

# Cellular Automata with Cell Clustering

Lynette van Zijl\* and Eugene Smal

Computer Science, Stellenbosch University,  
P/Bag X1, Matieland, 7601, South Africa  
<http://www.cs.sun.ac.za/~esmal>

**Abstract.** We consider the modelling of a particular layout optimisation problem with cellular automata, namely, the LEGO construction problem. We show that this problem can be modelled easily with cellular automata, provided that cells are considered as clusters which can merge or split during each time step of the evolution of the cellular automaton. The LEGO construction problem has previously been solved with optimisation techniques based on simulated annealing and with a beam search approach, but we show that the use of cellular automata gives comparable results in general, and improves the results in many respects.

## 1 Introduction

Cellular automata (CA) and their many variations typically have a constant grid size, irrespective of the problem being modelled. That is, in modelling a given problem using CA, the starting point is a grid of a fixed size consisting of single cells, where each cell has a specific interpretation. Each cell is evaluated simultaneously in each time step. However, in problems where *clustering* plays a meaningful role in the information represented by a cell, each *cluster* has to be evaluated at each time step in an efficient manner, instead of the traditional cell-by-cell evaluation. The LEGO construction problem falls into this last category. The LEGO construction problem, in short, concerns the optimal layout of a set of LEGO bricks to represent a given shape. In this article we discuss the modelling of the LEGO construction problem using CA, with emphasis on the necessity for efficient cluster evaluation. To our knowledge, the idea of CA with cell clustering has not been investigated before.

The rest of this article is organised as follows: In Sect. 2 we define the necessary terminology. In Sect. 3 we show how to encode the LEGO construction problem with cellular automata, followed by a discussion of the results in Sect. 4. We point out possibilities for future work in Sect. 5, and conclude in Sect. 6.

## 2 Background

We assume that the reader is familiar with the theory of CA, as for example in [1]. We briefly recap on the definitions used in the rest of this article.

---

\* This research was supported by NRF grant number 2053436.

A cellular automaton  $C$  is a multidimensional array of automata  $c_i$ , where the individual automata  $c_i$  execute in parallel in fixed time steps. The individual automata  $c_i$  can reference the other automata by means of a *rule*. Typically, each  $c_i$  only references the automata in its immediate vicinity, called its neighbourhood.

If all the automata  $c_i$  are identical, then  $C$  is called a uniform CA. If all the automata  $c_i$  only have two possible states, then  $C$  is called a binary CA. We restrict ourselves to two-dimensional (2D) uniform binary CA in this article. When a given CA is two-dimensional, it forms a two-dimensional grid of cells, where each cell contains one of the automata  $c_i$ . In an  $n \times n$  grid we assume that both the rows and columns are numbered from 0 to  $n - 1$ .

As an example, consider the CA  $C$  with a  $3 \times 3$  grid and with the rule  $c_{ij}(t + 1) = c_{i-1,j}(t) \oplus c_{i+1,j}(t)$ . Suppose the rows of  $C$  are initialised with the values 000, 010 and 111 at time step  $t = 0$  (see Fig. 1). Then the value of  $c_{11}(t = 1) = 0 \oplus 0 = 0$  and  $c_{21}(t = 1) = 1 \oplus 1 = 0$ . Note that we assume null boundaries, so that if  $i = 0$ , then  $i - 1$  is simply ignored in the formula. Likewise, if  $i = n - 1$ , then  $i + 1$  is ignored. In the example above,  $c_{22}(t = 1) = c_{12}(t = 0) \oplus c_{32}(t = 0) = c_{12}(t = 0) = 1$ .

$t = 0$	0	0	0	$t = 1$	0	0	0
	0	1	0		1	0	1
	1	1	1		1	0	1

**Fig. 1.** An example CA.

We now define a so-called *cluster*, which we will need to intuitively describe the LEGO construction problem as a CA. Usually, a CA has a given number of cells, and the number of cells stays fixed throughout all of its time evolutions. In our particular model, however, it is easier to assume that cells may merge or split during time steps, so that the number of cells in the CA may vary between different time steps.

