ELSEVIER

Theoretical Computer Science 143 (1995) 23-49

Theoretical Computer Science

Nondeterministic, probabilistic and alternating computations on cellular array models*

Kamala Krithivasan[a], Meena Mahajan[b,*,1]

[a] Department of Computer Science and Engineering, Indian Institute of Technology, Madras 600036, India
[b] The Institute of Mathematical Sciences, CIT Campus, Madras 600 113, India

Received April 1993; revised May 1994
Communicated by M. Nivat

Abstract

A new mechanism for introducing nondeterminism on the cellular automaton model is introduced. It is shown that this form of nondeterminism corresponds to the traditional notion in the unbounded-time case, but there appear to be differences when real-time or linear-time cellular automata are considered. The notion is then generalised to include probabilistic and alternating computations. Restricted nondeterminism classes are also defined and studied, in an attempt to refine the power of nondeterminism.

1. Introduction

Cellular automata (CA) are a simple model for parallel recognition of languages, and have been the object of study for several years [3,6,12,15,20,21]. A CA consists of an array of identical finite-state machines (FSM), one for each letter of the input. The FSMs are called cells. Let $c(i,t)$ denote the state of the ith cell at time t. The CA is initialised by setting $c(i,0)$ to a_i, where the input is $a_1a_2 \ldots a_n$. Subsequently the operation of the CA is autonomous. At discrete time steps $t = 1,2,\ldots$, all cells synchronously update their states, with $c(i,t+1)$ expressed as a function of $c(i-1,t)$, $c(i,t)$ and $c(i+1,t)$. (If a neighbouring cell is missing, i.e. at the boundaries of the array, a special state # is taken to be the missing argument). The leftmost cell of the array is the accepting cell, and the input is accepted if this cell ever enters an accepting state.

* Corresponding author. Email: meena@imsc.ernet.in.

* The financial support of the Department of Science and Technology, Government of India, is gratefully acknowledged.

[1] This work was done when the second author was at the Department of Computer Science and Engineering, Indian Institute of Technology, Madras 600 036, India.

SSDI 0304-3975(94)60096-2

Note that for nontrival language recognition acceptance requires at least n time steps on an input of length n. See Fig. 1.

Formally, a CA is defined as follows.

Definition 1.1. A cellular automaton is a 4-tuple $C = (Q, \#, \delta, A)$, where

(i) Q is a finite set of states,

(ii) $\# \in Q$ is the boundary state,

(iii) $\delta: Q \times Q \times Q \to Q$ is the local transition function satisfying

$$\delta(a,b,c) = \# \text{ if and only if } b = \#,$$

(iv) $A \subseteq Q$ is the set of accepting states.

Throughout this paper we consider only CA which are space-bounded; on an input of length n the CA has exactly n cells. We denote by $L(C)$ the language accepted by the CA C.

The operation of the CA is frequently represented using a time–space diagram. This is an array where the topmost row has the input configuration, and successive configurations appear in successive rows beneath it. Thus the ith row gives the configuration of the CA after i time steps, and the jth column gives the sequence of states entered by the jth cell of the CA. Such diagrams give a visual representation of the CA computation and make it more easily comprehensible. Signals travelling across the array of FSMs are shown in such diagrams by lines of varying slopes, depending on the speed at which the signal is travelling.

A CA C is said to operate in $T(n)$ time if for each n, for each string w of length n, if w is accepted then it is accepted within $T(n)$ time steps. In other words, if $c(1,0)c(2,0)\ldots c(n,0) = w$ and $w \in L(C)$, then $\exists t \leqslant T(n)$ such that $c(1,t) \in A$. Here $T(n)$ could be any function $T: \mathbb{N}^+ \to \mathbb{N}^+$. Of special interest are the cases when $T(n) = n$, giving "real-time" CA (rCA), and $T(n) = cn$ for some constant c, giving "linear-time" CA (lCA).

A restricted version of a CA is the one-way CA (OCA), where $c(i, t+1)$ is a function of $c(i-1, t)$ and $c(i, t)$ only, i.e. a cell's right neighbour does not affect it. Thus δ is now a function from $Q \times Q$ to Q. For OCA, the rightmost cell is the accepting cell, where acceptance is as defined earlier.

The definitions of CA and OCA can be generalised to the nondeterministic case. Now δ will map $Q \times Q \times Q$ (or $Q \times Q$, for OCA) to subsets of Q, and the input will be accepted if for some computation of the CA satisfying δ, the accepting cell enters

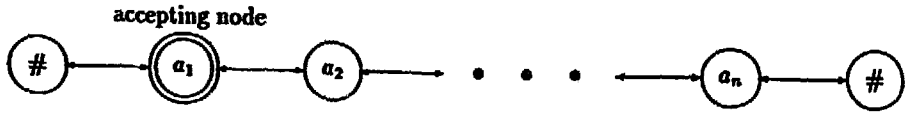


Fig. 1. A cellular automaton.

a state from A. Nondeterministic CA (NCA and NOCA) have also been studied in some detail in the past; some of the results can be found in [13, 21]. For a nondeterministic CA, for the same input there can be several time–space diagrams, corresponding to different nondeterministic choices.

The results currently known about the language classes defined by these models can be summarised as follows.

(1) A $T(n)$-time CA can be simulated by a DSPACE(n) Turing machine in $O(nT(n))$ time: This simulation is done in a straightforward fashion – for one step of the CA, where n cells update their states in parallel, the Turing machine sweeps down the array and updates each tape cell sequentially.

(2) NOCA = NCA = NSPACE (n), the class of context-sensitive languages: The NCA to NSPACE(n) simulation is as described above, with the Turing machine being nondeterministic to simulate the nondeterministic moves of the CA. An NCA can easily simulate an NSPACE (n) machine, by letting most cells idle most of the time. Only the cell representing the tape square where the tape head is positioned, and its neighbouring cells, change state at each step. For an NOCA to simulate an NCA, each cell has to guess its right neighbour's state, and special signals have to travel across the array verifying that the guesses were correct. The technique is described in detail in [9].

(3) NSPACE $(\sqrt{n}) \subseteq$ OCA $\subseteq$ CA = DSPACE(n) $\subseteq$ NSPACE(n): The first containment is shown in [5, 11] by first showing that any language accepted by a linear-time alternating Turing machine can also be accepted by a restricted deterministic Turing machine equivalent to an OCA. The remaining containments are obvious.

(4) For CA and OCA, $T(n) + c$ time can be speeded up to $T(n)$ time: This has been independently proved in [3, 6, 14, 15], as also the following result.

(5) For CA and OCA, linear time can be speeded up to cn time for any $c > 1$.

(6) rOCA $\subset$ rCA = lOCA $\subseteq$ lCA $\subseteq$ OCA $\subseteq$ CA: The proper containment was shown in [6] by using a pumping lemma kind of argument for rOCA. The rCA = lOCA equality has been shown in [3] by a direct construction, and the lCA $\subseteq$ OCA containment was shown in [5, 11] using the restricted Turing machine characterisations of OCAs and another parallel recognition device, the one-way iterative array.

It is also known that rNOCA contains an NP-complete problem [13], lCA is contained in P (follows from (1) above), and OCA contains a PSPACE-complete problem.

The following problems posed variously in [3, 12, 21] are still open.

(a) Are linear-time CA more powerful than real-time CA?
(b) Are nonlinear-time CA more powerful than linear-time CA?
(c) Are nonlinear-time CA more powerful than real-time CA?
(d) Are CA more powerful than OCA? i.e. are there languages accepted by CA which are provably not accepted by OCA?
(e) Are real-time CA closed under reversal? In [12] it has been shown that this is the case if and only if the answer to (a) is No.

In this paper, we consider a new model of nondeterminism based on the structure of time-varying automata, and, imposing this model upon CA and OCA, investigate the

power of the resulting classes. The notion of a time-varying CA (TVCA) was introduced in [18], and in [17] TVCA were interpreted as relativised CA i.e. CA which compute with some help from an oracle. In this paper we further generalise this notion, and interpret the computation of a TVCA as a nondeterministic CA, a probabilistic CA, and an alternating CA. The description and definitions of TVCA are presented in Section 2. In Section 3, we consider nondeterministic TVCA and compare them with the traditionally defined NCA. In Section 4, probabilistic TVCA are considered, especially vis-à-vis NTVCA. Section 5 considers alternating computations on TVCA. In Section 6, some restricted forms of nondeterministic and probabilistic TVCA are studied, with the intention of trying to identify how much nondeterminism is required, if at all, to enhance the power of a particular class. Section 7 considers some closure properties of these TVCA classes.

