after reading the first a-block, it keeps reading b's while remaining in state p_a. M is designed such that if it reads an a again after reading the b's, it will crash (trying to make the second turn). Therefore, in this case, when the right end marker is read, it makes a turn and makes a right-to-left sweep while remaining in state p_b and then in state p_a, before it accepts. That is, M is constructed in the way that it only accepts input in the form of $\rhd a^i b^j \lhd$ with $i > 0$ and $j \geq 0$. On input $\rhd a^i \lhd$, the accepting run is $ssq_a^i p_a^{i+1} f$. However, on input $\rhd a^i b^j \lhd$ ($j > 0$), the accepting run is $ssq_a^i p_a^j p_b^{j+1} p_a^i f$. Hence, $L_{\mathrm{run},M}$ is exactly the set $\{ssq_a^i p_a^{i+1} f : i > 0\} \cup \{ssq_a^i p_a^j p_b^{j+1} p_a^i f : i > 0, j > 0\}$.

Note that M is not special because of transitions $(q_a, \lhd) \to (p_a, -1)$ and $(p_a, b) \to (p_a, +1)$.

Suppose that $L_{\mathrm{run},M}$ is accepted by a DPDA. Let $R = \{ssq_a^i p_a^j f \mid i, j \geq 2\} \cup \{ssq_a^i p_a^j p_b^k p_a^n f \mid i, j, k, n \geq 2\}$. Clearly, R is regular; hence, $L' = L_{\mathrm{run},M} \cap R = \{ssq_a^i p_a^{i+1} f \mid i \geq 2\} \cup \{ssq_a^i p_a^j p_b^{j+1} p_a^i f \mid i, j \geq 2\}$ can be accepted by a DPDA and, by change of notation, $L_1 = \{ssa^i b^{i+1} f \mid i \geq 2\} \cup \{ssa^i b^j c^{j+1} b^i f \mid i, j \geq 2\}$ can also be accepted by a DPDA. Then, $L_2 = \{a^i b^{i+1} f \mid i \geq 2\} \cup \{a^i b^j c^{j+1} b^i f \mid i, j \geq 2\}$ (i.e., the prefix ss is deleted)

can also be accepted by a DPDA. Since DPDA languages are closed under right quotient with regular sets, $L_3 = \{a^i b^{i+1} \mid i \geq 2\} \cup \{a^i b^j c^{j+1} b^i \mid i, j \geq 2\}$ (i.e., the end symbol f is deleted) is also a DPDA language. However, we show below that L_3 cannot be accepted by a DPDA. □

Lemma 4.8. $L_3 = \{a^i b^{i+1} \mid i \geq 2\} \cup \{a^i b^j c^{j+1} b^i \mid i, j \geq 2\}$ *cannot be accepted by a DPDA (even when it is allowed ϵ- moves and the stack is unrestricted).*

Proof. As far as we know, no language similar to the language L_3 has been shown not recognizable by a DPDA in the literature. We provide a detailed argument here, as the technique might be applicable to showing that other (similar) languages cannot be recognized by DPDAs. This proof technique has not been used before.

Suppose that L_3 can be accepted by a DPDA M_3. We can then construct a DPDA M_4 accepting $L_4 = \{a^i b^{i+1} \mid i \geq 2\} \cup \{a^i b^{i+1} c^{i+2} b^i \mid i \geq 2\}$ which operates as follows:

1. M_4, when given an input, simulates M_3 faithfully. Clearly, when the input is $a^i b^{i+1}$ ($i \geq 2$), M_4 accepts. When the input is $a^i b^j$, where $j \neq i+1$, M_4 rejects.
2. Suppose that the b's are followed by c's. When M_4 reaches the first c, we have two cases:
   (a) When M_4 reaches the first c, it rejects the input if in the sequence of ϵ-moves between the last b and the first c no accepting state has been visited. (This means that the number of b's in the input is not equal to 1 + the number of a's.)
   (b) If when M_4 reaches the first c, the sequence of ϵ-moves between the last b and the first c an accepting state has been visited (this means that it has just processed $a^i b^{i+1}$), it simulates M_3 faithfully on the remaining input segment. So if the input is $a^i b^{i+1} c^{i+2} b^i$, M_4 accepts. It rejects if the input is $a^i b^{i+1} c^j b^k$, where $j \neq i+2$ or $k \neq i$.

