The packing problem : A divide and conquer algorithm on cellular automata

Nicolas Bacquey

The packing problem: A divide and conquer algorithm on cellular automata

Nicolas Bacquey
Groupe de Recherche en Informatique, Image, Automatique et Instrumentation de Caen
Université de Caen Basse-Normandie, CNRS, Campus Côte de Nacre
Boulevard du Maréchal Juin, 14 000 Caen, France
20606898@etu.unicaen.fr

This paper introduces a new algorithm to deal with the packing problem on cellular automata of dimension 2 in linear time complexity; which means $O(n)$ for an input square picture of size $n \times n$. We use a divide and conquer approach to achieve this linear time complexity. This algorithm preserves the black cells at each step of the algorithm, i.e. only moves them, which was the main design flaw in previous algorithms.

Introduction

About linear time complexity

In this paper, we exhibit a linear time algorithm on a 2-dimensional cellular automata, that means an algorithm whose execution time is $O(n)$ for any input square picture of side n (a $n \times n$ matrix). Linear time complexity class is the minimal robust complexity class currently known on 2-dimensional cellular automata. Real time complexity class could also be a good candidate, but it strongly depends on the neighbourhood you choose for the automaton, whereas linear time complexity class is the same regardless the neighbourhood you choose, assuming it is a complete one (see [7] and [9]). There already are some basic linear time algorithms in the literature, e.g. shrinking algorithms for counting connected components (see [5] and [1]). See also [2] for general questions on cellular automata.

The packing problem

According to S. R. Kosaraju in [3], the packing problem can be defined as follows: We are given an input square picture of size $n \times n$ composed of black and white cells (i.e. the input state set is {B,W}) and we want to transform it into another picture composed of the same number of black cells. In this new picture, we want the cells to be packed into solid rows at the top of the pattern (the last row that contains black cells has to be left-justified). Some slightly different versions of the packing problem have also been studied in the context of VLSI, for instance in [8] and [4]

Submitted to:
AUTOMATA & JAC 2012

The packing problem has many applications such as image processing, or the recognition of the majority language, which is the language of pictures that contains more black cells than white cells. But beyond these naive applications, the packing problem can be used to achieve a key point of many algorithms: gathering information. Indeed, let us consider an algorithm on cellular automata that only processes the information located on a particular subset of cells (let us call these black cells) and ignores the other ones (let us call these white cells). In that case, pre-processing the computation area with the packing algorithm could highly increase its performances, provided only that the information that can be stored on black cells is preserved; we will see that our algorithm fulfils this requirement.

Two previous algorithms for the packing problem were presented by S. R. Kosaraju in [3], but none of them fulfils our requirements. Indeed, the first algorithm encodes the number of black cells into a binary number, thus losing all the information possibly stored in these cells. Then this number is compared to the sizes of different regions of the picture, which are filled with black or left white. By using clever operations, linear time complexity is achieved.

The second one is much more simple: it only moves the black cells according to a very simple set of rules. On any picture, draw a right to left arrow along each odd numbered row, and a left to right arrow along each even numbered row, then the cells move as follows: A black cell moves up a column if the cell immediately above is white. Otherwise, it moves along the row in the direction of the arrow. If a white cell can get a black cell from either a column and a row, then the column has preference. These rules are summarized on Fig 1, in which question marks can either be a black cell or a white cell. The author stated erroneously that this algorithm only takes linear time, but one can easily see that it is quadratic in the worst case, for example by processing a square picture whose left half is black and whose right half is white.