2. Preliminaries and Definitions

In a TVCA, the transition function to be applied to each cell depends not only on the states of cells in the neighbourhood but also on the number of time steps elapsed since the CA operation began. The dependence on time is expressed in the following way: a set of transition functions $\delta_1, \delta_2, \ldots, \delta_k$ is associated with the CA, and δ, the effective transition function of the CA, agrees with one of $\delta_1, \delta_2, \ldots, \delta_k$ depending on the time. In other words,

$$\delta(a,b,c,i) = \delta_{f(i)}(a,b,c),$$

where $a,b,c \in Q$, $i \in \mathbb{N}$, and $\delta_{f(i)}$ is the transition function used at time $t = i$. The manner in which f is chosen thus crucially affects the overall computation. Such a TVCA with k transition functions is called a k-TVCA.

In the above description, the function $f: \mathbb{N}^+ \to \{1, \ldots, k\}$ can be viewed as an oracle which guides the computation of the TVCA; for more on such "relativised" CA see [17, 18]. As pointed out in [18], any language over a unary alphabet can be accepted by a 2-TVCA in real time, even if the language is undecidable. Thus, for meaningful results, we are interested only in situations where the function f is computable within some specific resource bound. (CA-based resource bounds on f are considered in [17].)

One important fact to note about TVCA is that speed-up does not necessarily hold. Neither (4) nor (5) from Section 1 can be shown to trivially apply to TVCA. Since we are essentially interested in the dependence of running time on input length, we will still continue to ignore additive constants, and treat $(T(n) + c)$ time as equivalent to $T(n)$ time. However, for multiplicative constants, there is a trade-off. To be more precise, consider speeding up the operation of a k-TVCA by a factor of 2. Even assuming that an initial phase achieves the required packing of the input, to be able to simulate two steps of the TVCA in one step calls for the ability to simulate k^2 different combinations of the form $\delta_i \delta_j$. So the simulating TVCA will need k^2 different transition functions. Thus speed-up is achieved at the cost of the number of functions

required. Conversely, the number of functions can be reduced at the expense of slowing down the computation – a k-TVCA operating in $T(n)$ time can be simulated by a 2-TVCA operating in $(\log_2 k)\,T(n)$ time [18]. Since the slowing down is only by a constant factor, for (linear-time) TVCA it is sufficient to consider 2-TVCA. But for real-time computation, it appears that k is a crucial parameter; whether $k+1$ functions are better than k for real-time TVCA is an open problem posed in [18].

In [17], 2-TVCA have been interpreted as relativised CA. A tally language $L \subseteq 0^*$ is the oracle, and δ is now expressed as follows:

$$\delta(a,b,c,i) = \begin{cases} \delta_1(a,b,c) & \text{if } 0^i \in L, \\ \delta_2(a,b,c) & \text{otherwise.} \end{cases}$$

Note that for a 2-TVCA operating in time $T(n)$, there are $2^{T(n)}$ possible computation paths, and the structure of L determines which of these paths is chosen. In [17], we have examined how varying the complexity of the oracle L affects the computational power of the TVCA.

In this paper we relax the notion of a single computation path being checked for acceptance. First we define the characteristic bit strings of a language and of a TVCA computation path as follows.

Definition 2.1. The characteristic bit string of a language $L \subseteq \Sigma^*$ is a bit string $a_0a_1a_2\ldots$ where each $a_i \in \{0,1\}$, and for the standard enumeration of Σ^*, $a_i = 0$ if and only if the ith word of Σ^*, w_i, is in L.

Thus for a tally language L, $a_i = 0$ if and only $0^i \in L$.

For a 2-TVCA, on input w of length n, a $T(n)$-time computation path is a sequence $(w =)\, w_0, w_1, \ldots, w_{T(n)}$ where for each i, $|w_i| = |w_0|$, $w_i \in Q^*$ for $i > 0$, and for $i > 0$, w_i can be obtained from w_{i-1} by applying either δ_1 or δ_2. It is an accepting computation if $w_{T(n)}$ is an accepting configuration, i.e. the leftmost state is an accepting state.

Definition 2.2. The characteristic bit string of a $T(n)$-time computation path is a $T(n)$-length bit string $b_1b_2\ldots b_{T(n)}$ where $b_i = 0$ if w_i can be obtained from w_{i-1} by applying δ_1, and $b_i = 1$ otherwise.

Note that this definition assigns a unique bit string as the characteristic bit string for a given computation. However, more than one bit string may still determine the same computation. This could happen, for instance, if, from a particular configuration, both δ_1 and δ_2 lead to the same next configuration. The unique characteristic bit string of a given computation is the lexicographically smallest of the bit strings determining the computation.

Consider Fig. 2, a binary tree. The root node holds w_0. The left (right) child of a node holding c holds the configuration obtained by applying δ_1 (δ_2) to c. Such a binary tree, of height $T(n)$, gives all possible computations of a $T(n)$-time 2-TVCA on input w_0. A bit string of length $T(n)$ picks out a particular path in this tree. In

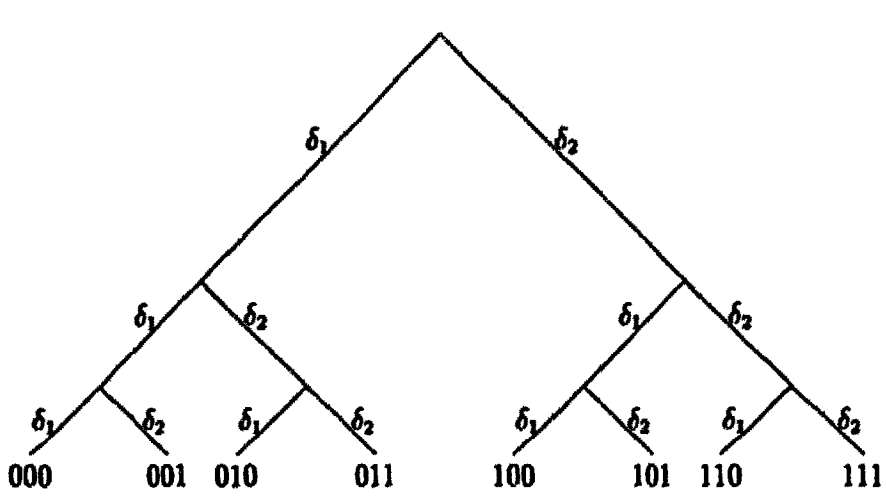


Fig. 2. Binary tree of possible computations and characteristic bit strings.

a relativised CA operation, the unique computation path whose characteristic bit string is a prefix of the characteristic bit string of the oracle is picked, and the input is accepted if and only if this computation path ends in an accepting configuration. Instead, we can check whether at all there exists an accepting computation, thus giving a nondeterministic interpretation to the TVCA. Or, we can check whether more than half of the computation paths are accepting computations, thus interpreting the TVCA operation as a probabilistic computation. Additionally, if the states of the TVCA are partitioned into universal, existential, accepting and rejecting states, then the NTVCA can be generalised to an alternating CA, an ACA. These types of TVCA are formalised and studied in the following sections. For probabilistic TVCA, when we count the number of accepting computations, we want to have a binary tree, of computations, which is pruned at a particular height. So we impose the condition that the TVCA's running time, $T(n)$, be CA-time-constructible, a notion defined below.

Definition 2.3. A function $T(n): \mathbb{N}^+ \to \mathbb{N}^+$ is said to be CA-time-constructible if there is a CA which, on any input of length n, puts its accepting cell into a special state after exactly $T(n)$ time steps. The function is said to be strongly CA-time-constructible if the CA puts every cell into a special state, for the first time, after exactly $T(n)$ steps. In other words, after $T(n)$ steps, all cells simultaneously "fire" for the first time.

(Strong CA-time-constructibility is a generalisation of the famous firing squad synchronisation problem, for which a tight lower bound of $2n-2$ is known [21]. This lower bound applies when only one end of the array can initiate synchronisation action. If both ends can do so, then synchronisation can be achieved in real time.)

3. Nondeterministic TVCA

Definition 3.1. A nondeterministic TVCA is a construct $C = (Q, \#, \delta_1, \delta_2, A)$ defined as a 2-TVCA. A string w is accepted by C in time $T(n)$ if $\exists \alpha \in \{0,1\}^{T(|w|)}$ such that the computation path of C beginning with w and with characteristic bit string α is an accepting computation.

Thus the crucial difference between NCA and NTVCA is that in NCA, each cell independently makes a nondeterministic choice about the next state, whereas in NTVCA, a global nondeterministic choice is made about whether to use transition function δ_1 or δ_2, and then all cells use this chosen transition function.

