Automated Brick Sculpture Construction

A THESIS PRESENTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT THE UNIVERSITY OF STELLENBOSCH

By
Eugene Smal
August 2008

Supervised by:  Dr. Lynette van Zijl
Declaration

I the undersigned hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

Signature: .................. Date: .................
Summary

In this thesis we consider the modelling of a particular layout optimisation problem, namely, the LEGO construction problem. The LEGO construction problem, in short, concerns the optimal layout of a set of LEGO bricks to represent a given object.

Our goal is to develop a software package which LEGO enthusiasts can use to construct LEGO sculptures for any real-world object.

We therefore not only consider the layout optimisation problem, but also the generation of the input data required by the LEGO construction problem. We show that by using 3D geometric models to represent the real-world object, our implemented voxelisation technique delivers accurate input data for the LEGO construction problem.

The LEGO construction problem has previously been solved with optimisation techniques based on simulated annealing, evolutionary algorithms, and a beam search approach. These techniques all indicate that it is possible to generate LEGO building instructions for real-world objects, albeit not necessarily in reasonable time.

We show that the LEGO construction 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. We show that the use of cellular automata gives comparable layout results in general, and improves the results in many respects. The cellular automata method requires substantially less memory and generally uses fewer LEGO bricks to construct the LEGO sculpture when using comparable execution times.
Afrikaanse opsomming

In hierdie tesis beskou ons die modellering van 'n spesifieke uitleg-optimeringsprobleem, naamlik, die LEGO konstruksie probleem. Die LEGO konstruksieprobleem, kortliks gestel, beskou die optimale uitleg vir 'n stel LEGO blokke in die bou proses van 'n gegewe voorwerp.

Die doel van hierdie tesis was om 'n rekenaarpakket te ontwikkel wat deur LEGO entoesiaste gebruik kan word om LEGO standbeelde van enige regte-wêreld objek te konstrueer. Ons beskou dus nie net die LEGO konstruksieprobleem as sulks nie, maar ook die generasie van die toevoerdata wat benodig word vir die LEGO konstruksieprobleem. Deur gebruik te maak van 3D geometriese modelle om die regte-wêreld objek voor te stel, wys ons dat akkurate toevoerdata vir die LEGO konstruksie probleem genereer word deur ons voorgestelde voxelisasie tegniek.

Vorige oplossings vir die LEGO konstruksieprobleem het gebruik gemaak van optimeringstegnieke gebaseer op gesimuleerde tempering, genetiese algoritmes, en 'n straalsoektog metode. Hierdie tegnieke dui aan dat dit wel moontlik is om LEGO bou instruksies te genereer vir regte-wêreld objekte, alhoewel nie noodwendig in 'n redelike tyd nie.

Ons toon aan dat die LEGO konstruksieprobleem maklik gemodelleer kan word deur sellulêre automate, gegee dat selle as versamelings beskou word wat kan saamsmelt of verdeel in elke tydstep van die evolusieproses van die sellulêre automaat. Ons toon aan dat die gebruik van sellulêre automate oor die algemeen vergelykbare resultate lewer met vorige metodes, en die resultate verbeter in baie opsigte. Die sellulêre automate
gebruik aansienlik minder geheue en gebruik oor die algemeen minder LEGO blokke om die LEGO standbeeld te konstrueer vir vergelykbare uitvoer tye.
Acknowledgements

I would like to thank

- first and foremost, my supervisor Dr Lynette van Zijl, for her invaluable guidance and motivation throughout this thesis; and

- my family for their continuous support.

Special appreciation to the National Research Foundation (NRF) and the Harry Crossley Bursary for their financial assistance.
Contents

1 Introduction ........................................ 1
   1.1 Thesis outline ................................ 3

2 Literature overview .................................. 4
   2.1 The LEGO construction problem .............. 4
   2.2 Definitions .................................. 7
   2.3 LEGO construction as a combinatorial
       optimisation problem ....................... 10
   2.4 Optimisation techniques ...................... 14
       2.4.1 Local search ............................ 14
       2.4.2 Simulated annealing .................. 15
       2.4.3 Library pattern filling .............. 16
       2.4.4 Evolutionary algorithms .............. 17
       2.4.5 Beam search .......................... 25
   2.5 Related work ................................ 27
3 3D real-world object representation

3.1 Exploratory implementations ........................................... 30
   3.1.1 Using a ray-tracer .................................................. 30
   3.1.2 Using triangle-subdivision ....................................... 32
3.2 Voxelisation .............................................................. 33
   3.2.1 Fast triangle-cube intersection test ............................ 35
   3.2.2 Solidifying the legolised representation ....................... 39
   3.2.3 Hollowing out the legolised representation .................... 39
   3.2.4 Results .............................................................. 40

