Shrinking and Expanding Cellular Automata

Augusto Modanese, Thomas Worsch

Shrinking and Expanding Cellular Automata

Augusto Modanese and Thomas Worsch

Institute for Theoretical Informatics
Karlsruhe Institute of Technology
Am Fasanengarten 5
76131 Karlsruhe, Germany
augusto.modanese@student.kit.edu, worsch@kit.edu

Abstract. Inspired by shrinking cellular automata (SCA), we investigate another variant of the classical one-dimensional cellular automaton: the shrinking and expanding cellular automaton (SXCA). In addition to the capability to delete some cells as in SCA, an SXCA can also create new cells between already existing ones. It is shown that there are reasonably close (polynomial) relations between the time complexity of SXCA and the space and time complexity of Turing machines and alternating Turing machines respectively. As a consequence the class of problems decidable in polynomial time by SXCA coincides with **PSPACE**.

1 Introduction

Rosenfeld, Wu and Dubitzki [5] have introduced so-called shrinking cellular automata.

The idea is that the transition function allows a cell to "delete" itself. If one cell does that, then its both neighbors to the left and right become direct neighbors, and similarly if several cells are deleted (a precise definition will be given in Sect. 2). Rosenfeld and Wu observed that some formal languages can be recognized in less than real-time, using the shrinking process to mimic reductions according to the productions of a context free grammar.

It took more than 30 years before Kutrib et al. [3] started a more thorough investigation of shrinking CA, explaining relations between shrinking and non-shrinking CA for small time bounds, i. e. real-time and linear time.

It seemed natural to also have a first look at cellular automata which can not only shrink, but also have the opposite ability expand by generating additional cells between already existing ones. As in the shrinking case, at least at first sight, it is only clear how to define this in the one-dimensional case. It turns out that shrinking and expanding CA are quite powerful.

The remainder of this paper is organized as follows. In Section 2 we define shrinking and expanding cellular automata and related notions like the set **SXCAP** of problems decidable by SXCA in polynomial time. In Section 3 we relate time complexity of alternating Turing machines to that of SXCA, implying that **PSPACE** is included in **SXCAP**. In Section 4 we obtain results showing that

in particular **SXCAP** in included in **PSPACE**. (Hence SXCA belong to the so-called second machine class [6].)

This paper is partially based on the bachelor thesis by the first author [4].

2 Basics

The set of all integers is denoted $\mathbb{Z}$, the set of all positive integers $\mathbb{N}_+$ and $\mathbb{N}_0 = \mathbb{N}_+ \cup \{0\}$. For two sets A and B the set of all total functions mapping from A to B is denoted B^A.

2.1 Standard CA

In this paper we are looking at one-dimensional cellular automata with the standard neighborhood $N = \{-1, 0, 1\}$ of radius 1. Let Q denote the set of states of a single cell.

Then a global configuration is a function $c\colon \mathbb{Z} \to Q$. The local transition function $\delta\colon Q^N \to Q$ induces a global transition function $\Delta\colon Q^{\mathbb{Z}} \to Q^{\mathbb{Z}}$ in the usual way: $\Delta(c)(x) = \delta(\ell_{c,x})$ where $\ell_{c,x}\colon N \to Q\colon n \mapsto c(x+n)$.

A (finite or infinite) *computation* is a sequence $(c_0, c_1, \dots)$ of configurations such that $c_{t+1} = \Delta(c_t)$ holds for all t.

We will always assume that there is a special state q such that the local transition functions satisfies two properties:

- As long as the cells in the neighborhood of a cell x are all in state q the next state of x is q again, i. e. q is a quiescent state.
- A cell which is not in state q will never enter state q in the next step.

We will only look at decision problems. The input for a CA is always a non-empty word $w \in A^+$ over some finite alphabet $A \subset Q$. The initial configuration of a word $w = a_0 \cdots a_{n-1}$ of length $|w| = n$ with $a_i \in A$ for all $i \le |w|$ is defined as

$$c_w\colon \mathbb{Z} \to Q\colon x \mapsto \begin{cases} a_i, & \text{iff } 0 \le x = i < |w| \\ q, & \text{otherwise} \end{cases} . \tag{1}$$

The requirements for q ensure that all configurations reachable from initial configurations c_w as just defined all consist of a finite connected block of cells all of which are in a non-quiescent state, extended on both sides with infinitely many cells in state q. Of course the block of non-quiescent cells may grow during a computation.