First we consider the unbounded-time classes of NTVCA.

Lemma 3.2. NTVCA $\subseteq$ NSPACE(n); *a* $T(n)$*-time* NTVCA *can be simulated by an* NSPACE(n) *machine in* $O(nT(n))$ *time.*

Proof. A $T(n)$-time NTVCA can be simulated by an NSPACE(n) machine which simulates one step of the NTVCA as follows. It first decides, nondeterministically, whether to use δ_1 or δ_2, and then moves down the entire array, deterministically updating the state of each cell accordingly. □

Lemma 3.3. NOCA $\subseteq$ NTVOCA; *an* NTVOCA *can simulate a* $T(n)$*-time* NOCA *in* $O(nT(n))$ *time.*

Proof. Let $C = (Q, \#, \delta, A)$ be an NOCA, where δ maps $Q \times Q$ to subset of Q. Let k be the size of the largest subset of Q in the range of δ. We will construct a $(k+1)$-NTVOCA C' accepting the same language as C. Then, as described in Section 2, an equivalent 2-function NTVOCA can be constructed.

Each cell of C can independently choose one of upto k options when making a transition according to δ. But in C', at a single time step, all cells must use the same option. So to simulate the n independent choices made by C in one step on an n length input, C', needs n steps, where at each step exactly one cell of C' makes a transition and all other cells merely maintain their state. Now the first k distinct transition functions of C' can implement the k options provided by δ. The leftmost cell of C' sends a pulse right at unit speed. As this pulse passes through a cell, that cell makes a state transition. When the pulse reaches the right end, all cells have updated their states and one step of C has been simulated. One row in the time–space unrolling of C appears as a diagonal in the time–space unrolling of C'. See Fig. 3.

The problem which now arises is that the leftmost cell does not know when to send out the next pulse. Pulses should be at least n time steps apart, but, in the absence of two-way communication, counting upto n is not possible. What happens if we allow arbitrary spacing of pulses? The leftmost cell sends a pulse whenever δ_{k+1} is used. If the pulses are more than n steps apart, then in between there will be some idle steps,

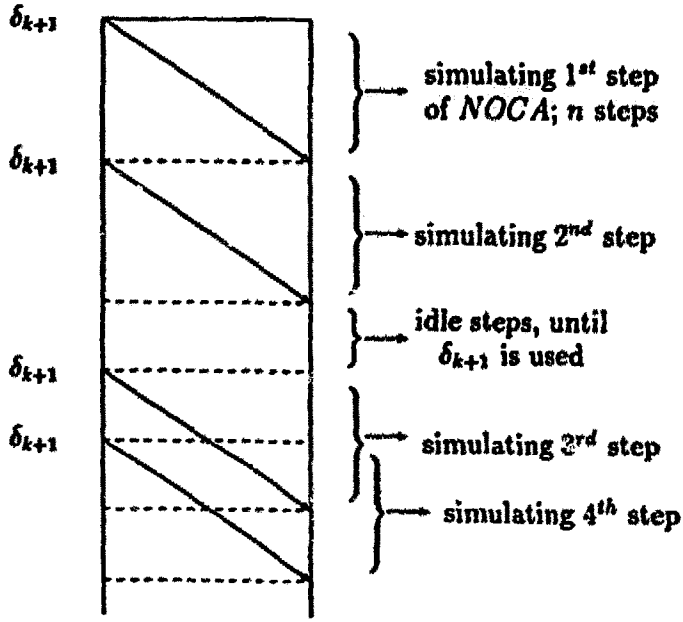


Fig. 3. An NTVOCA simulating an NOCA.

when C' does nothing. If the pulses overlap, then some cells have to make related choices. But these choices could have been made even if all cells were acting independently. The crucial observation is that with arbitrary spacing allowed, properly spaced pulses also occur along some computation paths of C', guaranteeing the checking of all possible paths of C. The other paths, with overlapping pulses, are already simulated on some of these paths and are thus redundant, but not wrong.

For a $T(n)$-time computation path of C, there will be a computation path of C' where the pulses are exactly n steps apart; this computation path will be of length $(n+1)T(n)$. So if C has an accepting path of length $T(n)$, then C' certainly has at least one accepting path of length $(n+1)T(n)$. Further, if C has no accepting path, neither does C'.

With this construction, $T(n)$ steps of the NOCA are simulated in $O(nT(n))$ steps by the NTVOCA. This NTVOCA can be converted to one having only two transition functions, with a slowing down only by a constant factor. This is the required NTVOCA. □

From these two lemmas and (2) in Section 1, we can now conclude the following theorem.

Theorem 3.4. NTVOCA = NTVOCA = NSPACE(n).

The following lemma further strengthens the statement NTVCA ⊆ NCA; it claims that a real-time simulation is possible.

Lemma 3.5. *An* NTVCA *can be simulated by an NCA with no loss of time. If the* NTVCA *uses only one-way communication, so does the simulating* NCA.

Proof. In an NTVCA, all cells must use the same transition function, at any given time instant. This condition can be enforced in an NCA as follows: Each cell of the NCA nondeterministically uses δ_1 or δ_2 at any time instant. Additionally, each cell also records, in its state, which transition function was used. From time step $t = 2$ onwards, each cell also checks that the cells in its neighbourhood used the same transition function as itself at the previous step. If this is not the case, a reject signal is generated and sent to the accepting cell. Thus if the NCA accepts its input, it must be via a computation where all cells had used the same transition function at each time instant; i.e. it must be via a computation corresponding to a computation path of the NTVCA. □

We now look at the time-bounded NTVCA classes. The next result highlights the difficulty of determining membership for real-time NTVOCA languages. It was known that the membership problem for real-time NOCA is NP-complete [13]. We show that this continues to hold even if we consider NTVOCA rather than NOCA.

Theorem 3.6. *The class of real-time* NTVOCA *languages contains a language which is* NP*-complete.*

Proof. Consider the language of satisfiable Boolean formulas in 3-clause conjunctive normal form 3-CNFSAT. Let the formulas be coded as follows:

$$*v_1 ¢ v_2 ¢ \ldots ¢ v_m \$ F_1 \wedge F_2 \wedge \ldots \wedge F_k *,$$

where

- $*$ is a special end-marker,
- $v_i \in \{0,1\}^*$,
- $|v_i| = |v_j|$ for each i, j,
- $v_i \neq v_j$ for $i \neq j$, and
- each F_i is of the form $w \vee x \vee y$, where w, x and y are of the form $0v_t^R$ or $1v_t^R$ for some t.

Thus the input has a list of variables, coded as equal length bit strings separated by ¢s, followed by a \$, followed by a set of clauses separated by $\wedge$'s, where each clause has three terms separated by $\vee$ s, and each term is either $0v^R$, representing the variable v, or $1v^R$, representing the negation of the variable v, for some variable v. Here v^R denotes the string obtained by reversing the characters in v. We store the variable descriptions in different orders in the initial part and in the formula part to facilitate cross-checking.

This language is well known to be NP-complete [1, 10]. Consider the following NTVOCA accepting it.

At the first time step, a signal Assign_and_Evaluate starts moving right from the \$ cell. This signal does two things. Firstly, as it passes over each cell holding a $\vee$ or a $\wedge$, that cell nondeterministically chooses a value 0 or 1. This value is considered to be the value of the variable preceding it in the formula. Secondly, it collects these chosen values as it travels right, and evaluates the formula; at each $\vee$ or $\wedge$ and at the last $*$, the partial value of the formula to its left is stored.

If the value that reaches the last $*$ is 0, then the input is not accepted. If it is 1, then we must check that different occurrences of the same variable in the formula are assigned values at the nondeterministic steps in a consistent way. To do this check, the entire input stream moves right at unit speed, while each cell also retains a copy of its original input symbol. Now the variables in the left part (i.e. before the \$) encounter those in the right part after the Assign_and_Evaluate signal has gone over them. Consider a situation when variable v_i is moving over a substring $\vee 0 w \vee$. The first $\vee$ indicates that comparison should begin after the next bit. The first position of the moving stream records that the variable is not negated. So the Boolean value in the moving stream (this value is set when the first instance of v_i^R is found) should match that in the last $\vee$ if $w^R = v_i$. But checking whether a substring is of the form $w \vee 0 w^R$ is in rOCA by standard techniques; see [6–8]. So the moving stream can check if the same variable is represented under it, and, if so, check that the value is consistently assigned. The $\vee$ after this variable allows the moving stream to "reset" itself, to be ready to check the next variable.

If any consistency check fails, it is recorded in the cell where the failure is discovered.