We first define the *adjacency* of cells to mean cells that touch on a joint border:

**Definition 1.** *Let  $C$  be a 2D CA. Two cells  $c_{ij}$  and  $c_{km}$  in  $C$  are adjacent if either  $|i - k| = 1$  or  $|j - m| = 1$ . If both  $|i - k| = 1$  and  $|j - m| = 1$ , then the cells are not adjacent.*

We now define a cluster as a set of adjacent cells:

**Definition 2.** *Let  $C$  be a 2D CA. A cluster  $\mathcal{B}$  in  $C$  is a set of cells from  $C$  such that any cell  $c_{ij}$  in  $\mathcal{B}$  is adjacent to at least one other cell  $c_{km}$  in  $\mathcal{B}$ .*

It is convenient to define disjointed clusters:

**Definition 3.** *Two clusters  $\mathcal{A}$  and  $\mathcal{B}$  are disjoint if there is no cell  $a_{ij}$  in  $\mathcal{A}$  such that  $a_{ij}$  is also in  $\mathcal{B}$ , and no cell  $b_{ij}$  in  $\mathcal{B}$  such that  $b_{ij}$  is in  $\mathcal{A}$ .*

The clusters in our modelling of the LEGO construction problem are typically disjoint, but it is not a necessary condition for our algorithms to function correctly.

We can also define adjacency of clusters:

**Definition 4.** *Two clusters  $\mathcal{A}$  and  $\mathcal{B}$  are adjacent if there exists at least one cell  $a_{ij}$  in  $\mathcal{A}$  and at least one cell  $b_{km}$  in  $\mathcal{B}$  such that  $a_{ij}$  is adjacent to  $b_{km}$ .*

00	01	02
10	11	12
20	21	22

**Fig. 2.** Cell clustering in a CA.

*Example 1.* In Fig. 2, cell  $c_{11}$  is adjacent to cell  $c_{12}$ , but  $c_{11}$  is not adjacent to  $c_{22}$ . Also, the set of cells  $\{c_{11}, c_{21}, c_{22}\}$  forms a cluster, but the set  $\{c_{12}, c_{21}\}$  does not. Lastly, the clusters  $\{c_{01}, c_{02}\}$  and  $\{c_{12}, c_{21}, c_{22}\}$  are adjacent, since cells  $c_{02}$  and  $c_{12}$  are adjacent. These two clusters are also disjoint.

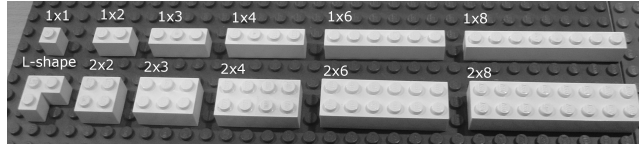
□

When using clusters to model the LEGO construction problem, we will further restrict the cluster to have only the forms that represent standard LEGO bricks. That is, the clusters can only have the rectangular shapes and sizes as shown in Fig. 3. Note how the bricks are identified by the number and rectangular arrangement of its studs. For example, a brick with two rows of three studs each is called a  $2 \times 3$  brick.

**Definition 5.** *A valid LEGO brick is one with dimensions as defined in Fig. 3.*

The reader should note that clusters are considered to be homogeneous entities, in the sense that all the cells forming the cluster will contain the same status information. However, clusters are typically of different sizes (for example, the cluster representing a  $2 \times 4$  LEGO brick may lie next to the cluster representing a  $1 \times 3$  LEGO brick). This means that we have to define the meaning of the neighbourhood of a cluster.

**Definition 6.** *The (Von Neumann) neighbourhood of a cluster  $\mathcal{A}$  is the collection of clusters adjacent to  $\mathcal{A}$ .*



**Fig. 3.** The list of standard LEGO bricks.

Note that the above definition implies that a cluster may not necessarily have four neighbours in its Von Neumann neighbourhood. Also, the number of Von Neumann neighbours may vary between different clusters. For example, in Fig. 2, suppose that there are four  $1 \times 1$  clusters, namely, are  $c_{00}$ ,  $c_{10}$ ,  $c_{20}$  and  $c_{11}$ . Also,  $\{c_{01}, c_{02}\}$  forms a cluster of size  $1 \times 2$  and  $\{c_{12}, c_{21}, c_{22}\}$  forms an L-shaped cluster. Then the Von Neumann neighbourhood of cluster  $c_{11}$  consists of three other clusters, namely, the cluster  $c_{10}$ , the cluster  $\{c_{01}, c_{02}\}$  and the cluster  $\{c_{12}, c_{21}, c_{22}\}$ .

In addition, a single cluster may have a different number of neighbours in different time steps. We also note that the definition of different types of neighbourhoods (such as Von Neumann or Moore) now simply defines the type of adjacency. This is indeed the case with CA without cell clustering as well, but the neighbourhoods based on a fixed size grid enforce a fixed number of neighbours for all cells through all time steps.

The next issue to consider is the merging of clusters.

**Definition 7.** Let  $\mathcal{A}$  and  $\mathcal{B}$  be two clusters in a binary 2D CA. Define a new cluster  $\mathcal{C}$  to contain every cell  $a_{ij}$  in  $\mathcal{A}$  and every cell  $b_{km}$  in  $\mathcal{B}$ , and no other cells. Then  $\mathcal{C}$  is said to be the merge of  $\mathcal{A}$  and  $\mathcal{B}$ .