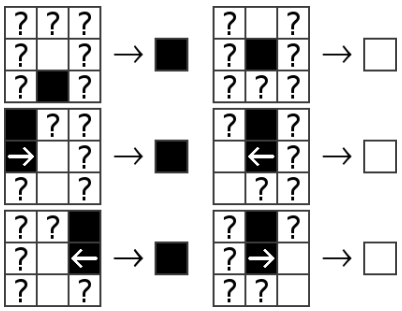

Figure 1: Kosaraju's packing rules

We will try to improve this previous algorithm, by using a divide and conquer approach. More precisely, we will establish two interesting points:

1. In a general matter, how is it possible to implement divide and conquer algorithms on cellular automata, for packing or other tasks ? More precisely, we will present the division of the initial space into smaller spaces and the proper reunification of those subspaces.
2. How, given our square $n \times n$ picture divided into four square sub-pictures of side $\left\lfloor \frac{n}{2} \right\rfloor$, do we process them to obtain a single packed picture ? Note that if we can achieve such a single merging in linear time, we can also solve the packing problem in linear time, due to the parallel execution of divide and conquer on cellular automata.

1 Divide and conquer on cellular automata

The general method of divide and conquer is an efficient way to solve many sequential problems. Moreover, it seems to be even more efficient on cellular automata, because the different sub-problems can be processed at the same time, thanks to their parallel structure. The main difficulty we encounter when we try to develop divide and conquer algorithms on cellular automata is the fact that we often need a global overview of the problem to properly divide it into sub-problems, and merge them together once they are solved. Such a global overview can be difficult to achieve on such a local model as cellular automata.

It is interesting to see that we only need to solve the problem of dividing the computation area for one dimension automata. Indeed, once you have an algorithm able to divide and merge areas in one dimension, you just have to apply it in parallel on each line of each dimension of your computation area to get a divide and merge algorithm for every dimension.

In the following sections, we will see how to perform this divide and merge process by adapting a naive one dimension firing squad algorithm. First, we will assume for simplicity that the width of the computation area is a power of 2. All along these sections, we will refer to "borders" as "the space between two cells". We can store information about these borders on any adjacent cell. Moreover, for geometric elegance purposes, we will design our diagrams as if borders could send signals. Note that signal information is actually stored in adjacent cells.

Dividing the computation area

Several algorithms already exist for dividing a one dimension computation area. We will use the following, which is commonly used in a naive firing squad algorithm and was first introduced in [6] (see also Fig 2):

1. From the leftmost cell, send two signals with 1/3 and 1 speeds, respectively. The fastest (1-speed) signal bounces on the borders.
2. When the two signals intersect, send a vertical signal that will act as a border for future signals, and send two new signals on each side of the border. We will also mark this border with 'L' if the 1-speed signal comes from the left, or 'R' if the 1-speed signal comes from the right.

- Repeat step 2 until each cell is isolated.

Merging the computation areas

What we need here is the word produced by the previous firing squad division algorithm. We assign the corresponding symbol (L/R) to each border and tag every odd-numbered border (see the circles on Fig 3). Then we repeat the following two phases :

1. Wait for the packing phase (or another processing) to be performed on each bordered sub-picture
2. Once this processing is over, delete every tagged border, then send a 1-speed signal in the direction corresponding to the symbol of the border, and a 1/3-speed signal in the opposite direction from the deleted border. These signals will bounce on borders. When four signals intersect on a border, tag this border and delete those four signals.

This will ensure us that the borders will be deleted in the reverse order that they were created, thus ensuring us the correctness of divide and conquer algorithms using these borders (see Fig 3).

Complexity of the divide and conquer algorithm

Assuming that each processing (e.g. packing) is performed in a time which is linear in the sub-picture side length, and that our initial picture has a size of $n \times n$, then there is a constant C such that the overall complexity is :

$$C \times n + C \times \frac{n}{2} + C \times \frac{n}{4} + \cdots + C = O(n)$$

General case: divide and conquer on an area of any side length

We have two ways of adapting this algorithm to the general case, i.e. when the side of the computation area is $2k+1$:

- Assign the extra square to an arbitrary sub-area (then the areas side lengths will be k and $k+1$). In that case, we will have to slightly modify the tagging process and the initially tagged borders in the merging phase.
- Make a one square thick border instead of a thin one, in such a way that the two remaining areas have the same side length k. We will also have to modify the overlying merging algorithm to consider the squares included in the border, if any.