In this fashion, when the left end-marker $*$ reaches the right one, it can find whether any inconsistency has been recorded. If this is not the case, then the input is accepted. Clearly, this process takes exactly as much time as the length of the input. See Fig. 4. □

We next look at the relationship lOCA = rCA (refer to (6) in Section 1) in the context of nondeterminism. For the traditionally defined nondeterministic classes, the equality rCA = lOCA continues to hold, since the speed-up of lOCA to $2n$ time and the simulation of an rCA by a $2n$-time lOCA and vice versa are not affected by nondeterminism in the transition function. This is not the case for NTVCA. In fact, an equivalent result does not seem to hold, but we have a restricted version.

Theorem 3.7. *The class of* NTVOCA C *running in time* $2n$ *and satisfying the condition*

$$\forall x \in \Sigma^*,\quad x \in L(C) \Leftrightarrow \exists \text{ a computation path accepting } x, \text{ whose characteristic bit string } a_1 a_2 \ldots a_{2|x|} \text{ has } a_{2i-1} = a_{2i} \text{ for } i = 1 \text{ to } |x|$$

is exactly equivalent to the class of real-time NTVCA.

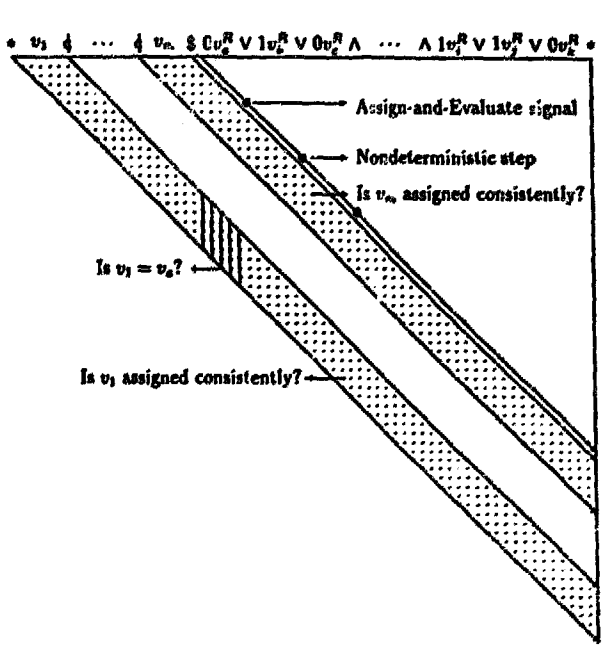


Fig. 4. An rNTVOCA accepting SAT.

Proof. Though $2n$ steps are allowed for the computation of the NTVOCA, effectively only n steps are allowed for the nondeterministic choice. Under such conditions, the equivalence of $2n$-time NTVOCA and real-time NTVCA can be shown in a manner identical to the proof that rCA = lOCA [3]. □

As for the containment ICA ⊆ OCA from (6), Section 1, we show that even linear-time NTVCA are no more powerful than deterministic OCA. We do not know whether a similar result holds for INCA. We do know, however, that the containment holds if both classes are made nondeterministic using the traditional notion, i.e. that INCA are contained in NOCA, because NOCA and NCA have the same power.

Theorem 3.8. *Linear-time* NTVCA ⊆ OCA.

Proof. We resort to the sequential machine characterisation of OCA to prove this result. It has been shown [5, 11, 15] that OCA are equivalent to a restricted form of an on-line single-tape Turing machine, called a sweeping automaton (SA). An SA consists of a semi-infinite worktape (bounded at the left by a special boundary marker ¢) and a finite-state control with an input terminal at which it receives the serial input $a_1a_2 \ldots a_n$. The symbol \$ is used as end-marker. The SA operates in left-to-right sweeps as follows.

Initially, all cells of the worktape to the right of ¢ contain the blank symbol λ. A sweep begins with the read-write-head (RWH) scanning ¢ and the machine in a distinguished state q_0. In the ith sweep, the machine reads a_i and moves right of ¢ into a non-q_0 state. It continues moving right, rewriting non-λ symbols by non-λ symbols and changing states except into q_0. When the RWH reads a λ, it rewrites it by a non-λ symbol and resets to the leftmost cell in state q_0 to begin the next sweep. When \$ is first read, the machine completes the $(n+1)$th sweep, writes a \$ on the $(n+1)$th tape cell, and resets to ¢ in state q_0. Subsequent sweeps are performed between ¢ and \$ without expanding the workspace. \$ is assumed to be always available for reading after the input is exhausted. The input is accepted if the machine eventually enters an accepting state at the end of a sweep.

Several techniques for programming an SA have been described in [5]. We use some of these techniques in the following; for a full description of how the techniques are implemented on an SA, the reader is referred to [5].

Given a linear-time NTVCA, we will construct a sweeping automaton SA accepting exactly the same language.

Let C be an NTVCA running in cn time. A valid computation path thus has a cn length characteristic bit string. We design the SA to generate all cn length strings in lexicographic order, and, for each string, to trace out the corresponding computation path. The SA will accept its input if it ever finds an accepting computation in this process.

As the SA reads its input, it shifts and packs symbols on the tape. When the entire input has been read, the worktape will be partitioned into three areas as shown in Fig. 5.

The first area is a counter of length cn, and holds the string α reversed, i.e. with its least significant bit first. The second area is also of length cn, for holding the bit string β currently being tested. Initially both α and β are set to 0^{cn}. Further, the first 0 of β has a special marker under it. The marker indicates which step of the bit string currently being tested is to be simulated next; it moves left to right. The purpose of the counter α is to indicate when the next string β may be considered; any counter which resets more than cn steps apart will do. Such a counter is required because the SA cannot carry information from right to left; based on the β marker alone, it cannot know when to generate the next β.

The third area is of length cn and has two tracks. The first track has a permanent copy of the input x in its leftmost n cells, and is blank elsewhere. The second track is initially a copy of the first track, and is to be used for tracing out the computation

Fig. 5. Worktape of an SA accepting an INTVCA language.

corresponding to the string in the second area. This requires cn space and not n space because the SA can only move from left to right, while the NTVCA has two-way communication. So in simulating each step of the NTVCA, the SA shifts the configuration one cell right.

While reading \$, i.e. after all the input has been read, α is incremented in each sweep. This can be done in a left to right scan, because α is stored in reversed order. Simultaneously, the marker moves right, one cell per sweep, under the string β. If the marker is on a 0 (1, respectively), then the NTVCA configuration y, which is stored on the second track of the third area, is updated as per δ_1 (δ_2, respectively). When the marker reaches the end of the second area, a full computation path has been traced. The marker is now erased, and the third area is reset to its initial status. It remains unchanged in subsequent sweeps until α overflows. When this happens (every 2^{cn} sweeps), the next bit string is generated in the second area (β is incremented; again, in a left-to-right scan). The marker is placed again on its leftmost bit, and the tracing out of the corresponding computation path begins in the third area. Thus all computation paths are traced, and an accepting computation, if any, can be found by the SA. □

4. Probabilistic TVCA

In this section we look at some of the classes obtained by viewing the operation of a TVCA as a probabilistic computation, i.e. a computation which is deemed to be accepting if more than half of the subcomputations are accepting. For such probabilistic TVCA (PTVCA), we impose the condition that $T(n)$ be CA-time-constructible. PTVCA are formally defined as follows.

Definition 4.1. Let $T(n)$ be a CA-time-constructible function. A $T(n)$-time probabilistic TVCA (PTVCA) is a construct $C = (Q, \#, \delta_1, \delta_2, A)$ defined as a 2-TVCA. Acceptance is defined as follows: A string w is accepted by C if more than half of the $T(|w|)$-time computations of C on w are accepting computations.

Henceforth, for PTVCA, when we talk of a $T(n)$-time computation we implicitly assume that $T(n)$ is CA-time-constructible.

The following result is easily shown.

Theorem 4.2. *$T(n)$-time* NTVCA $\subseteq$ *$T(n)$-time* PTVCA.

Proof. Let C be a $T(n)$-time NTVCA. On any input w of length n, it has $2^{T(n)}$ computation paths. If even one of these is an accepting path, then C accepts w. This condition can be easily incorporated into a probabilistic computation of C' as follows: at the first step, δ_1 takes C' into a special dummy configuration from which every ensuing computation is accepting. δ_2 takes C' into the start configuration of C. If this

has an accepting computation, then the PTVCA C' so defined has strictly more than half accepting computations and so accepts its input.