Given a merge operation on two clusters, it is natural to consider the dual operation of splitting up a cluster into smaller parts. In our case, we simply consider the splitting of a cluster into its smallest constituent parts, namely, clusters of size  $1 \times 1$ :

**Definition 8.** Let  $\mathcal{A}$  be a cluster in a binary 2D CA, and let  $\mathcal{A}$  contain  $k$  distinct cells  $c_i$ , with  $1 \leq i \leq k$ . Then the operation  $\text{split}(\mathcal{A})$  creates  $k$  new clusters  $\mathcal{A}_i$ , such that  $c_i$  is the only cell in  $\mathcal{A}_i$ , for  $1 \leq i \leq k$ . All the clusters  $\mathcal{A}_i$  are thus disjoint.

We now discuss the LEGO construction problem in more detail, and show how to encode it using a uniform binary CA with clustering.

### 3 Encoding the LEGO construction problem as a CA

The LEGO construction problem in essence concerns the following issue: given a real-world three-dimensional (3D) object, find the most optimal way to build a LEGO sculpture of the object, given a specific set of standard LEGO bricks.

The traditional approach to solving the LEGO construction problem is to virtually cut the 3D object into horizontal two-dimensional (2D) layers (we assume that a digital representation of the real-world 3D object is given as input). The problem then reduces to a series of 2D solutions which can be joined together (under certain restrictions) to find the final 3D solution. We note that each horizontal layer will eventually form one layer of bricks in the final solution.

Given the digital representation of a 2D layer, the layer is then encoded as a grid of zeroes and ones, where the ones represent the solid parts of the original object in this layer. To solve the LEGO construction problem, we therefore have to find an optimal layout of LEGO bricks to cover the ones in each layer. It is assumed that there is an unlimited number of each type of brick of the standard LEGO bricks.

An optimal layout of LEGO bricks is defined by a number of factors [2], the most important of which are:

- the number of bricks used should be minimised;
- larger bricks should be used where possible, to allow for ease of building, better brick connectivity and to indirectly reduce the number of bricks used;
- bricks in consecutive layers should have alternate directionality to increase the stability of the final sculpture;
- a large part of each brick in the current layer should be covered by other bricks in the next layer; and
- a brick in the current layer should cover as many other bricks in the previous layer as possible.

The traditional way of solving the LEGO construction problem is to use combinatorial optimisation methods to find an optimal layout [3]. One of the first proposed solutions was based on genetic algorithms [4], while the current *de facto* solution makes use of a beam search to cover the search space [5]. As with all optimisation methods, some models can take excessively long to solve. Also, in some cases, sub-optimal solutions are found when the search tree is trimmed. Our CA approach is an attempt to alleviate exactly these problems.

We are now ready to encode the LEGO construction problem as a CA. Assume a two-dimensional grid of clusters, representing one 2D layer of the real-world object as described previously. Each  $1 \times 1$  cluster has the value zero or one, which indicates whether this is an area to be filled with a brick, or to be kept open. Additionally, we allow each cluster to keep separate status information as needed.

The time evolution of each layer of the model intuitively proceeds as follows: initially, each cluster of size  $1 \times 1$  which contains a 1 value, is assumed to represent a LEGO brick of size  $1 \times 1$ . Then, for each time step, there are two phases.

In the first phase, each cluster investigates every other cluster in its Von Neumann neighbourhood, to decide whether it is *possible* to merge with that neighbouring cluster. The status information for each cluster is then updated to indicate with which of the clusters in its neighbourhood the cluster is willing to merge. This operation happens in parallel and simultaneously for all clusters, similar to the time evolution of cells in a standard CA.

In the second phase, a sequential pass through the individual clusters investigates the status information of each cluster. This status information is used to decide which clusters will merge to form larger clusters. As the merging into clusters takes place in a sequential and progressive fashion, the order of the actual merges can be random, or front-to-back, or any other implementable way of choosing the order. This has a slight influence on the final layout.

We now discuss each phase in more detail.

### 3.1 Merging

In the first phase of the merging step, each cluster examines its neighbours to determine with which of the neighbours it can merge to form a valid new LEGO brick. We assume a Von Neumann neighbourhood, which uses the clusters directly adjacent to the current cluster. If there is more than one possible merge amongst the neighbours, the cluster selects the best possible merge by using a local cost function rule (described below)<sup>1</sup>. If there are no clusters with which the current cluster can merge, the cluster can split into smaller clusters (see Sect. 3.2). The status information of the cluster is now updated to indicate its best possible merge neighbour.