Then $L_5 = L_4 \cap a^+ b^+ c^+ b^+ = \{a^i b^{i+1} c^{i+2} b^i \mid i \geq 2\}$ is also a DPDA language. This is a contradiction, since it is easily shown, by the pumping lemma, that L_5 is not a context-free language. □

For one-way machines:

Proposition 4.9. *If M is an NPDA (resp., NPCM, NFA, NCM), then $L_{\text{run},M}$ can be accepted by an NPDA (resp., NPCM, NFA, NCM) M'.*

Proof. This is straightforward, since, given M, we can construct M' which, when given w, checks that it is in $L_{\text{run},M}$ by guessing an input x to M (symbol-by-symbol) and simulating M on this input and checking that x is an accepting run of M on x. □

There is a well-studied acceptor, called a checking stack automaton (CSA), introduced by Ginsburg *et al.* [13]. A CSA is a one-way NFA with a special stack. The special stack is like a pushdown stack: it can write but not pop, but it can enter the stack in a two-way read-only mode. However, once it enters the stack, it can no longer write, but it can continue using the stack in a read-only mode.

Theorem 4.10. *If M is a 2NFA, then $L_{\text{run},M}$ is accepted by a CSA M'.*

Proof. M when given an input x, checks that there is an input w accepted by M with accepting run x as follows: M' on ϵ moves, guesses a string w (including the left and right end markers) by pushing the symbols comprising w into the stack. Next, M' enters the the stack and moves its stack head to the bottom of the stack (which is has the left end marker). Then M' simulates the computation of M on w and verifies that the input x corresponds to an accepting run of M on w. □

5. Applications

As we have shown in Examples 2.1, 2.2, and 2.3, a program's execution can be roughly regarded as a run of a two-way automaton in the following sense. The program can be thought of, statically, a finite sequence of symbols (each symbol corresponds to a statement). This sequence is treated as a two-way input. When the program executes, the PC register stores the current line number that is being executed. Hence, the PC also stores the head position when the program is treated as a two-way read head working on the two-way input.

In Example 2.1, we assume that (unsigned) integer variables are in a finite range (e.g., encoded in 32-bit memory words). In fact, those variables are unbounded, whose sizes depend on the k-bit architecture that the program runs upon, where the k is theoretically unbounded. (Most laptops nowadays run on 64-bit architecture.) In this case, as we shown in previous sections, a two-way automaton that is used to simulate a program is likely of infinite states and hence the information rate of (sampled) runs/traces in the automaton is difficult, or even impossible, to compute.

In our previous paper [20], we have shown an approach and experiments to estimate information rate as well as information flow in a program modeled as a one-way automaton. Clearly, the approach can be easily adapted to two-way automaton modeling since the approach uses Lempel-Ziv compression (LZ in short) [36, 37] on a sequence (i.e., a sampled trace) while in a practical setting, the difference between one-way and two-way automata are simply the (slightly) different ways we obtain the trace. In the rest of this section, we plan to show a probably more interesting application of sampling a two-way automaton.

When a program is modeled as a two-way automaton, the information rate on how the head moves on the input should be an indicator on the program's dynamic complexity, which is explained as follows. In testing a program, not all lines of the code are equal. Some lines carry more information than the others, which is often caused by, for instance, the internal logical complexity inside these information-concentrated lines. This has already been observed in some research (e.g., "hot spot" in [31]). However, identifying these information-concentrated lines can not usually be carried out by statically inspecting the code. In fact, this is a dynamic property of the program, which, in our opinion, is closely related to information rate of the runs of the code. In this section, we introduce an approach in helping practitioners locate the lines in a program.