The problem with this method is that it requires $T(n) + 1$ time steps because of the initial dummy step. To recover this time step, the simulation of the NTVCA has to be speeded up by one step. Let C_1 and C_2 be the configurations resulting from applying δ_1 and δ_2, respectively, to the start configuration of C. In the PTVCA, after the first step when δ_2 is applied, C' goes into a configuration with two channels, holding C_1 and C_2. At subsequent steps, each channel is updated according to the transition function in use at that step. Thus each computation of C' with characteristic bit string 1α holds the results of two computations of C – namely, the computations with characteristic bit strings 1α and 0α – in its two channels. C' is programmed to accept its input if either of the two channels holds an accepting configuration. Clearly, C' accepts the same language as C probabilistically, and does so in $T(n)$ time. □

Corollary 4.3.

Real-time NTVCA $\subseteq$ *real-time* PTVCA.

Linear-time NTVCA $\subseteq$ *linear-time* PTVCA.

By a construction similar to that in the proof of Theorem 3.8, we can also show that linear-time PTVCA are no more powerful than deterministic OCA.

Theorem 4.4. *Linear-time* PTVCA $\subseteq$ OCA.

Proof. Given any linear-time PTVCA, we will construct an SA (sweeping automaton) accepting the same language as the PTVCA. This will prove the theorem. Most of the details of the construction of the SA are as in the proof of Theorem 3.8; here we will only describe the additional details.

While simulating the NTVCA, the SA traced out the different computation paths of the NTVCA in lexicographic order, and accepted the input if any accepting computation path was found. For a PTVCA simulation, the SA must check all computation paths, and count how many of them are accepting computations. For this, a fourth area is created on the worktape, beyond the three areas described earlier. This area has cn cells, and is used as a counter γ initialised to zero. γ is incremented whenever an accepting computation is found. When all computation paths have been checked, the SA checks whether γ contains a number greater than 2^{cn-1}. If this is the case, the SA moves right in an accepting state; otherwise it moves right in a rejecting state. Thus the probabilistic acceptance condition is checked. □

5. Alternating computations on CA

Further generalising the concept of nondeterministic and probabilistic TVCA, we now introduce alternation in the CA model of computation. This follows the notion of

alternation in Turing machines, introduced in [4]. While nondeterminism allows a computation to proceed locally along any of two paths, and is said to be accepting if either of these two paths leads to acceptance, alternation also allows the computation to accept if and only if *both* resulting paths end in acceptance. Thus an alternating computation could have deterministic moves, nondeterministic moves, and universal moves. A proof is now not just a single path in the computation tree but a subtree which keeps track of all universal moves. The results in [4] show that this generalisation moves up the class of accepted languages by one step, in the hierarchy logspace, polynomial time, polynomial space, ... Thus ALOGSPACE coincides with PTIME, and APTIME coincides with PSPACE. We do not expect such a dramatic shift when alternation is introduced in CA, because the CA are already space-bounded. We investigate the precise extent to which alternation affects CA. We follow the notation from [4].

An alternating CA (ACA) is a CA which, at each time step, may globally (i.e. at all cells) use either of two transition functions δ_1 and δ_2. The states of Q are partitioned into four classes – accepting states, rejecting states, universal states and existential states. Whether a particular configuration is a universal or an existential configuration is determined by the state of the leftmost cell. The computation tree of an ACA on input w is a binary tree where the root node holds w. The left (right) child of a node holding c holds the configuration obtained by applying δ_1 (δ_2) to c. The input w is said to be accepted if this binary tree has a subtree satisfying the following properties:

The root node of the subtree is the root node of the overall computation tree.

At each node, if the leftmost state in the configuration represented at that node is universal, then both children of the node are present in the subtree.

At each node, if the leftmost state in the configuration represented at that node is existential, then exactly one child of the node is present in the subtree.

At each node, if the leftmost state in the configuration represented at that node is accepting, then the node is a leaf of the subtree.

No leaf of the subtree has a rejecting state as the leftmost state in its configuration. Such a subtree represents an accepting computation of the ACA.

Investigating the power of such ACA necessarily begins with examining the relationship DSPACE(n) = CA. We shall first show that the corresponding equality for alternating computations also holds. Without loss of generality we assume that the Turing machines considered have a single tape.

Lemma 5.1. ASPACE(n) $\subseteq$ ACA.

Proof. This follows from a slight modification of the proof for DSPACE(n) $\subseteq$ CA [21]. The state of the alternating Turing machine (ATM) at each time step indicates whether the ATM is in a universal or an existential state. So, in the simulating ACA, this state must always be represented at the leftmost cell. The ACA holds tape configurations of the ATM in its array in a folded fashion in two tracks, so that the tape square over which the ATM head is positioned is always represented at the

leftmost cell. When the tape head moves, the ACA correspondingly shifts the contents of the two tracks. The changes in the leftmost cell are made at once, and they gradually propagate away to the right. For details of such a construction, see the proof of Theorem 3.8 in [21]. (This proof claims a real-time simulation, but then it uses the input packed two symbols per cell.) This construction requires one data movement step after each real simulation step; thus the ACA takes twice as much time as the ATM and finally performs the same computation. □

Lemma 5.2. ACA $\subseteq$ ASPACE(n).

Proof. Given an ACA, the ASPACE(n) machine construction is akin to constructing an NSPACE(n) machine simulating an NTVCA (Lemma 3.2). The state of the ATM at the beginning of each sweep reflects the state – universal or existential – of the ACA, while the operation within a sweep is deterministic. □

From the above two lemmas Theorem 5.3 now follows.

Theorem 5.3. ACA = ASPACE(n).

The time-bounded ACA classes correspond to time-bounded ASPACE(n) computations. We use ASPT$I(s(n), t(n))$ to denote computations of alternating Turing machines which use $s(n)$ space and run in $t(n)$ time, and ACA($t(n)$ to denote ACA running in $t(n)$ time. The next two lemmas are quite easy to see; they follow from the constructions outlined in Lemmas 5.1 and 5.2.

Lemma 5.4. ASPTI($n, t(n)$) $\subseteq$ ACA($2t(n)$).

Consequently, the DTIME(n) $\subseteq$ lCA containment carries over to alternating computations too.

Carollary 5.5. ATIME(n) $\subseteq$ lACA.

Lemma 5.6. ACA($t(n)$) $\subseteq$ ASPTI(n, O($nt(n)$)).

Thus, if *poly* denotes the class of polynomial-valued functions, then the following corollary holds.

Corollary 5.7. ACA(*poly*) = ASPTI(n, *poly*) $\subseteq$ PSPACE.

It is also quite easy to see, by a process similar to that outlined in Theorem 3.6, that the language of fully quantified Boolean formulas evaluating to True, QBF, is in lACA. Since QBF is PSPACE-complete [1], the membership problem for lACA is also PSPACE-complete.

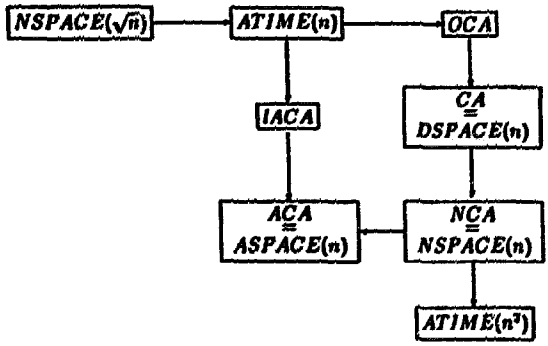


Fig. 6 Inclusions among CA and ATM classes.

Since it is known that NSPACE($s(n)$) is contained in ATIME($s^2(n)$) [2, 10], we thus have the overall setup shown in Fig. 6.

6. Restricted nondeterminism

For a $T(n)$-time computation, an NTVCA as defined in Section 3 looks at all the $2^{T(n)}$ computation paths. We can define restricted versions, where only certain computation paths, whose characteristic bit strings possess some special properties, are of interest. This takes us closer to relativised CA, where exactly one computation path is of interest. Two such restrictions are defined below.

Definition 6.1. A 1-turn (1-kink) NTVCA is a TVCA C which accepts input w if and only if there is an accepting computation of C on w, with a characteristic bit string of the form 0^*1^* $(0^*(\varepsilon + 10^*))$.

A 1-turn NTVCA uses only δ_1 for some time and then switches over to using only δ_2. The nondeterminism is in deciding when to switch from δ_1 to δ_2. So for a $T(n)$-time 1-turn NTVCA, there are $T(n) + 1$ computation paths of interest. A 1-kink NTVCA can use δ_2 at most once; again, for a $T(n)$-time 1-kink NTVCA, there are $T(n) + 1$ computation paths of interest. These paths are shown in Fig. 7.