The analysis of the time complexity is easily adapted to this general case.

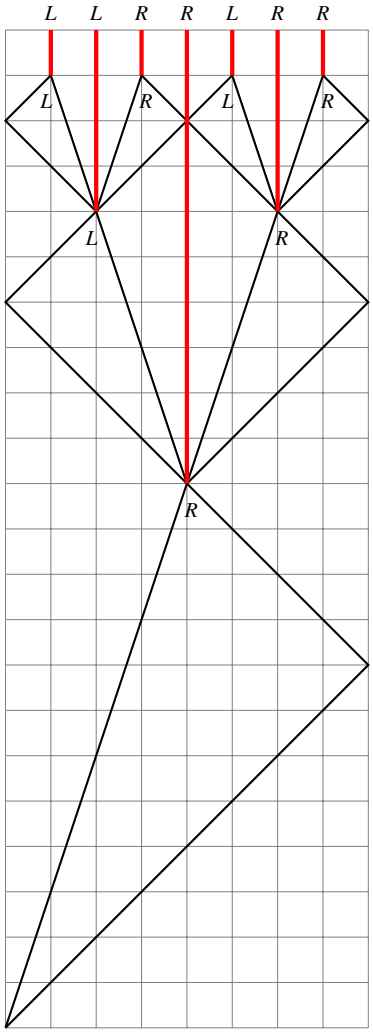

Figure 2: The firing squad division algorithm

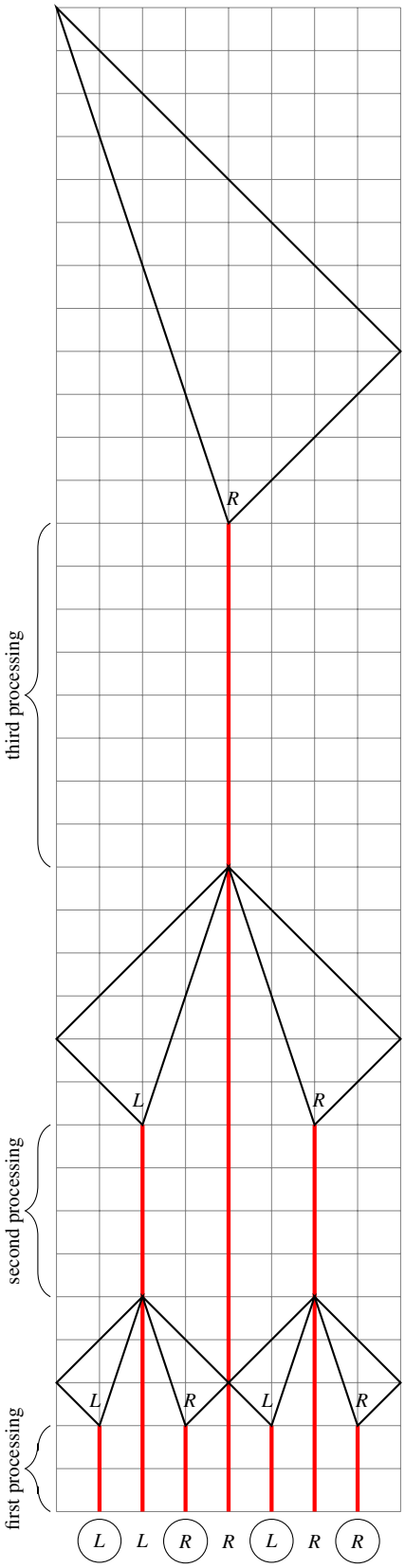

Figure 3: The merging algorithm

2 The packing algorithm

Before going any further, we need to assign a direction to each cell of the picture. Each cell of an odd numbered row will have a left to right arrow. Conversely, each cell of an even numbered row will have a right to left arrow. Those arrows are permanently maintained in a specific layer. This can obviously be done in linear time.