We use a local cost function as the rule to determine with which neighbour each cluster should be merged. The local cost function is defined as:

$$\begin{aligned} \text{Cost} = & W_{\text{perpend}} \times \text{perpend} \\ & + W_{\text{edge}} \times \text{edge} + W_{\text{uncovered}} \times \text{uncovered} \\ & + W_{\text{otherbricks}} \times \text{otherbricks} \end{aligned}$$

where the  $W$ 's are weight constants and

- the *perpend* variable corresponds to the directionality of the bricks in consecutive layers;
- the *edge* variable represents the number of the edges of each brick that coincide with edges of bricks from the previous layer;
- the *uncovered* variable describes how much of the area of each brick is not covered by bricks in the previous and following layers; and
- the *otherbricks* variable represents the number of bricks in the previous layer covered by this brick.

This cost function directly corresponds to the fitness function used by Petrovic [4], but without the *numbricks* and *neighbour* heuristics used in his function. These were removed since they have no effect when local optimisation is used. The *numbricks* heuristic is used to minimise the total number of bricks globally. Our method indirectly minimises the number of bricks, since a brick will always merge with a neighbouring brick if possible. Even though the *numbricks* heuristic

<sup>1</sup> If the local cost function evaluates to the same minimal value for more than one neighbour, a random choice is made amongst the minimal value neighbours.

was not included in the local cost function, we do still add it in our test results when calculating the total sculpture cost. This is useful in comparing the results to those achieved with other methods.

The cost function must be minimised to help ensure that the total cost of the LEGO sculpture is kept to a minimum while its stability is maximised. After each time step, the total cost for the layer is calculated. The best layer constructed throughout the time steps is used as the final building instruction for that layer.

In the second phase, the new set of (merged) clusters are constructed. The process involves two steps:

1. Each cluster is considered, and a potential new cluster is calculated by using the information from phase 1. If the new cluster forms a valid LEGO brick, the merge information for the cluster is completed.
2. For all potential new clusters that do not form valid bricks, the potential new cluster is traversed until a further traversal step would lead to an invalid brick. The traversed clusters are then flagged for merging into a new cluster, and the process is repeated until all the elements of the potential cluster have been traversed and flagged for merging into valid brick clusters.

Clearly, phase 2 is a sequential phase, where all clusters are calculated sequentially by visiting each cluster and using its best possible merge status information.

Once the new clusters have been calculated, all the clusters are merged simultaneously. The exact detail for the implementation of the merging can differ substantially, and this can have a noticeable effect on the performance of the method. In our case, we simply merge the clusters by starting with the first cluster in the new potential cluster, and merging it with its best possible merge neighbour. The resulting cluster is then merged again in a similar manner until either it cannot merge any further, or the whole cluster has been merged together. We note that this method of merging the clusters will not necessarily always deliver the optimal result. However, the local optimisation propagates to a satisfactory result in almost all cases, with a low execution time.

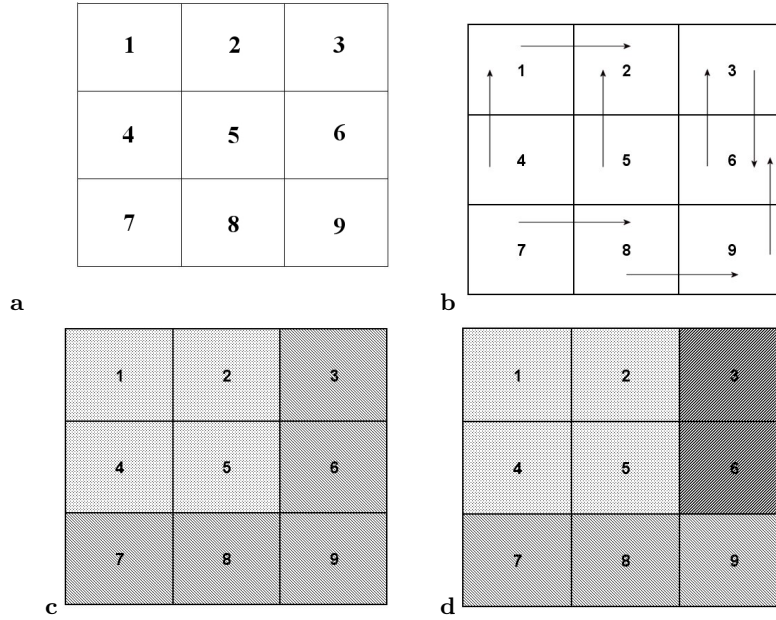
We give an example of the merging process below.

*Example 2.* Suppose that a  $3 \times 3$  grid, with all values consisting of ones, must be constructed from the standard LEGO brick set (see Fig. 4). Therefore, there are initially nine clusters in the grid, each representing a brick of size  $1 \times 1$ .