We can also define restricted versions of PTVCA in a similar fashion.

Definition 6.2. A 1-turn (1-kink) PTVCA is a TVCA C which accepts input w if and only if more than half of the computation paths determined by bit strings of the form 0^*1^* $(0^*(\varepsilon + 10^*))$ are accepting computations.

These classes are important in that they help us identify the amount of nondeterminism needed to enhance the power of other classes. To make this clearer, note

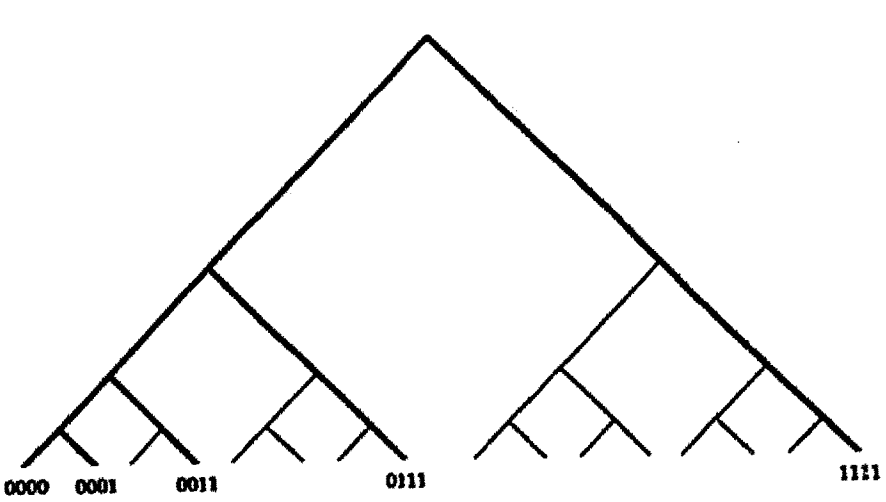


Fig. 7. Restricted nondeterminism computation paths: (a) 1-turn paths; (b) 1-kink paths.

that a $T(n)$-time NTVCA has $2^{T(n)}$ computation paths. Picking any one of these involves choosing $T(n)$ bits, corresponding to the characteristic bit string of the chosen computation path. A $T(n)$-time 1-turn NTVCA, on the other hand, has only $T(n)+1$ computation paths of interest. Picking any one of these involves picking one of the $T(n)$ positions in the charcteristic bit string where the TVCA switches over from using

δ_1 to using δ_2. Since making a choice from $T(n)$ positions would involve setting $\log(T(n))$ bits, the "amount" of choice, due to nondeterminism, available to an NTVCA and to a 1-turn NTVCA differ by an exponential factor. As is to be expected, we will show that the 1-turn classes are quite weak compared to the other NTVCA classes. First we show that 1-turn and 1-kink are equivalent notions; an NTVCA or PTVCA of one type can be simulated by an NTVCA or PTVCA of the other. Before this, we prove an intermediate technical result.

Lemma 6.3. *Let r be a regular expression denoting a subset R of* $\{0,1\}^*$. *Given any* NTVCA *C, we can produce a modified* NTVCA *C' which performs the same computation as C, but additionally, along each computation path, also indicates whether the bit string determining the computation path belongs to R.*

Proof. Let C be an NTVCA, and let M be a deterministic finite-state machine (FSM) accepting R. We construct the required NTVCA C' to function as follows: The states of C' are 2-tuples. The first component of each cell, put together, gives the configuration of C. In the second component, the state of M while processing the bit string corresponding to the current computation path is recorded. Thus, along any computation path, at any given time step, the second component of the state of each cell holds the same value. The bit string determining the computation path is in R if and only if this value is a final state of M. □

This is in fact a weak result in that each cell is able to recognise R by acting as an FSM in isolation. By collectively using all cells in the array, some nonregular subsets of bit strings can also be recognised at the leftmost cell; however, for our purposes now, regular sets suffice.

Theorem 6.4.

$T(n)$-*time* 1-*turn* NTVCA = $T(n)$-*time* 1-*kink* NTVCA.

$T(n)$-*time* 1-*turn* PTVCA = $T(n)$-*time* 1-*kink* PTVCA.

Proof. Consider simulating a 1-turn NTVCA by a 1-kink NTVCA. Let the 1-turn NTVCA be $C = (Q, \#, \delta_1, \delta_2, A)$. We define a 1-kink NTVCA, with transition functions h_1 and h_2, and with one unmarked state and one marked state corresponding to each state in Q. h_1 on unmarked states acts as δ_1. h_2 on unmarked states acts as δ_2 and also marks the resulting states. Subsequently, all operation is on the marked version of the states. h_1 on marked states acts as δ_2. (If h_2 encounters marked states, then the result is immaterial, since this does not correspond to a 1-kink path.) Thus the 1-turn path 0^i1^j using δ_1 and δ_2 is simulated by the 1-kink path 0^i10^{j-1} using h_1 and h_2.

The other inclusions can be similarly shown. □

Since 1-turn and 1-kink nondeterminism allow less choice, it is to be expected that the classes they define are contained in the unrestricted nondeterminism classes. This is shown in the proof of the following theorem.

Theorem 6.5. (a) *$T(n)$-time* 1-*turn* NTVCA $\subseteq$ *$T(n)$-time* NTVCA.
(b) *$T(n)$-time* 1-*turn* PTVCA $\subseteq$ *$T(n)$-time* PTVCA.

Proof. (a) A 1-turn NTVCA must have an accepting computation with a characteristic bit string $0^i1^{T(n)-i}$ to accept its input. We can design an NTVCA which uses the transition functions of the given 1-turn NTVCA, and also checks the regular expression 0^*1^* along its computation paths, as described in Lemma 6.3. A state is an accepting state if and only if its first component is an accepting state for the 1-turn NTVCA and its second component is an accepting state for an FSM accepting 0^*1^*. Thus if the NTVCA has an accepting computation, then it must be along a 1-turn path. Hence the NTVCA accepts exactly the same language as the 1-turn NTVCA, and within the same time.

(b) A 1-turn PTVCA has $T(n)+1$ computation paths of interest. A PTVCA, on the other hand, has to consider $2^{T(n)}$ computation paths. In a simulation of a 1-turn PTVCA, $2^{T(n)}-T(n)-1$ of these carry no information; they correspond to invalid paths. To prevent these computation paths from affecting the overall outcome, we must ensure that exactly half of these are accepting computations. Consider the following method of division of these paths into accepting and rejecting paths:

Invalid paths (i.e. of the form $\Sigma^*10\Sigma^*$): $2^{T(n)}-T(n)-1$.

1. Paths beginning with 0 (i.e. of the form $0^+1^+0^+\Sigma^*$): $2^{T(n)-1}-T(n)$ paths.
2. Paths beginning with 1 (i.e. of the form $1^+0^+\Sigma^*$): $2^{T(n)-1}-1$ paths.

(a) Paths of the form 1^+0^+: $T(n)-1$ paths.

(i) Paths with odd number of 1's (i.e. of the form 1^k0^j, where $0<k,j<T(n)$ and k is odd): $\lceil T(n)/2\rceil$ paths.

(ii) Paths with even number of 1's (i.e. of the form 1^k0^j, where $0<k,j<T(n)$ and k is even): $\lceil T(n)/2\rceil-1$ paths.

(b) Other paths (i.e. of the form $1^+0^+1\Sigma^*$): $2^{T(n)-1}-T(n)$ paths.

Make all paths in 1 and 2(a)(i) accepting, and all paths in 2(a)(ii) and 2(b) rejecting.

The accept cell can determine the type of the path currently being followed using the procedure described in Lemma 6.3.

Let A and R denote the number of invalid accepting and rejecting paths, respectively. Then $A=2^{T(n)-1}-T(n)+\lceil T(n)/2\rceil$ and $R=2^{T(n)-1}-T(n)+\lceil T(n)/2\rceil-1$. Clearly, if $T(n)$ is odd, then $A=R$, as desired. If $T(n)$ is even, then $A=R+1$. But in this case, the number of valid paths is itself odd ($T(n)+1$), and so the 1-turn PTVCA cannot have a tie between the number of accepting and rejecting paths. So this distribution of invalid paths does not introduce error. Thus, in either case, the PTVCA accepts its input if and only if the number of valid accept paths exceeds the number of valid reject paths; i.e. if and only if the 1-turn PTVCA accepts its input. □

For these restricted choice classes also, we show below that an NTVCA class is contained in the corresponding PTVCA class.