Given a program P and a set Δ of line numbers, we try to estimate the information complexity of the program's runs when the program executes through these lines. To do this, we propose a procedure with the following two steps:

1. execute the program P and obtain a long sequence α of line numbers. Delete those line numbers from α that do not appear in Δ and then obtain a sequence β, which is called a sampled trace; [2]
2. compress this sampled trace β with Lempel-Ziv algorithm [36, 37], which is known to be asymptotically optimal and is a most widely used string compression algorithm. We use the compression ratio (in inverse proportion) to estimate information rate concentrated on the lines in Δ.

[2]This step, when interpreted in terms of a two-way automaton M that models the program (see Example 2.1), is to sample M at positions $i \in \Delta$ of the input word, and obtain a trace where only the state variable PC is kept.

The intuition behind this procedure deserves more explanations. In a program, the information concentrated on the Δ is known to be inversely related to the compression ratio of the sampled trace [20]. It is also known that the compressibility of a trace is closely related to the predictability in the trace [11, 25]. Hence, a high information concentration in the set Δ indicates that those lines in Δ would cause more un-predictable behaviors in the run, and therefore in practice, need be tested more thoroughly and intensively.

In the rest of this section, we conduct five experiments to preliminarily confirm our intuition. The experiments are performed on five sorting algorithms over an array of 10,000 randomly generated integers. Each experiment contains two sets of Δ_{tricky} and Δ_{plain}. The rationale behind the choices of the sets is as follows. A program's run, when passing through the lines in Δ_{tricky} (resp. Δ_{plain}), takes more complex (resp. simpler) logical judgments. More precisely, the sets are selected in each of the five algorithms as follows.

1. **Quick sort.** Quick sort is a recursive algorithm and the key procedure is the partition. In the set of Δ_{tricky}, we select each recursive call statement and the pivot location statement. On the other hand, in the set Δ_{plain}, only the recursive call statements are included. The detail of position setting, as an example, is illustrated by Fig. 2.
2. **Selection sort.** Selection sort is a procedure that iteratively locates the minimum element in the unsorted part of the array and then swaps it to the first entry of the unsorted part. The set Δ_{tricky} contains each line where the minimum in the unsorted part is located, which is in the conditional within a two-fold loop, whilst the set Δ_{plain} contains only statements outside the conditional.
3. **Heap sort.** Heap sort is also a two-fold loop. The inner loop is used to swap the maximum element in a heap to the top of the heap. The swap statement are contained in Δ_{tricky}. On the contrary, the set Δ_{plain} only contains two assignments in the two loops respectively.
4. **Merge sort.** Merge sort is a recursive algorithm and the key procedure is merging two sorted arrays into a sorted one. The difference between the two sets Δ_{tricky} and Δ_{plain} is that Δ_{tricky} contains the statements that move elements in the sorted array to the temporary array, whilst Δ_{plain} only contains statements that move elements in the temporary array back to the sorted array.
5. **Bubble sort.** Bubble sort is a two-fold loop over comparisons between

```
 void quick_sort (int *a, int n) {
1     if (n < 2)
2         return;
3     int p = a[n / 2];
4     int *l = a;
5     int *r = a + n - 1;
6     while (l <= r) {
7        if (*l < p) {
8            l++;
9            continue;
        }
10        if (*r > p) {
11            r--;
12            continue;
        }
13        int t = *l;
14        *l++ = *r;
15        *r-- = t;
       }
16     quick_sort(a, r - a + 1);
17     quick_sort(l, a + n - l);
    }
```

Figure 2: The line numbers contained in Δ_{tricky} for quick sort are 3, 8, 11 and 13, while in Δ_{plain}, the line numbers are 3 and 13.

two neighboring entries. Δ_{tricky} contains statements where there is no exchange of two neighboring entries, as well as the loop statements. On the other hand, Δ_{plain} only contains the loop statements.

The experiment results are presented in Table 1, where the second (respectively, third) column contains information rate concentrated on Δ_{tricky} (respectively, Δ_{plain}). The rates are estimated with the inverse proportion of the respective compression ratio. From the table and in each row, one can see the rate in the second column is consistently larger than one in the third column, which confirms our aforementioned rationale behind the choices of the two sets. These preliminary experiments suggest an effective approach in locating statements in a program that have high information concentration,

which will be useful in software engineering.