In the first phase each cluster determines with which of its neighbours it can merge to form a new valid brick. The cluster then sets its status information to indicate with which neighbouring brick it would best merge (see Fig. 4(b)). Here, each cluster indicates its best possible merge brick with an arrow.

In the second phase, all the potential new clusters are calculated. In this example we have two clusters (see Fig. 4(c)).

After the potential new clusters are calculated, they are traversed and merged together to form the larger clusters. The merging process in this case ends with



**Fig. 4.** (a) The 2D grid, (b) potential merge neighbours, (c) potential new clusters, (d) the final three clusters.

three clusters instead of two, since the second cluster cannot merge into one large valid brick (see Fig. 4(d)). The second cluster is therefore merged, until it cannot merge any further without resulting in an invalid brick. The remaining clusters are then traversed and merged together to form the third cluster.

In the next time step (see Fig. 5), each cluster will again determine with which brick its group of clusters should merge, using its local neighbourhood, to form a new valid brick. The best possible merge for the group is then selected. In this example cluster  $\{1, 2, 4, 5\}$  can only merge with cluster  $\{3, 6\}$  to form a valid larger brick. Cluster  $\{7, 8, 9\}$  can then not merge with any of the other two bricks. Therefore, there will only be one cluster to merge containing the merge of  $\{1, 2, 4, 5\}$  and  $\{3, 6\}$ . After the clusters have been merged, there are two bricks in the layout, which cannot be merged again into a larger valid brick.

□

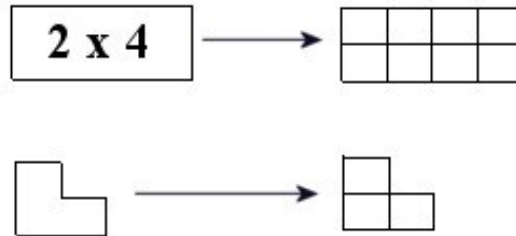
The counterpart to the merge operation is the *split* operation. As our merge operation can quickly reach the point where no more merging is possible, it is useful to split a given cluster so that the search for an optimal solution can continue.

1	2	3
4	5	6
7	8	9

**Fig. 5.** The resulting bricks after time step 2.

### 3.2 Splitting

Our splitting method simply splits a given cluster into  $1 \times 1$  clusters (see Fig. 6). It is in principle possible to use other splitting strategies (such as splitting into  $k$  smaller clusters). However, due to the large number of possible brick sizes in the LEGO construction problem, this would be computationally expensive with no immediate advantage over our complete dissolution of the cluster.



**Fig. 6.** Two bricks splitting into  $1 \times 1$  bricks

A cluster can potentially split if it cannot merge with any of its neighbouring clusters. We assign a splitting time delay to each cluster, which is the maximum number of time steps during which the cluster will unsuccessfully attempt possible merges. When the splitting time delay expires, the cluster will attempt to split. The time delay allows neighbouring clusters enough time to grow into clusters with which the current cluster can merge. If there were no time delay, clusters could split too early and larger clusters will fail to form. The time delay can be a random or fixed number of time steps for each cluster.

If a cluster was unable to merge within the time delay, it will attempt to split. For the LEGO construction problem, we assign a different splitting probability to every brick (basically, the larger a brick, the less its splitting probability).

When a cluster attempts to split, it generates a random number. If that number is less than its splitting probability, the cluster splits. Otherwise, the time delay of the cluster is reset and the cluster will again try to merge with neighbouring clusters.

In summary, our algorithm considers each layer separately. For each layer, it executes phase 1 and phase 2, and calculates the cost for the layer. Phase 1 finds the best merge neighbour for each cluster, and phase 2 calculates new clusters sequentially before merging all the clusters in parallel.

We implemented our CA-based method, as well as the beam search method of Winkler [5]. In Appendix A, we give some screenshots of our system, as well as some of the LEGO instructions generated by the system. In the next section, we analyse the results obtained by our CA with cell clustering method.

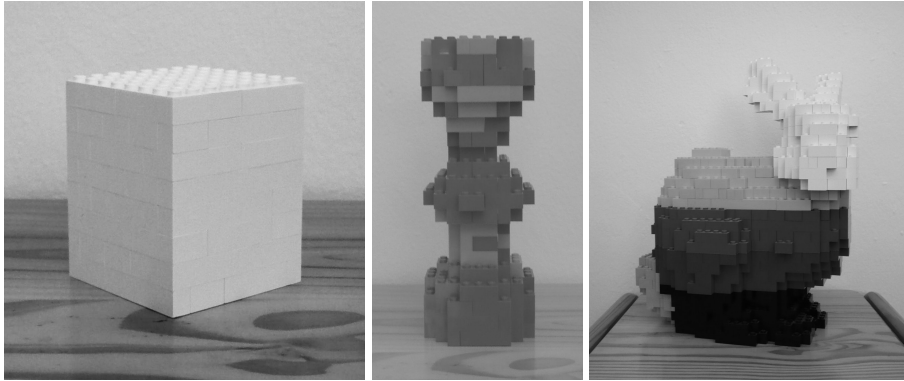
## 4 Results

We implemented our CA with cell clustering method, as well as the beam search method. We compare these two methods below, and discuss their respective advantages and disadvantages. We also point out some interesting issues that occurred in the implementations.