For the acceptance and rejection of input words we assume that a special state a ("accept") and a special state r ("reject") are present in Q. A configuration c is *final* iff cell 0 is in the accepting or the rejecting state. The computation $(c_w = c_0, c_1, \dots, c_t)$ for an input w is halting if c_t is final and if c_t is the only final configuration in the computation. We call c_t the result configuration for w. An input is accepted or rejected iff in the result configuration cell 0 is accepting or rejecting respectively.

The *support* of a configuration c is the set $\{\, x \mid c(x) \neq q \,\}$. Let $\mathcal{C}_{fin}$ denote the set of all configurations $c\colon \mathbb{Z} \to Q$ with finite support.

2.2 Shrinking and expanding CA

We now have a look at non-standard CA which can *shrink* by deleting cells "not needed" any longer [5, 3] and which can *expand* by generating new cells between already existing ones [4].

In the present paper the focus is on CA which are allowed to do both during a computation. The following definition of *shrinking and expanding CA (SXCA)* is adapted to this case.

Besides the set Q of proper states there is a "pseudo state" $\otimes$. Formally we assume $\otimes \notin Q$ and write $\bar{Q} = Q \cup \{\otimes\}$. In analogy to $\mathcal{C}_{fin}$ let $\bar{\mathcal{C}}_{fin}$ denote the set of all functions $\bar{c}\colon \mathbb{Z} \to \bar{Q}$ with having finite support $\{\, x \mid \bar{c}(x) \neq q \,\}$.

The local transition function $\delta = (\varepsilon, \sigma)$ of an SXCA is given by two functions $\varepsilon\colon Q^N \to \bar{Q}$ and $\sigma\colon Q^N \to \bar{Q}$. Given a configuration $c \in \mathcal{C}_{fin}$ these induce a global transition of the SXCA defined in two substeps.

- To a given configuration c first ε and σ are applied to each local configuration $\ell_{c,x}$ observed by cell x in c. This gives rise to a function $X\colon \mathcal{C}_{fin} \to \bar{\mathcal{C}}_{fin}$ in the following way.

$$\text{for all } x \in \mathbb{Z}\colon \quad \begin{aligned} X(c)(2x-1) &= \varepsilon(\ell_{c,x}) \\ X(c)(2x) &= \sigma(\ell_{c,x}) \end{aligned} \tag{2}$$

  One can imagine that the existing cells are pulled apart by doubling their indices entering the preliminary states $\sigma(\ell_{c,x})$ respectively. To the left of each such cell a "new" cell is generated with preliminary state $\varepsilon(\ell_{c,x})$.
- Secondly, the cells which are in state $\otimes$ "are removed" and the "remaining cells are renumbered" in the following sense: Given any $\bar{c} \in \bar{\mathcal{C}}_{fin}$, define a renumbering

$$\begin{aligned} r_{\bar{c}}\colon \mathbb{Z} &\to \mathbb{Z} \\ 0 &\mapsto \min\{\, x' \geq 0 \mid \bar{c}(x') \in Q \,\} \\ x &\mapsto \min\{\, x' > r_{\bar{c}}(x-1) \mid \bar{c}(x') \in Q \,\}, \quad \text{if } x > 0 \\ x &\mapsto \max\{\, x' < r_{\bar{c}}(x+1) \mid \bar{c}(x') \in Q \,\}, \quad \text{if } x < 0 \end{aligned} \tag{3}$$

  This induces a function $S\colon \bar{\mathcal{C}}_{fin} \to \mathcal{C}_{fin}$ by

$$S(\bar{c})(x) = \bar{c}(r_{\bar{c}}(x)) \tag{4}$$

  Figure 1 shows an example.

The composition $\Delta = S \circ X$ is the global transition function of the SXCA.

CA in the classical definition are a special case, where existing cells are never deleted, i. e. all $\sigma(\ell) \neq \otimes$, and additional cells are never generated, i. e. all $\varepsilon(\ell) = \otimes$. Shrinking CA are recovered by again requiring all $\varepsilon(\ell) = \otimes$ and interpreting $\otimes$ as the "dissolve" value used in [3].