Name	Rate on Δ_{tricky}	Rate on Δ_{plain}
Quick	0.095	0.028
Selection	0.061	0.0008
Heap	0.058	0.001
Merge	0.04	0.014
Bubble	0.0014	0.0008

Table 1: The information rates concentrated on Δ_{tricky} and on Δ_{plain}.

6. Conclusions

In this paper, we have studied position sampling in a two-way nondeterministic finite automaton (2NFA) to measure the information flow between state variables, based on the information-theoretic sampling technique proposed in [20]. We have proved that for a 2NFA, the language generated by position sampling is regular. We have also showed that for a 2NFA, we can effectively find a vector of sampling positions that maximizes information flow in a run of the 2NFA. Finally, we have summarized some language properties of sampled runs of 2NFAs augmented with restricted unbounded storage. We have conducted preliminary experiments to identify information concentration in a subset of statements in a program.

Acknowledgment. The research of Oscar H. Ibarra was supported by the United States NSF Grant CCF-1117708. The research of Qin Li was supported by: National NSF of China under Grant No. 61170070, 90818022, 61321491; National Key Technology R&D Program of China under Grant No. 2012BAK26B01; National High Technology Research and Development Program of China (863 Program) under Grant No. 2011AA1A202; the Major Program of the Natural Science Foundation of the Anhui Higher Education Institutions of China (Grant No. KJ2014ZD05); Natural Science Key Research Foundation of Education Bureau of Anhui Province, China (No. KJ2012Z022). We also thank Giovanni Pighizzini and Kai Salomaa for helpful comments concerning Lemma 4.8. We would like to thank Cewei Cui, Eric Wang and the reviewers for comments that improved the presentation of our results.

References

[1] R. Alur, D. L. Dill, A theory of timed automata, Theoretical Computer Science 126 (2) (1994) 183 – 235.

[2] B. Baker, R. Book, Reversal-bounded multi-pushdown machines, J. Comput. Syst. Sci 8 (1974) 315-332.

[3] S. Cavadini, Secure slices of insecure programs, in: Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security, ASIACCS '08, ACM, New York, NY, USA, 2008, pp. 112–122.

[4] N. Chomsky, G. A. Miller, Finite state languages, Information and Control 1 (1958) 91–112.

[5] D. Clark, S. Hunt, P. Malacaria, A static analysis for quantifying information flow in a simple imperative language, Journal of Computer Security 15 (2007) 321–371.

[6] C. Cui, Z. Dang, T. Fischer, O. Ibarra, Execution information rate for some classes of automata, in: A.-H. Dediu, C. Martín-Vide, B. Truthe (Eds.), Language and Automata Theory and Applications, Vol. 7810 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 226–237.

[7] C. Cui, Z. Dang, T. R. Fischer, O. H. Ibarra, Similarity in languages and programs, Theor. Comput. Sci. 498 (2013) 58–75.

[8] C. Cui, Z. Dang, T. Fischer, Bit rate of programs, submitted.

[9] Z. Dang, O. Ibarra, T. Bultan, R. Kemmerer, J. Su, Binary reachability analysis of discrete pushdown timed automata, in: E. Emerson, A. Sistla (Eds.), Computer Aided Verification, Vol. 1855 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2000, pp. 69–84.

[10] Z. Dang, Pushdown timed automata: a binary reachability characterization and safety verification, Theoretical Computer Science 302 (13) (2003) 93 – 121.

[11] M. Feder, N. Merhav, M. Gutman, Universal prediction of individual sequences, Information Theory, IEEE Transactions on 38 (4) (1992) 1258–1270.

[12] I. Gray, J.W., Toward a mathematical foundation for information flow security, in: Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on, 1991, pp. 21–34.

[13] S. Ginsburg, S. A. Greibach, M. A. Harrison, One-way stack automata, Journal of the ACM (JACM) 14 (2) (1967) 389–418.