In comparing the CA with cell clustering against the beam search method, one is in essence comparing a local optimisation method against a global optimisation method. Having noted that, we had to define measures of quality for the comparison. We list these below:

- *number of bricks used in the final model*: in general, the fewer bricks used, the cheaper it is to produce the model. This was one of the criteria listed by the LEGO company in the original definition of the problem;
- *cost (as defined by the cost function)*: as the cost function contains all the parameters against which to optimise, a low cost function represents a ‘good’ model;
- *execution time of the implementation on a specific model*: the quicker a good layout can be reached, the better;
- *extending the solution to allow coloured sculptures*: the traditional solutions to the LEGO construction problem assume the input to be monochrome, which significantly limits the practical use of the implementation;
- *ease of building*: this is a ‘fuzzy’ measure, and we simply experimented by building a number of the same sculptures based on the instructions generated by the beam search method and the instruction generated by the CA method, respectively; and
- *ease of implementation of the solution*: here, we consider the complexity of the coding of the solution.

We set up a number of experiments to evaluate our measures of quality. For the experiments, we selected three different models (see Fig. 7). The first is a simple hollow cube with dimensions  $8 \times 8 \times 8$  unit bricks, which is representative of small geometric models without rounded surfaces. The second is a



**Fig. 7.** The LEGO sculptures.

chess pawn with dimensions  $10 \times 10 \times 20$  unit bricks, which is representative of small models with many rounded surfaces. The third is the well-known Stanford bunny with dimensions  $30 \times 20 \times 25$  unit bricks, which is representative of a larger sculpture with more complexity and finer detail. We selected weight constants<sup>2</sup>  $C_{numbricks} = 200$ ,  $C_{perpend} = -200$ ,  $C_{edge} = 300$ ,  $C_{uncovered} = 5000$ , and  $C_{otherbricks} = -200$ .

Reaching a good layout with the CA method depends on the number of time steps that the CA executes, whereas to reach a good layout with the beam search method depends on the width of its search tree (and how much it is pruned). These two parameters are not necessarily easily comparable. In addition, there is a random element to be taken into account in the CA method (in the merging phase), so that the values given for the CA method are averaged over a number of runs.

A summarised view of the results is given in Table 1. Here, the number of iterations for the CA was set at 1500, and the width of the search tree in the beam search method was set to 4000 (a wider search tree is possible, but will dramatically increase the execution time of the beam search).

Model	Number of bricks		Cost function value		Execution time (s)	
	CA	BS	CA	BS	CA	BS
Cube	42	43	20357	20158	75	8
Chess pawn	170	196	99889	106543	181	18
Bunny	1133	1436	630739	714223	280	313

**Table 1.** Experimental results for CA with cell clustering (CA) and beam search (BS).

<sup>2</sup> These values are similar to the values suggested by [4]. The large integer values prevent underflow errors in the evaluation of the cost function.

The first measure is the number of bricks in the final sculpture. From Table 1 it is easy to see that, for the more complicated sculpture (the chess pawn) and the larger sculpture (the bunny), the CA method uses less bricks than the beam search method. This is to be expected, as the CA method forces merging into larger bricks whenever possible so that fewer but larger bricks are used. The beam search, on the other hand, will evaluate the size of the bricks as part of its cost function, and the requirements for non-coincidental edges and alternate directionality then result in smaller bricks being chosen. It is possible to carefully choose the weight constants for the beam search to weigh the size of the bricks as highest priority. However, that leads to weaker sculptures, as the lack of alternate directionality and non-coincidental edges result in disconnected sculptures. Therefore, in the beam search, it often happens that two adjacent smaller bricks rather than a single larger brick appears. For example, we often found two adjacent  $1 \times 3$  bricks produced in the beam search sculptures, whereas the CA method would almost always force those into a single  $2 \times 3$  brick. This influences the ease with which a sculpture can be built, as it is usually more uncomfortable to connect many small bricks than to connect a single larger brick. In that sense, the CA method produces ‘better’ building instructions.