The definitions for the notions *computation*, *acceptance*, etc., are defined in the same way as for standard CA.

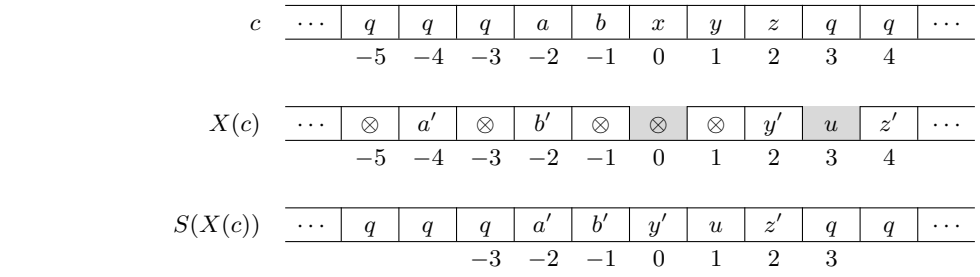

Fig. 1. The two substeps of a global transition of an SXCA: The cells in states a, b and y should enter states $a' = \sigma(q, a, b)$, $b' = \cdots$ and $y' = \cdots$ respectively without generating an additional cell to their left ($\varepsilon(q, a, b) = \otimes, \dots$). The cell in state x deletes itself (left gray cell, $\sigma(b, x, y) = \otimes$) and also doesn't generate an additional cell to its left ($\varepsilon(b, x, y) = \otimes$). The cell in state z enters $z' = \sigma(y, z, q)$ and generates a cell in state $u = \varepsilon(y, z, q)$ to its left (second gray cell). Cells which are "not needed" are deleted and the remaining ones "renumbered" during the second substep.

The *time complexity* of an SXCA is the function $f\colon \mathbb{N}_+ \to \mathbb{N}_+$ where $f(n)$ is the maximum number of steps needed for an input of length n until it is accepted or rejected. The set of problems which can be decided by SXCA with polynomial time complexity will be written **SXCAP**.

3 Relating ATM time to SXCA time

Fellah and Yu [2] have shown an intricate close relation between tree-shaped cellular automata (with sequential input at the root) and alternating Turing machines (ATM) [1].

While the repeated generation of additional cells in an SXCA can yield an exponential growth of the number of cells —which superficially looks similar to the exponentially growing number of cells reachable from the root of a tree— there is a difference: In a tree the root can quickly broadcast a value in t steps to 2^t cell. On the other hand, in a growing CA a cell (the "root") from which the generation of 2^t cells has been initiated, *cannot* in t steps broadcast a value to 2^t cells (once the generation process is complete). This seems to increase the complexity of some constructions. Still, we will prove in this section:

Theorem 1. *An alternating TM with time complexity bounded by some function $t(n)$ can be simulated by an SXCA whose time complexity is $O((t(n))^2)$.*

Since alternating time is polynomially related to sequential space [1], one immediately obtains:

Corollary 2. PSPACE $\subseteq$ SXCAP, *i. e. the set of problems which can be decided by SXCA in polynomial time comprises* **PSPACE**.

[			s				]
[	a	b	c	d	e	f	]

Fig. 2. A block storing 6 tape symbols. The head of the ATM is visiting the third square and the ATM is in state s.

[	[			s				]	∃/∀	[			s'				]	]
[	[	a	b	c	d	e	f	]		[	a	b	c	d	e	f	]	]

Fig. 3. Two copies of a block storing 6 tape symbols. The head of the ATM is visiting the third square and the ATM is in state s. Two "versions" of state s are used in order to allow the simulation of the ATM for its two possible continuations. The extra cell between the block remembers whether s is an existential or a universal state.

We now describe a construction proving Theorem 1. We assume the reader is familiar with ATMs.

Let T be an alternating Turing machine with one work tape on which also the input is provided at the beginning of a computation. Furthermore let S denote the set of states, B the set of tape symbols, and $A \subset B$ the input alphabet of the ATM. One may assume that in each situation there are exactly two possible continuations for the ATM. We will construct an SXCA C which for each input $w \in A^+$ checks whether T would accept or reject w and then accepts or rejects w accordingly.