All along the description of this algorithm, we will refer to "pushing" as the action to vertically pack the black cells (Actually, it is the one-dimensional packing problem).

We will illustrate how to merge 2×2 packed squares of size $k \times k$ into a packed square of size $2k \times 2k$. The initial configuration can be seen on Fig 4.

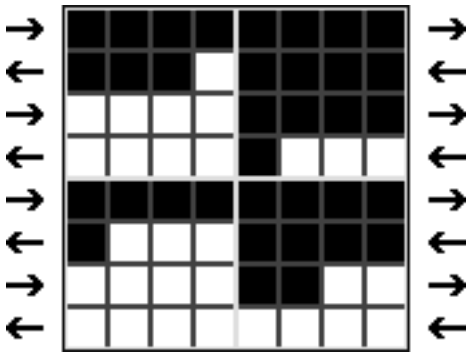

Figure 4: Four square packed pictures

Step 1

The first thing we have to do is push the filled rows of the squares to be merged at the top of the picture, and the incomplete ones at the bottom. These rows can be discriminated, for instance, by sending a signal to the left from the vertical separation lines on the original picture (the vertical light lines on Fig 4). We use two layers for this packing, layer 1 for the processing of the full rows and layer 2 for the processing of the incomplete ones. When the packing is achieved, the picture is composed of at most 3 connected components (see Fig 5). Clearly, there are at most two incomplete rows in layer 2.

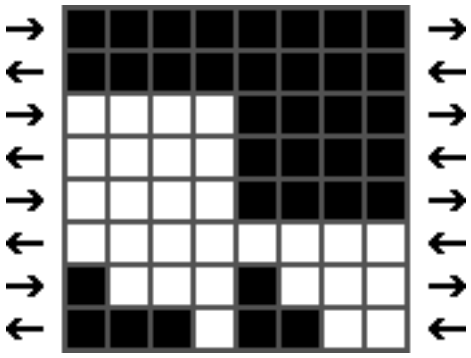

Figure 5: After the first pushing

Step 2

We now want to balance the former full rows such that there will be at most one incomplete row at the top of the picture. In order to do so, we move every cell of layer 1 according to the direction of their row, until each cell is blocked (see Fig 6). Then we push the rows up to achieve our balancing (see Fig 7). Note that there is now at most one incomplete row in layer 1.

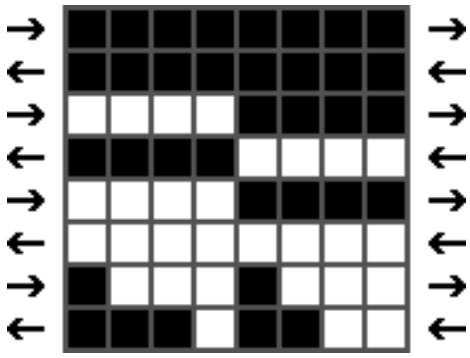

Figure 6: First moving...

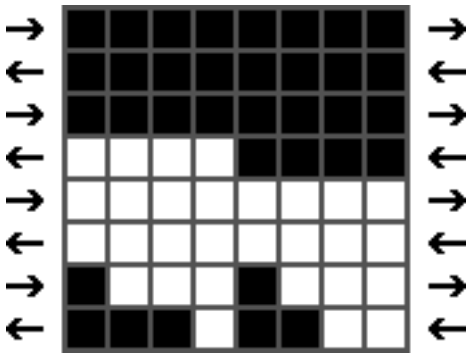

Figure 7: ... Then pushing up

At last, we merge the two layers and we push the incomplete rows up, to obtain a picture such as the one displayed on Fig 8. Now, we only have at most three incomplete rows on the picture (one from layer 1, and two from layer 2). Moreover, as we pushed them up, the number of black cells in those three rows is decreasing from up to down.