However, the above comments on the number and size of the bricks are mostly true for larger and more complex sculptures. In simple and small sculptures, the beam search can stabilise more quickly than the CA approach. For example, for the hollow cube, if the tree width is set at 10000, the number of bricks for the final sculpture is reduced to 32 within 19 seconds. The CA, on the other hand, takes 2000 time steps and 96 seconds to reach a brick count of 40.

Our second measure of quality is the value of the cost function. This represents the quality of the sculpture in total, including number of bricks, alternate directionality and non-coincidental edges between layers. Here, as can be seen in Table 1, the difference between the CA method and the beam search is comparable. It should be noted that it is possible to get a smaller cost value for the beam search, but at the expense of a much wider search tree and substantial increase in execution time. For example, if we increase the width of the tree to 40000 for the chess pawn, the value of the cost function falls to 95366, but the execution time increases to 275 seconds. Corresponding to the values in Table 1, the sculptures built from the CA generated instructions and the beam search generated instructions showed no perceivable difference in stability.

The last direct measure of quality of our method is the execution time. Although Table 1 indicates an order of magnitude difference in the execution times between the two methods, the values can be potentially misleading. We note that the execution of the CA is dependant on the number of time steps, which is a user parameter. There is no easy way to let the implementation decide its own point where no improvement is possible. For example, with the chess pawn model, a number of 192 bricks is reached within 14 seconds and just 100 iterations. To improve the number of bricks to 168 takes 2000 iterations and 246 seconds. The beam search, on the other hand, could reduce the number of bricks only to 180 in 275 seconds (with a tree width of 40000). In addition, the CA has a transitional

startup phase before stabilisation which shows clearly in the smaller models, but which is relatively shorter in the larger models. However, it is fair to say that for small and simple models, the beam search method executes faster than the CA method on a single processor machine. As we point out in the next section, we intend to parallelise both methods in the future.

There are two areas where the CA method shows distinct advantages over the beam search method. The first is ease of implementation – although the number of lines of code for each of our implementations is roughly comparable, it was much simpler to implement the CA (even with merging and splitting) than it was to implement a search tree with its associated pruning. The second area where the CA has a distinct advantage over beam search, is the extension of the implementation to cater for coloured sculptures. In the case of the CA method, this has a small effect on the merging phase, where merging of clusters is restricted to clusters of the same colour only. Our current implementation of the CA method already incorporates coloured models. The beam search method, however, is much more difficult to extend to coloured models. Every sculpture has a certain ‘thickness’ – that is, the depth of any one side of the sculpture. Traditionally, a thickness of four unit bricks works well to ensure enough stability without unnecessarily increasing the number of bricks used. When colour is incorporated, it is possible to make the outer brick the required colour and use the inner bricks (that are hidden from view) to ensure stability. In that way, even checkered patterned sculptures can be built without losing stability. However, in the beam search, this single extra requirement leads to an explosion in the size of the search tree, and in many cases tree pruning leads to disconnected sculptures.

Our last observation is a rather obvious one, but still important: the CA method uses significantly less memory than the beam search method.

In summary then, we conclude that the CA method shows results comparable to that of the beam search method, except for small and simple models. For complex models, the CA model typically uses fewer bricks. And lastly, the distinct advantages of the CA method is the ease of implementation and the fact that it is easily extendable to handle coloured sculptures.

## 5 Future Work

It is not immediately obvious that a theoretical investigation into CA with cell clustering would yield interesting results. However, we plan to consider the concept of clustering in more detail, even if only for modelling some additional problems.

As far as our implementation is concerned, we plan to parallelise the code and run experiments on a Beowulf cluster to consider the extent of the speedup over a single processor machine. As the parallel implementation of game trees is well-known, we do not expect either method to improve upon the other by using parallelisation.

We have previously implemented a 3D CA with clustering, in order to do the layout for all layers in the sculpture simultaneously. The results were poorer than expected, but we intend to revisit this aspect in more detail.

Although not part of the CA solution to our problem, we wish to improve on the manual effort necessary to build a virtual 3D model for input to the system. Currently, the user must construct a 3D model by hand using a 3D modelling package. However, ideally, it should be possible to input a series of synchronised photographs, so that the system can build its own 3D model. We have completed an initial version of such a system based on the shape from silhouette technique [6], but there are still some issues to be resolved.

## 6 Conclusion

The paper presented a new approach to solving the LEGO construction problem, based on a CA with cell clustering.

We showed that a CA incorporating cell clustering is a simple generalisation of traditional CAs. This generalisation allows us to easily model real world clustering problems with CA.

We discussed the practical results from our implementation of the LEGO construction problem, and pointed out the differences between the traditional beam search approach and the CA approach. We pointed out the various circumstances in which one approach would perform better than the other.