Without loss of generality T never writes blank symbols, and hence there is always one contiguous tape segment with non-blank symbols comprising all (input and) visited squares. The SXCA C uses k consecutive cells to store such segment of length k with symbols $a_1 \cdots a_k$, along with a representation of the ATM head being positioned on symbol a_p, and the ATM being in state $s \in S$. Each cell stores one symbol, and cell p in addition the state s while the others store a value indicating that the head is not on the respective square. In addition the block is surrounded by two "bracket cells". See Fig. 2 for an example of such a *block* of the SXCA.

In its very first step, C changes the states storing the input symbols such that the cells represent the corresponding initial ATM configuration (as described above). Afterwards, C iterates the following procedure:

1. New cells are generated to the left of the block and the block contents are copied to it. For each state s of the ATM, the states in both blocks are distinguished by using an unprimed representation s in the left one, and a primed representation s' in the right one.
2. Between the blocks an additional cell is used to remember whether the ATM state is an existential or a universal one. The pair of blocks is surrounded by an additional pair of brackets. See Fig. 3.
3. In the left block, the first alternative for the next step of the ATM is simulated, while the second alternative is simulated in the right one. The SXCA can distinguish between the two cases because different representations of the state are used in the two blocks.

4. In each block C checks whether the ATM state is a final one.
   - If this is not the case, then C continues with step 1 above.
   - If the state is final, C breaks out of the loop for the block and continues with step 5 below.
5. If the ATM state is a final one, then signals are sent in both directions to the nearest enclosing brackets deleting all cells which do not store the state or the brackets. Thus, the block is shrunk to a single cell storing state a or state r.
6. Eventually, an extra cell storing a quantifier will observe final states in both its neighboring cells. It then sends signals in both directions to the nearest enclosing brackets. This causes the deletion of the two cells which contain final states, as well as of the ones containing the brackets. The cell itself enters state a or r according to the following rules:
   - It enters a in case the quantifier is $\exists$ and at least one of the neighboring final states was a, or if the quantifier is $\forall$ and both neighboring final states were a.
   - Otherwise the cell enters state r.

As long as a configuration is not a final one, the block, inside the brackets, representing it will be replaced by two bracketed blocks and the $\exists/\forall$ marker. The resulting nested hierarchy of brackets will be deleted from the inside out, once final configurations have been reached.

In the end, all non-quiescent cells are deleted except for a single cell, which is in state a or r if and only if the ATM would accept or reject the input, respectively.

It remains to estimate the time complexity of the SXCA. If the ATM is $t(n)$ time-bounded, then, for an input of length n, it will never make more than $t(n)$ steps during a possible computation and hence never visit more than $t(n)$ squares. Therefore, the length of all blocks which have to be copied during step 1 is at most $t(n)$. The simulation of one step of the ATM requires only constant time. Consequently after $O((t(n))^2)$ SXCA steps in every block a final state is reached.

Deleting the non-state cells within a block again only takes time $t(n)$, and this also has to be done $t(n)$ times. Therefore the result of the ATM is available after $O((t(n))^2)$ steps.

4 Relating SXCA time to TM space

At the beginning of the previous section it has already been pointed out that the (compared to tree CA) somewhat restricted possibilities of communication between cells may manifest themselves in somewhat less efficient simulations of other models by SXCA. On the other hand, the ability to delete a possibly large number of cells simultaneously, until now seems to impede particularly efficient "simulations" of SXCA by other models such as ATM. Instead of elaborating a somewhat arduous ATM, in this section we only exhibit a space efficient approach:

Theorem 3. *An SXCA whose time complexity is bounded by $t(n)$ can be simulated by a deterministic TM with space complexity bounded by $O((t(n))^2)$.*

Together with theorem 1 this implies:

Corollary 4. SXCAP = PSPACE, *i. e. the set of problems which can be decided by SXCA in polynomial time is exactly* **PSPACE**.

Thus SXCA are a model in the so-called *second machine class* [6].

In order to prove Theorem 3 we first describe an algorithm and discuss its space complexity afterwards. For better readability at some points we have forced page breaks to make sure that pseudo code and corresponding explanations are on the same page.

We proceed "top down", starting with a very simple function, and proceed in three refining steps.