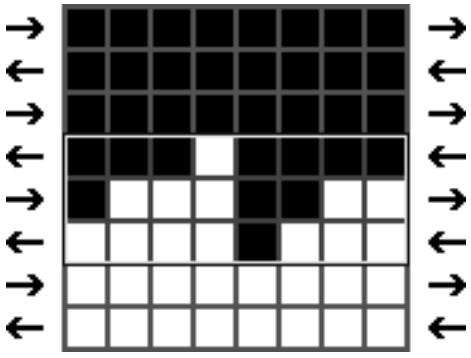

Figure 8: At the end of step 2

Step 3

We will now repeat the first half of Step 2, i.e. moving the cells according to their row and pushing them up, until we obtain a picture in which there is at most one incomplete row. We will prove that only two repetitions of these steps are enough to get such a picture.

Indeed, by construction the three incomplete rows are consecutive. Let us show that in this situation, each execution of these steps either fills the uppermost row, or empties the lowermost, which will conclude our proof :

Let n be the width of the rows, and n_1, n_2, n_3 be the number of black cells in the uppermost (row 1), middle (row 2) and lowermost row (row 3) respectively. We previously noted that $n_1 \geq n_2 \geq n_3$.

Let us now suppose that row 1 has not been filled by the first execution of the algorithm. Since row 1 and row 2 are packed in opposite directions, that means $n_1 + n_2 < n$, consequently we must have $n_2 < \frac{n}{2}$. Since $n_2 \geq n_3$ we also have $n_3 < \frac{n}{2}$, which means there is enough room in row 2 for all the black cells of row 3 to go up and therefore row three is empty after the first execution of the algorithm, QED.

Note that, in our example, a single repetition is enough to both fill the uppermost row *and* empty the lowermost (see Fig 9 and 10).

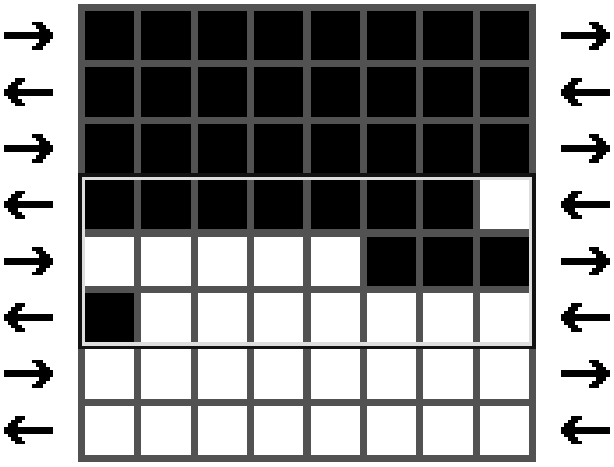

Figure 9: First moving...

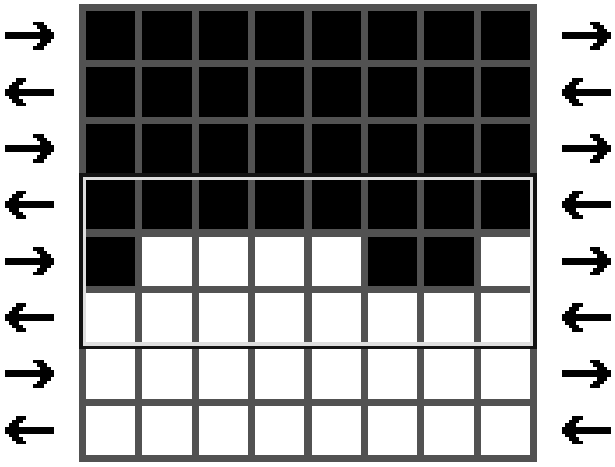

Figure 10: ... Then pushing up

We only have to pack the last incomplete row to the left to obtain a fully packed picture and conclude the algorithm (see Fig 11).