Theorem 6.6. *$T(n)$-time* 1-*turn* NTVCA $\subseteq$ *$T(n)$-time* 1-*turn* PTVCA.

Proof. A $T(n)$-time 1-turn NTVCA has $T(n)+1$ computation paths of interest, i.e. valid computation paths. If any of these is an accepting computation, then the input is to be accepted. To achieve the same effect in a probabilistic computation within the same time, we must construct a $T(n)$-time PTVCA where half of the valid computation paths are accepting paths and each of the remaining valid computation paths simulates two distinct computation paths of the NTVCA.

This construction differs from that outlined in the proof of Theorem 4.2 in only one aspect; we now want to consider only 1-turn paths and partition them into equal-sized sets. But Lemma 6.3 allows us to do such identifying and partitioning easily. The details are left to the reader. □

Corollary 6.7.

Real-time 1-*turn* NTVCA $\subseteq$ *real-time* 1-*turn* PTVCA.

Linear-time 1-*turn* NTVCA $\subseteq$ *linear-time* 1-*turn* PTVCA.

Finally, we observe that even linear-time 1-turn PTVCA are contained in P. This merely points out that the weakness of 1-turn computations is not overcome by going from nondeterministic to probabilistic computations.

Lemma 6.8. *Linear-time* 1-*turn* PTVCA $\subseteq P$.

Proof. A linear-time TVCA, whether nondeterministic or probabilistic, has only polynomially many valid computation paths. So a deterministic Turing machine can check them all sequentially. □

Clearly, this argument holds even if the PTVCA requires $p(n)$ time for some polynomial p.

This last result shows the limitations of 1-turn nondeterminism. However, we believe that even this much nondeterminism can increase the power of a class. As a specific example, consider any language L and define $\exists\text{MID}(L)$ as follows:

$$\exists\text{MID}(L) = \{xyz \in \Sigma^* \mid |x| = |z|,\ y \in L\}.$$

For any L in rCA, we can show that $\exists\text{MID}(L)$ can be accepted by a real-time 1-turn NTVCA. We do not know whether, for L in rCA, $\exists\text{MID}(L)$ can always be accepted by an rCA. Similarly, if we define $\exists\text{PRE}(L)$ as follows,

$$\exists\text{PRE}(L) = \{xy \in \Sigma^* \mid x \in L\},$$

then we can show that for any L in fCA, $\exists$PRE(L) is in linear-time 1-turn NTVCA. We do not know of any fCA construction to accept $\exists$PRE(L). (However, if L is an rCA language, then $\exists$PRE(L) can also be shown to be an rCA language.) These and other such closure properties are studied in [16, 19].

The idea behind examining 1-turn NTVCA is essentially to see how many distinct computation paths need to be checked for acceptance. The concept can be generalised to k-turn for some constant k, and to finite-turn. A k-turn NTVCA is an NTVCA where an accepting path, if one exists, alternates between using δ_1 and δ_2 at most k times. Similarly, we can consider k-turn PTVCA. Clearly, k-turn is contained in $(k+1)$-turn for NTVCA. The nontrival question is whether the containment is strict. It is easy to see that k-turn paths and k-kink paths (δ_2 is used at most k times) also have characteristic bit strings representable by regular expressions; thus Theorems 6.4 and 6.5 hold for k-turn (k-kink) NTVCA too. Lemma 6.8 also holds for linear-time k-turn NTVCA and k-turn PTVCA, because it is easy to see that the number of distinct computation paths of interest in a k-turn NTVCA is $O((T(n))^k)$.

For PTVCA, showing that k-turn is contained in $(k+1)$-turn is not as easy as for NTVCA. As in the proof of Theorem 6.5, we need to show that the paths which are $(k+1)$-turn but not k-turn can be divided equally between accepting and rejecting computations, so that they do not affect the overall outcome. It is straightforward, but tedious, to outline such a division, using Lemma 6.3. For details, see [6]. Thus, we have the following result.

Theorem 6.9. $\forall k > 0$, *k-turn $T(n)$-time* PTVCA $\subseteq$ *$(k+1)$-turn $T(n)$-time* PTVCA.

The other results in this section can be similarly generalised.

7. Closure properties

In this section we examine some closure properties of the language classes defined in the preceding sections.

Theorem 7.1. *If L_1 and L_2 can be accepted by* NTVCA *in $T(n)$ time, then $L_1 \cup L_2$ can be accepted by an* NTVCA *in $T(n)$ time.*

Proof. Let C_1 and C_2 be the NTVCA accepting L_1 and L_2, respectively. We can construct an NTVCA C which simulates C_1 and C_2 in two separate channels and thus accepts $L_1 \cup L_2$. □

Theorem 7.2. *If L_1 and L_2 can be accepted by* NTVCA *in time $T_1(n)$ and $T_2(n)$, respectively, then $L_1 \cap L_2$ can be accepted by an* NTVCA in $T_1(n) + T_2(n)$ *time.*

Proof. A construction similar to that outlined in the above proof will not work in this case, because even if x belongs to both L_1 and L_2, the accepting computations of C_1 and C_2 need not have the same characteristic bit string. So an NTVCA accepting $L_1 \cap L_2$ must run through all combinations of a computation of C_1 and a computation of C_2.

To achieve the claimed time-bound, we interleave the computations of C_1 and C_2. Two channels are created in the array of cells. One channel is updated at odd time steps as per C_1, the other at even steps as per C_2. This ensures that each computation can proceed with an independent characteristic bit string. If the computation in any channel reaches an accepting state, thenceforth that channel stays in an accepting state. The new CA accepts if and only if both its channels are in accepting states. Clearly, if the input is in $L_1 \cap L_2$, then this happens within $T_1(n) + T_2(n)$ time. □

Corollary 7.3. *Linear-time* NTVCA *are closed under union and intersection.*

Since NTVCA = NSPACE(n), Theorem 7.4 follows.

Theorem 7.4. NTVCA *are closed under complementation.*

However, it does not seem likely, especially in the light of Theorem 3.6, that real-time or linear-time NTVCAs are closed under complementation.

Note that Theorem 7.1 goes through even if we consider k-turn NTVCA. Theorem 7.2 does not, because interleaving the computations of C_1 and C_2 can introduce many more turns. In this case, therefore, we go through the computations of the two CAs sequentially, giving the following result.

Theorem 7.5. *Let L_1 and L_2 be accepted by k-turn and m-turn* NTVCA *in $T_1(n)$ and $T_2(n)$ time, respectively. Then*

(a) *$L_1 \cup L_2$ can be accepted by a* $\max(k,m)$*-turn* NTVCA *in* $\max(T_1(n), T_2(n))$ *time.*

(b) *$L_1 \cap L_2$ can be accepted by a j-turn* NTVCA *in* $T_1(n) + 2n + T_2(n)$ *time, where*

$$j = \begin{cases} k+m+1 & \text{if } k \text{ is odd,} \\ k+m & \text{otherwise.} \end{cases}$$

(c) *If $T_1(n)$ is strongly time-constructible, then $L_1 \cap L_2$ can be accepted by a j-turn* NTVCA *in $T_1(n) + T_2(n)$ time, where j is as in* (b).

Proof. (a) is straightforward. To show (b) and (c), the technique of Theorem 7.2 will not work directly, as explained above. Instead, consider the following method. Let C_1 and C_2 be the NTVCAs accepting L_1 and L_2, respectively. The NTVCA C accepting $L_1 \cap L_2$ begins simulating C_1 along all paths. If the input x belongs to L_1, then acceptance will be detected within $T_1(n)$ steps. When this happens, C initiates a firing squad synchronisation algorithm. This requires $2n$ steps. When the cells synchronise,

they start simulating C_2. If x belongs to L_2 as well, this will be detected within another $T_2(n)$ steps. Thus, if x is in $L_1 \cap L_2$, C will accept x within $T_1(n) + 2n + T_2(n)$ steps.

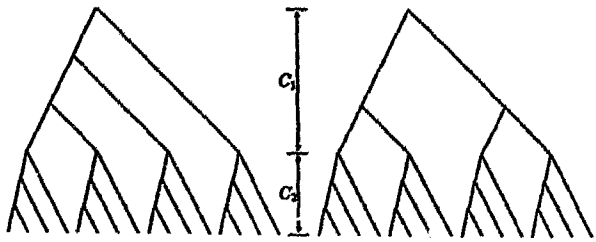
If $T_1(n)$ is strongly time-constructible, then the synchronisation stage can be avoided. C simply begins simulating C_1, while simultaneously computing $T_1(n)$. After $T_1(n)$ steps, the whole array of cells switches over to simulating C_2. The leftmost cell accepts its input if and only if both parts of the simulation end in accepting states.

