Two methods for finding cellular automata that perform simple computations

M. Bustamante, C. Guerra, J. Veerman*

Pontificia Universidad Católica del Perú

Abstract

We have used two different methods to find cellular automata (CA) that perform some simple computations: by specifically constructing them to do a certain task and by doing searches in CA rule space. For the first case (specific construction), we start by building a CA that computes some arithmetic or logic function in a serial way, that is, a particle first moves to read one of the inputs and then applies it to the other one. Even though the rule we find ends up with a lot of states, depending on the type of operation we want it to perform, and it is not simple anymore, we find some types of behavior that are then used to construct CA rules that perform the same tasks but in a parallel way, therefore requiring a fewer number of states and steps. The results obtained in the first method are then used as restrictions or assumptions for the second method (searches in a rule space) in order to narrow down the rule space to perform a more directed search. We show some results for arithmetic functions, bitwise logic operations and other simple computations.

Specific construction

The initial condition is encoded in the format *Input1-Particle-Input2*, where both pieces of input (written in unary) are static while the particle(s) is moving around performing the computation. Specific construction is a step-by-step process where new states (or colors) are added as needed, the final goal being to halt after the calculation is complete. However, careful attention must be paid in order to avoid colors from intervening in tasks they were not designed to take part in.

In general, there are two approaches to performing a calculation. In serial constructions, a single particle carries out different tasks while changing its state. This makes the process slow, but easier to understand and program. In parallel constructions, the computation is carried out by more than one particle simultaneosly, making the process faster, but harder to visualize and therefore to program. A comparison between these two approaches is shown in Figure 1.

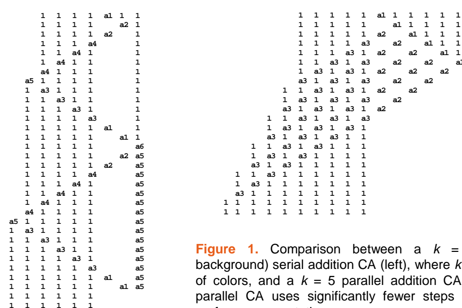

Figure 1. Comparison between a $k = 8$ (including background) serial addition CA (left), where k is the number of colors, and a $k = 5$ parallel addition CA (above). The parallel CA uses significantly fewer steps and colors to perform a computation.

Using these same two approaches, we have constructed CA rules that perform subtraction, division, multiplication (Figure 4) and exponentiation (see Figure 7 at the far right). Furthermore, we have explored similar constructions for bitwise logic operations (AND, OR, NOT and NAND, which is shown in Figure 3) as well as for other simple computations, such as bubble sorting (Figure 2). As a generalization, several of the arithmetic CA rules were merged into one multifunction CA which can perform different arithmetic calculations depending on the processing particle that is encoded in the initial condition.

As we move from addition to multiplication to exponentiation (to tetration, etc.), the difficulty in programming the CA rules increases. An alternative approach to directly programming the more complex arithmetic functions is to build them on simpler ones. For instance, a rule for multiplication can be directly constructed (see Figure 4) or it can be built as a series of embedded CA additions. Similarly, the exponentiation rule shown in Figure 7 is built as a series of multiplications, which themselves are constructed as embedded additions.

We have identified three types of basic behaviors in the arithmetic CA. Addition and subtraction rules are based on creating output and annihilating input. For example, in an addition operation, input at the right is annihilated while corresponding output is created next to the leftmost input. For higher arithmetic functions, a third type of behavior –memory– appears. This is due to the fact that in these types of calculations one input has to be used more than once before arriving at the result and some sort of tracking mechanism has to be used.

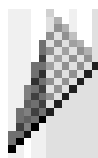

Figure 3. Bitwise NAND CA. Input at the right is reversed.

◄ Figure 2. Bubblesort CA with $k = 15$. The initial condition consists of three unsorted numbers (5, 3, 2) coded in unary. The output array is in ascending order.