Finally, we stated our planned extensions to our implementation, and noted the possibility for parallelising the implementation on multiple processors, as well as the need for an easier method of input creation.

## References

1. Wolfram, S.: Cellular Automata and Complexity. Perseus Books Group (2002)
2. Na, S.: Optimization for layout problem. PhD thesis, University of Southern Denmark (2002) <http://www.idi.ntnu.no/grupper/ai/eval/brick/>.
3. Gower, R., Heydtmann, A., Petersen, H.: LEGO: Automated model construction. 32nd European Study Group with Industry (1998) 81–94 <http://www.idi.ntnu.no/grupper/ai/eval/brick/>.
4. Petrovic, P.: Solving the LEGO brick layout problem using evolutionary algorithms. Technical report, The Maersk Mc-Kinney Moller Institute for Production Technology, Norwegian University of Science and Technology (2001) <http://www.idi.ntnu.no/grupper/ai/eval/brick/>.
5. Winkler, D.: Automated brick layout. BrickFest (2005) <http://www.brickshelf.com/gallery/happyfrosh/BrickFest2005/>.
6. Cheung, K., Baker, S., Kanade, T.: Visual hull alignment and refinement across time: a 3D reconstruction algorithm combining shape-from-silhouette with stereo. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. (June 2003) 375–382

## Appendix A

This appendix contains some screenshots and results generated from our system.

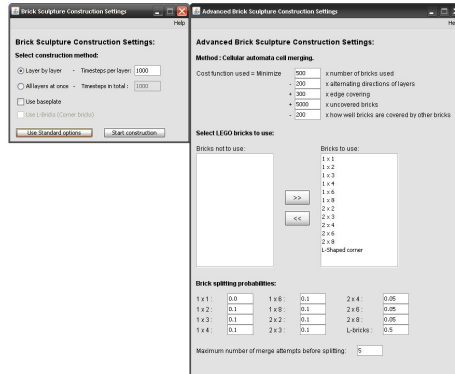


Fig. 8. Some parameter settings for the system.

### Layout Layer 5:

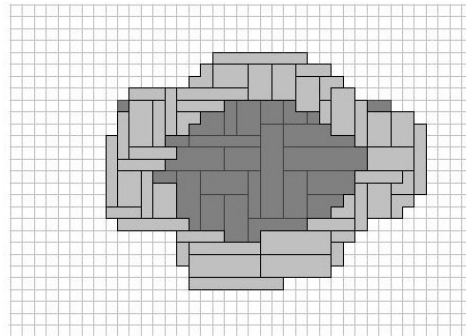


Fig. 9. Instructions generated for the 5th layer of the Stanford bunny. The dark grey areas indicate bricks in previous layers, and the light grey the bricks of the current layer.

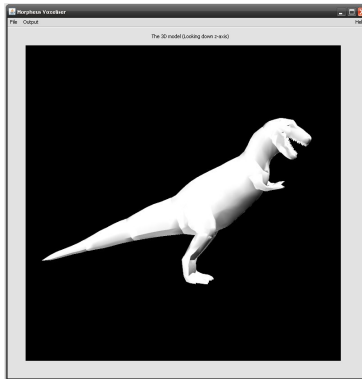


Fig. 10. A 3D model of a dinosaur.

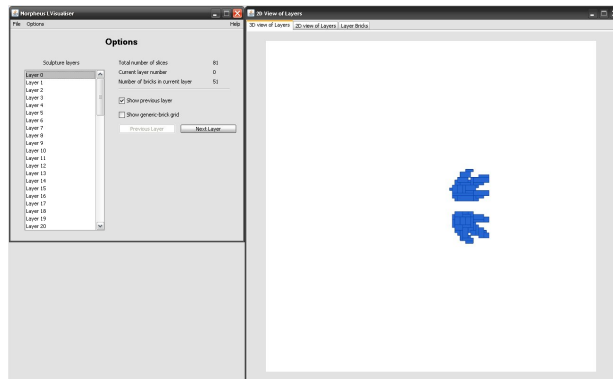


Fig. 11. Instructions for the dinosaur being generated.

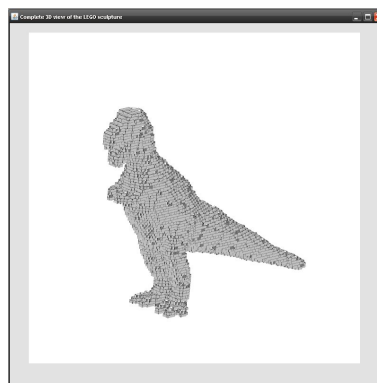


Fig. 12. Completed LEGO sculpture of the dinosaur.