The number of turns is explained as follows. An accepting path has at most k turns from the simulation of C_1, plus at most m turns from the simulation of C_2, plus possibly one more turn in changing over from the simulation of C_1 to that of C_2, and is thus a $(k + m + 1)$-turn path. The additional turn is not needed if k is even, because then the path of C_1 with maximum number of turns ends with δ_1 in use. See Fig. 8. □

Lastly, we consider the closure of PTVCA language classes under some simple operations. Closure under union or intersection does not seem to hold; the proofs of Theorems 7.1 and 7.2 do not carry over, since we now need to *count* the number of accepting computations of C_1 and C_2. On the other hand, for PTVCA, closure under complementation is relatively easy to show.

Theorem 7.6. *If L can be accepted by a* PTVCA *in $T(n)$ time, then $\bar{L}$ can be accepted by a* PTVCA *in $T(n) + 1$ time.*

Proof. Merely exchanging the accepting and the nonaccepting states of the PTVCA accepting L fails in case there are an equal number of accepting and rejecting computations. However, using one extra time step, this difficulty can be overcome. One extra step generates $T(n)$ additional computation paths. These can be divided, by using Lemma 6.3, between dummy accepting and rejecting paths in such a way that ties are correctly handled. The details are straightforward and are omitted. □



(a) 1-turn followed by 1-turn gives 3-turn paths

(b) 2-turn followed by 1-turn gives 3-turn paths

Fig. 8. Intersection of finite-turn NTVCA languages: (a) 1-turn followed by 1-turn gives 3-turn paths; (b) 2-turn followed by 1-turn gives 3-turn paths.

8. Conclusion

In this paper, we have presented a new mechanism for introducing nondeterministic, probabilistic and alternating computations in the cellular automaton model. We have compared our notion of nondeterminism with the traditional notion. We have also defined restricted versions of nondeterministic computations and have explored the power of the resulting automata. The relations between such language classes are depicted in Figs. 6 and 9. In Fig. 9, known (i.e. existing) classes are shown in ovals and the newly defined classes are shown in boxes. The containments depicted between ovals are known results; the other containments have been shown in this paper. All this investigation essentially aims at refining the open problems in the containments

$$\mathrm{rOCA} \subset \mathrm{rCA} = \mathrm{lOCA} \subseteq \mathrm{lCA} \subseteq \mathrm{OCA} \subseteq \mathrm{CA} = \mathrm{DSPACE}(n) \subseteq \mathrm{NSPACE}(n)$$

and thus providing an alternative approach to solving the problems. A lot more investigation still remains to be done. A relatively unexplored area is the use of traditionally defined nondeterminism in time-bounded CA classes, i.e. studying classes like rNCA and lNCA. We feel that some questions concerning the power of these classes can be answered independent of the long-standing open questions regarding rCA, lCA and CA.

The alternating CA classes are useful in considering closures of rCA and lCA languages under various operations. If lCA are not known to be closed under some

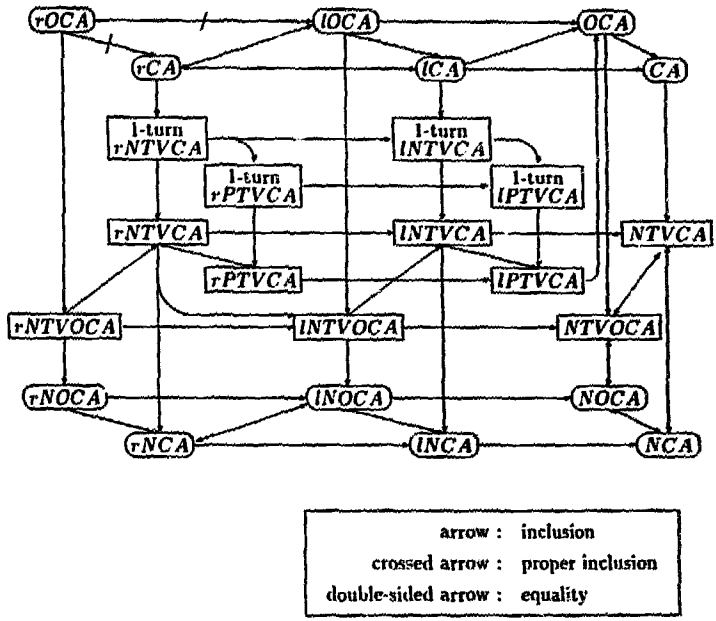


Fig. 9. Deterministic, nondeterministic and probabilistic CA classes.

operation, we would like to identify the smallest CA class containing this closure; this gives us some idea of the complexity of the operation considered. Alternating CA classes help in this respect. Some such closure results have been studied in [19], which deals with the closure of CA classes under a wide variety of language operations.

Acknowledgement

We would like to thank an anonymous referee for pointing out that firing squad synchronisation can be avoided in checking intersections. This allowed us to present a stronger version of Theorem 7.2.

We also thank Dr. Bruno Durand for pointing out how Theorem 3.6 goes through for rNTVOCA. Our earlier version only proved it for rNTVCA. He also went over our entire paper in great detail, and his comments helped us to rewrite the paper in a much more readable fashion. In particular, the proofs of Lemma 3.3, Theorem 3.8, and Theorem 4.2 are significantly more comprehensible now than they were in an earlier version.

References

[1] J.L. Balcázar, J. Diaz and J. Gabarró, *Structural Complexity I*, EATCS Monograph Series, Vol. 1 (Springer, Berlin, 1988).
[2] J.L. Balcázar, J. Diaz and J. Gabarró, *Structural Complexity II*, EATCS Monograph Series Vol. 22 (Springer, Berlin, 1990).
[3] W. Bucher and K. Culik II, On real-time and linear-time cellular automata, RAIRO *Inform. Theor.* 18 (1984) 307–325.
[4] A.K. Chandra, D.C. Kozen and L.J. Stockmeyer, Alternation, *J. ACM* 28 (1981) 114–133.
[5] J.H. Chang, O.H. Ibarra and A. Vergis, On the power of one-way communication, *J. ACM* 35 (1988) 697–726.
[6] C. Choffrut and K. Culik II, On real-time cellular automata and trellis automata, *Acta Inform.* 21 (1984) 393–409.
[7] K. Culik II, J. Gruska and A. Salomaa, Systolic trelis automata Part I, *Internat. J. Comput. Math.* 15 (1984) 195–212.
[8] K. Culik II, J. Gruska and A. Salomaa, Systolic trellis automata Part II, *Internat. J. Comput. Math.* 16 (1984) 3–22.
[9] C. Dyer, One-way bounded cellular automata, *Inform. and Control* 44 (1980) 261–281.
[10] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation* (Addison-Wesley, Reading, MA, 1979).
[11] O.H. Ibarra and T. Jiang, On one-way cellular arrays, *SIAM J. Comput.* 16 (1987) 1135–1154.
[12] O.H. Ibarra and T. Jiang, Relating the power of cellular arrays to their closure properties, *Theoret. Comput. Sci.* 57 (1988) 225–238.
[13] O.H. Ibarra and S.M. Kim, Characterizations and computational complexity of systolic trellis automata, *Theoret. Comput. Sci.* 29 (1984) 123–153.
[14] O.H. Ibarra, S.M. Kim and S. Moran, Sequential machine characterizations of trellis and cellular automata and applications, *SIAM J. Comput.* 14 (1985) 426–447.
[15] O.H. Ibarra, M. Palis and S.M. Kim, Some results concerning linear iterative (systolic) arrays, *J. Parallel Distributed Comput.* 2 (1985) 182–218.

[16] M. Mahajan, Studies in language classes defined by different types of time-varying cellular automata, Ph.D. Thesis, Indian Institute of Technology, Madras, India, 1993.
[17] M. Mahajan and K. Krithivasan, Relativised cellular automata and complexity classes, in: *Proc. 11th Internat. FST&TCS Conf.* New Delhi (1991) 172–185; Lecture Notes in Computer Science, Vol. 560.
[18] M. Mahajan and K. Krithivasan, Some results on time-varying and relativised cellular automata, *Internat. J. Comput. Math.* 43 (1992) 21–38.
[19] M. Mahajan and K. Krithivasan, Language operations on cellular automata classes, *J. Math. Phys. Sci.* 27 (1994).
[20] A.R. Smith III, Cellular automata complexity trade-offs, *Inform. and Control* 18 (1971) 466–482.
[21] A.R. Smith III, Real-time language recognition by one-dimensional cellular automata, *J. Comput. System Sci.* 6 (1972) 233–253.