Assuming that there is a function STATE(cell x, time t) which computes the state of cell $x \in \mathbb{Z}$ at time $t \in \mathbb{N}_0$, it is trivial to determine whether the SXCA accepts a given input w. One just has to find the first time, when cell 0 enters a final state and check that. See algorithm 1.

```
⟨ finalState returns a or r if the SXCA accepts or rejects an input w,
 respectively. ⟩
function final state ← finalState() is
    t ← 0
    repeat
        t ← t + 1
        s ← State(0, t)
    until s ∈ {a, r}
    return s
end
```

Algorithm 1: Finding the final result of an SXCA.

Next, we describe the process of determining the state of a cell x at time t. The case $t = 0$ is trivial. For $t > 0$, it is important to remember how the transition from a configuration c to $S(X(c))$ was defined in Sect. 2; in particular see Fig. 1. There are two possibilities how a cell z at time $t - 1$, given its state and those of its both neighbors, can give rise to the state of cell x at time t: (i) by changing its own state using σ; or (ii) by generating a new cell using ε. Assume that z, along with a flag *new* which signals whether x was created using ε or not, can be computed by a function $\textsc{Predecessor}(x, t)$. Then algorithm 2 shows a straightforward way to compute the state of cell x at time t.

```
⟨ Compute the state of cell x at time t: ⟩
function state s ← State(cell x, time t) is
    if t = 0 then
        ⟨ Initial configuration for input w: ⟩
        if 0 ≤ x < |w| then
            return w_x    ⟨ the x-th symbol of input w ⟩
        else
            return q
        end
    else
        (z, new) ← Predecessor(x, t)
        q_{-1} ← State(z − 1, t − 1)
        q_0 ← State(z, t − 1)
        q_1 ← State(z + 1, t − 1)
        if new then
            ⟨ cell was "generated" during transition to time t ⟩
            return ε(q_{-1}, q_0, q_1)
        else
            ⟨ cell already existed ⟩
            return σ(q_{-1}, q_0, q_1)
        end
    end
end
```

Algorithm 2: Determining the state of a cell x at time t.

The renumbering of cells was defined earlier by distinguishing between two cell groups: the ones with indices $x \geq 0$, and those with indices $x < 0$. Observe that the renumbering of cells never changes the index of a cell from one group to the other one. Now, consider the case $x \geq 0$ first, and assume that there is function $\textsc{Successor}(x, t)$ which, for a cell index x at time t, returns the index z at which the given cell will be located at time $t + 1$, or ∞ in case the cell is deleted. Then, determining $\textsc{Predecessor}(x, t)$ is a simple matter of counting how many cells z get positioned to the left of x at time t. The case $x < 0$ is similar; the difference is that one has to count the cells to the right. Using the function $\text{sign}\colon \mathbb{Z} \to \mathbb{Z}$ with $\text{sign}(0) = 0$, and $\text{sign}(x) = |x|/x$ otherwise, allows one to write down compact code working for both cases as shown in Algorithm 3.

```
⟨ Compute cell index z such that the states of cells (z − 1, z, z + 1) at time t − 1
  determine the state of cell x at time t; additionally return a flag new
  indicating whether the cell now at position x is "a new one" or not, i. e.
  whether ε of σ should be used to compute its state. ⟩
function (cell z, bool new) ← Predecessor(cell x, time t) is
    z ← 0
    while true do
        if Successor(z, t − 1) = ∞ ∨ |Successor(z, t − 1)| < |x| then
            z ← z + sign(x)
        end
    end
    if x = Successor(z, t − 1) then
        return (z, false)
    else
        ⟨ this can only happen if cell x was generated from t − 1 to t ⟩
        return (z, true)
    end
end
```

Algorithm 3: Counting how many cells existing at time $t - 1$ get indices "closer to 0" than cell z.