Complexity analysis

Each operation detailed in the previous part can be done in a time which is linear in the side length of the square containing the sub-problem being processed. Therefore, we can merge four squared packed pictures into a single one in a time linear in the side length of the resulting picture, thus implying that we have solved the packing problem in linear time.

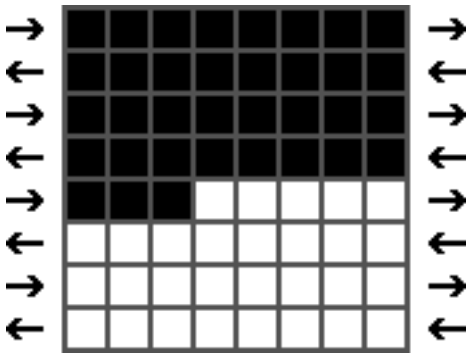

Figure 11: The final picture

Final remarks

Notice that our algorithm can easily be generalized for rectangular pictures of size $n \times m$, still in linear time, which means $O(n+m)$. In that case, we will have to deal with possibly odd-sized computation areas, and we will have to implement one of the two alternatives presented in the previous section. If we choose the second one (i.e. the thick border), the number of extra cells to deal with will be linear in the size of the sub-area being processed, and we can reasonably state that we can process them in linear time.

We are also convinced that it can be generalized to any d-dimensional cellular automata with any d-dimensional input picture of size $n_1 \times n_2 \times \cdots \times n_d$, still in linear time, i.e. $O(n_1 + n_2 + \cdots + n_d)$.

One may have noted that the linear time algorithm presented in [3] can be extended to preserve the information stored on black cells. If q is the number of states of the initial automaton, such an extension would need $2^{O(q)}$ states to be functional. Our algorithm only needs $k \times q$ states, where k is an intrinsic constant of the algorithm.

References

[1] W. T. Beyer (1969): *RECOGNITION OF TOPOLOGICAL INVARIANTS BY ITERATIVE ARRAYS*. Technical Report, Cambridge, MA, USA.

[2] J. Kari (2012): *Basic Concepts of Cellular Automata*, pp. 3–24. 1, Springer.

[3] S. R. Kosaraju (1974): *On Some Open Problems in the Theory of Cellular Automata*. *IEEE Transactions on Computers* C-23(6), pp. 561–565. Available at `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1672588`.

[4] H. Lang, M. Schimmler, H. Schmeck & H. Schröder (1985): *Systolic Sorting on a Mesh-Connected Network*. *IEEE Trans. Computers* 34(7), pp. 652–658. Available at `http://doi.ieeecomputersociety.org/10.1109/TC.1985.1676603`.

[5] S. Levialdi (1972): *On shrinking binary picture patterns*. *Commun. ACM* 15(1), pp. 7–10, doi:10.1145/361237.361240. Available at `http://doi.acm.org/10.1145/361237.361240`.

[6] J. McCarthy & M. Minsky (1964): *Sequential Machines: Selected Papers*, pp. 213–214. Addison-Wesley Longman Ltd., Essex, UK, UK.

[7] V. Poupet (2007): *Cellular Automata: Real-Time Equivalence between One-Dimensional Neighborhoods*. *Theory Comput. Syst.* 40(4), pp. 409–421. Available at `http://dx.doi.org/10.1007/s00224-006-1315-x`.

[8] C. Schnorr & A. Shamir (1986): *An Optimal Sorting Algorithm for Mesh Connected Computers*. In: *STOC*, pp. 255–263. Available at `http://doi.acm.org/10.1145/12130.12156`.

[9] V. Terrier (1999): *Two-Dimensional Cellular Automata Recognizer*. *Theor. Comput. Sci.* 218(2), pp. 325–346. Available at `http://dx.doi.org/10.1016/S0304-3975(98)00329-6`.