[14] J. A. Goguen, J. Meseguer, Security Policies and Security Models, in: IEEE Symposium on Security and Privacy, 1982, pp. 11–20.

[15] E. Gurari, O. H. Ibarra, The complexity of decision problems for finite-turn multicounter machines, J. Comput. Syst. Sci 22 (2) (1981) 220-229.

[16] O. H. Ibarra, Reversal-bounded multicounter machines and their decision problems, J. ACM 25 (1) (1978) 116–133.

[17] O. H. Ibarra, T. Jiang, N. Q. Tran, H. Wang, New Decidability Results Concerning Two-Way Counter Machines, Siam Journal on Computing 24 (1995) 123–137.

[18] O. H. Ibarra, Z. Dang, O. Egecioglu, G. Saxena, Characterizations of catalytic membrane computing systems, in: B. Rovan, P. Vojt (Eds.), Mathematical Foundations of Computer Science 2003, Vol. 2747 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2003, pp. 480–489.

[19] W. Kuich, On the entropy of context-free languages, Information and Control 16 (2) (1970) 173–20.

[20] Q. Li, Z. Dang, Sampling automata and programs, Theoretical Computer Science 577 (2015) 125–140.

[21] S. McCamant, M. D. Ernst, Quantitative information flow as network flow capacity, SIGPLAN Not. 43 (6) (2008) 193–205.

[22] J. McLean, Security models and information flow, in: Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on, 1990, pp. 180–187.

[23] A. McIver, C. Morgan, A probabilistic approach to information hiding, in: A. McIver, C. Morgan (Eds.), Programming Methodology, Monographs in Computer Science, Springer New York, 2003, pp. 441–460.

[24] J. K. Millen, Covert Channel Capacity, in: IEEE Symposium on Security and Privacy, 1987, pp. 60–66.

[25] J. Rissanen, Universal coding, information, prediction, and estimation, Information Theory, IEEE Transactions on 30 (4) (1984) 629–636.

[26] C. E. Shannon, W. Weaver, The Mathematical Theory of Communication, University of Illinois Press, 1949.

[27] G. Smith, D. Volpano, Secure information flow in a multi-threaded imperative language, in: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '98, ACM, New York, NY, USA, 1998, pp. 355–364.

[28] G. Smith, On the foundations of quantitative information flow, in: L. Alfaro (Ed.), Foundations of Software Science and Computational Structures, Vol. 5504 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 288–302.

[29] G. Paun, Membrane Computing, an Introduction, Springer Berlin Heidelberg, 2000.

[30] T. Terauchi, A type system for observational determinism, in: Computer Security Foundations Symposium, 2008. CSF '08. IEEE 21st, 2008, pp. 287–300.

[31] E. Wang, C. Cui, Z. Dang, T. R. Fischer, L. Yang, Zero-knowledge black-box testing: where are the faults? International Journal of Foundations of Computer Science 25 (2) (2014) 196-218.

[32] M. Weiser, Program slicing, Software Engineering, IEEE Transactions on SE-10 (4) (1984) 352–357.

[33] G. Xie, Z. Dang, O. Ibarra, A solvable class of quadratic diophantine equations with applications to verification of infinite-state systems, in: J. Baeten, J. Lenstra, J. Parrow, G. Woeginger (Eds.), Automata, Languages and Programming, Vol. 2719 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2003, pp. 668–680.

[34] B. Xu, J. Qian, X. Zhang, Z. Wu, L. Chen, A brief survey of program slicing, SIGSOFT Softw. Eng. Notes 30 (2) (2005) 1–36.

[35] L. Yang, C. Cui, Z. Dang, T. R. Fischer, An information-theoretic complexity metric for labeled graphs, submitted.

[36] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, Information Theory, IEEE Transactions on 23 (3) (1977) 337–343.

[37] J. Ziv, A. Lempel, Compression of individual sequences via variable-rate coding, Information Theory, IEEE Transactions on 24 (5) (1978) 530–536.