Finally, SUCCESSOR(x,t) has to be described; see algorithm 4 on the next page. Roughly speaking, this is again a simple counting procedure: For every cell "between 0 and up to, but excluding x itself", one simply counts whether it gives rise to 0, 1 or 2 cells at time $t+1$. After the trivial cases (cell x is deleted, or $x=0$) have been handled, the consequences of the actions of cell 0 are taken into account. Before the **while** loop is started, i is the number of the first cell $\neq 0$ in the direction of cell x, and z is already the correct value to be returned in case none of the cells $i,\ldots,x-\mathrm{sign}(x)$ lead to states $\neq \otimes$. Subsequently, z is increased/decreased (depending on which side of 0 is considered) when required.

```
function cell z ← SUCCESSOR(cell x, time t) is
    if σ(STATE(x−1,t), STATE(x,t), STATE(x+1,t)) = ⊗ then return ∞ end

    ⟨ the rest is only executed if the cell isn't deleted ⟩
    if x = 0 then return 0 end

    ⟨ the rest is only executed if x > 0 or x < 0 ⟩
    if x > 0 then
        i ← 1
        if σ(STATE(−1,t), STATE(0,t), STATE(1,t)) ≠ ⊗ then
            z ← 1
        else
            z ← 0
        end
    end
    if x < 0 then
        i ← −1
        if ε(STATE(−1,t), STATE(0,t), STATE(1,t)) ≠ ⊗ then
            z ← −2
        else
            z ← −1
        end
    end
    ⟨ Now, consider all cells from i up to, but excluding x, ⟩
    ⟨    and count how many non-⊗ cells they amount to ⟩
    while i ≠ x do
        q_{−1} ← STATE(i − 1, t)
        q_0 ← STATE(i, t)
        q_1 ← STATE(i + 1, t)
        if σ(q_{−1}, q_0, q_1) ≠ ⊗ then z ← z + sign(x) end
        if ε(q_{−1}, q_0, q_1) ≠ ⊗ then z ← z + sign(x) end
        i ← i + sign(x)
    end
    return z
end
```

Algorithm 4: Counting how many cells are generated from t to $t+1$ which are "closer to 0" than cell x.

Finally, the space complexity of the above algorithm has to be determined depending on the length $n = |w|$ of the input. Let $t(n)$ be the time complexity of the SXCA. First of all, one observes that, inside FINALSTATE, there will be calls to STATE for all t from 0 to $t(n)$. This requires a counter which uses $\log t(n)$ space.

Secondly, the functions STATE, PREDECESSOR and SUCCESSOR all have a parameter time t, and they call themselves recursively in such a way that at least on every second recursion the time parameter is decreased by 1. Hence the number of recursion levels is bounded by $2t(n)$.

It remains to determine the space needed on each recursion level in any of the three functions. In all of them,q a constant number of cell indices have to be stored. Since the number of non-quiescent cells can only grow from k to $2(k+2)$ in a single global step, from an initial configuration with $n \geq 2$ cellscan, after $t(n)$ steps, give rise to at most $n \cdot 4^{t(n)}$ non-quiescent cells. $O(t(n))$ bits suffice for storing the index of any cell in this resulting configuration.

Therefore the total space complexity of the above algorithm is bounded by $O((t(n)^2)$.

5 Conclusion and Outlook

We have proven that space complexity of Turing machines and time complexity of shrinking and expanding cellular automata are polynomially related. In the construction we made use of both shrinking and expanding, but only in a somewhat restricted way: During a first part of the computation the CA was only expanding, and during the second part only shrinking.

It is still an open problem to exactly characterize the set of problems which, for example, can be decided by polynomial time cellular automata which can only expand, but not shrink. The results may depend on the precise definition of acceptance for XCA.

References

1. Chandra, A. K., Kozen, D. C., Stockmeyer, L. J.: Alternation. Journal of the ACM **28** (1981) 114–133
2. Fellah, A, Yu, S.: Iterative tree automata, alternating Turing machines, and uniform Boolean circuits: relationships and characterization. In: Berghel, H. et al. (eds.): SAC '92 Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing: technological challenges of the 1990's. ACM, New York (1992) 1159–1166
3. Kutrib, M., Malcher, A., Wendlandt, M.: Shrinking One-Way Cellular Automata. In: Kari, J. (ed.): Cellular Automata and Discrete Complex Systems. Lecture Notes in Computer Science, Vol. 9099. Springer-Verlag, Berlin Heidelberg New York (2015) 141–154
4. Modanese, A. C. V.: Shrinking and Expanding Cellular Automata. Bachelor Thesis. Karlsruhe Institute of Technology (2016), to be submitted
5. Rosenfeld, A., Wu, A, Dubitzki, T.: Fast Language Acceptance by Shrinking Cellular Automata. Information Sciences **30** (1983) 47–53
6. van Emde Boas, P.: Machine Models and Simulations. In: van Leeuwen, J. (ed.): Handbook of Theoretical Computer Science. Elsevier (1990) 1–66