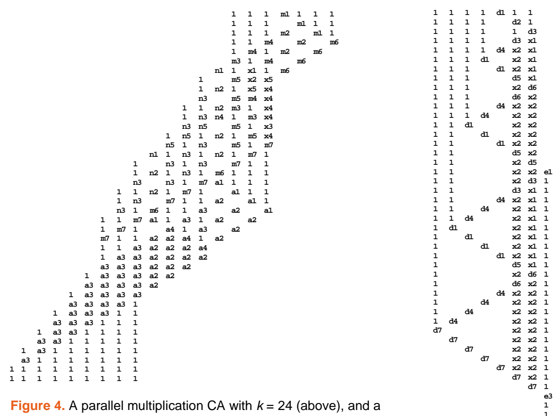

Figure 4. A parallel multiplication CA with $k = 24$ (above), and a serial division CA with $k = 14$.

Searches in CA rule space

The difficulty of performing searches in CA rule space is its rapid growth with the number of states. Based on what we have learned from doing specific constructions, some constraints and filters can be applied in order to reduce the search space. From the parallel constructs, a maximum number of steps can be taken into account if what we expect to find are time-efficient rules. An estimate of the number of colors needed to perform a certain computation can also be inferred from them. Finally, some basic interactions between particles which were observed in the specific constructions can be assumed to hold for the rules that we look for, therefore narrowing down the search by several orders of magnitude. Some examples of rules found for subtraction are shown in Figure 5.

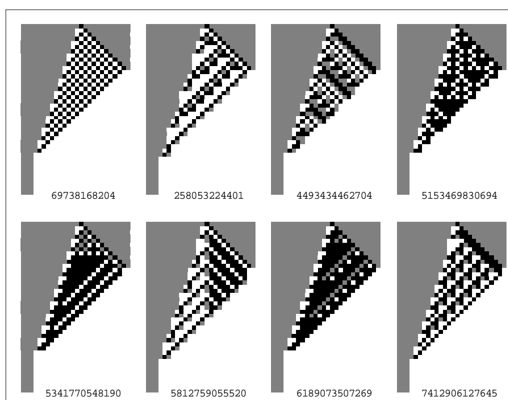

Figure 5. Some results of a search for subtraction rules in a CA space with $k = 3$. Rule numbers are shown beneath each graphic. Left input is 14; right input is 11.

Further work

Work is currently being done on designing algorithms to search in very large CA rule spaces by different methods.

A different line of research is the study of substitution systems that perform simple computations, with some results already found. Figure 6 shows some examples of sets of transformation rules that perform simple computations with two kind of sequential substitution systems (SSS): rulewise SSS (RW3S, changes the rule until a rule applies) and substringwise SSS (SW3S, changes the substring until a rule applies). These rules were found by perfroming exhaustive searches in small sets of all possible string transformation rules.

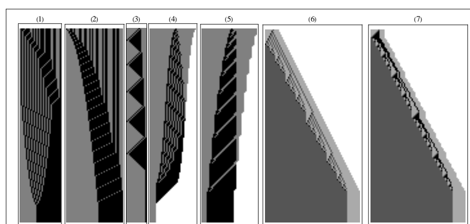

Figure 6. (1) SW3S for density classification, {BBA → BAB,BA → AB}. (2) RW3S for density classification, {BA → AB,BBA → ABB}. (3) RW3S for adding integers, {BAB → AAB,ABA → BAB, ABB → AAB}. (4) SW3S for subtracting integers, {BBA → BAB,ABA → BB,AB → A}. (5) RW3S for subtracting integers, {ABA → BBB,BAA → AAB,BA → A}. (6) SW3S for adding integers,{C → BB, AB → CA}. (7) RW3S for adding integers, {C → BA,AB → BC}.

Figure 7. Evolution of a CA with $k = 64$ that performs exponentiation x^y serially and halts ► when the result is reached. x is the left input and y, the right input. The case is shown for $x = 2$ and $y = 3$. Particles with similar behavior (background, input/output, memory, walls and moving particles) have been assigned the same color for clarity.

* To whom correspondence should be addressed: *jveerman@fisica.pucp.edu.pe*