4 Beam search ............................................................... 46
   4.1 Method overview ....................................................... 46
   4.2 Generation of successor nodes ...................................... 48
   4.3 Cost function .......................................................... 51
   4.4 Pruning the tree ....................................................... 52
   4.5 Data structures used .................................................. 56

5 Cellular automata with cell clustering .............................. 60
   5.1 Background ............................................................. 60
   5.2 CA with cell clustering ............................................... 61
   5.3 Encoding the LEGO construction problem as a CA ............... 65
      5.3.1 Merging .......................................................... 66
5.3.2 Splitting ................................................................. 70
5.4 Multicoloured LEGO sculptures ........................................... 71
5.5 Data structures used ....................................................... 74

6 Results 77
6.1 Measures of quality ......................................................... 77
6.2 Experiments ................................................................. 78
   6.2.1 Number of bricks used ................................................. 80
   6.2.2 Cost function value .................................................... 82
   6.2.3 Execution time .......................................................... 83
   6.2.4 Memory usage .......................................................... 83
   6.2.5 Multicoloured sculptures .............................................. 84
   6.2.6 Ease of building ....................................................... 84
   6.2.7 Ease of implementation .............................................. 85
   6.2.8 Restricted brick sets ................................................. 86

7 Conclusion and future work 88
7.1 Conclusion .............................................................. 88
7.2 Future work ............................................................ 90

Bibliography 93
Appendices 97

A Results 97

A.1 3D geometric models used 97

A.2 Complete test results 97

B LSculpt 109

B.1 Voxelisation of boundary representation using orientated LEGO plates. 109

C The “family” LEGO brick set 113
Chapter 1

Introduction

The LEGO company is one of the largest and most successful toy manufacturers in the world. Their success is partly due to the quality of their visual building instructions which accompany all LEGO sets. These instructions were, until quite recently, painstakingly and manually developed by so-called LEGO master builders [34].

In 1998 [34] and again in 2001 [30], the LEGO company presented an open problem to the scientific community at large, namely, “Given any 3D body, how can it be built from LEGO bricks?” (see Figure 1.1, page 3). This became known as the LEGO construction problem. What the LEGO company required, was a computer program that would be able to generate LEGO building instructions for any real-world object within a reasonable amount of time. Although the LEGO construction problem is easy to understand, it is not necessarily easy to solve on a computer. Due to the various different LEGO bricks available and the multiple ways in which an object can be constructed from these bricks, the problem quickly becomes intractable.

The LEGO construction problem can be seen as a three dimensional area filling problem. The two dimensional area filling problem is already considered to be NP-complete, but in special cases approximation methods can be used to deliver acceptable solutions. Thus, to solve the three dimensional area filling problem, one could potentially divide...
CHAPTER 1. INTRODUCTION

the problem into several smaller two dimensional problems. This can be achieved by
dividing the real-world object into different layers, and constructing each layer separately.
However, such a division will not always lead to viable solutions. If we construct each
layer separately and then join the separate layers on top of each other to construct the
final product, the resulting LEGO sculpture may not be connected\(^1\). Therefore, each
layer must at least take into account the layer above and below it to ensure that the
LEGO sculpture will be connected.

The LEGO construction problem, like most area filling problems, has the properties of
an optimisation problem. The cost of building the sculpture must be kept to a minimum,
while the strength and stability of the sculpture are not to be compromised. The search
space is extremely large and there can even be more than one viable solution, which
makes finding the best solution almost impossible.

A number of researchers have already investigated the LEGO construction problem in
terms of an optimisation problem. The main techniques used to solve the problem
were simulated annealing \([20]\), evolutionary algorithms \([27, 30]\) and using a beam search
\([39]\). These techniques all demonstrate that it is possible to generate LEGO building
instructions for real-world objects, albeit not necessarily in reasonable time. The LEGO
company currently uses a proprietary product called Brickbuilder for LEGO construction
problems. However, this product only creates the outline of the LEGO sculpture, and
not building instructions per se. It is therefore left to the user to select and place the
bricks to ensure a connected sculpture. The product is not available to the public.

\(^1\)A LEGO sculpture is connected if each layer is attached to the layers above and below it, in such
a way that the sculpture, when completely built, forms a single 3D object. A LEGO sculpture will
therefore not be connected if it can be separated into multiple smaller objects, without detaching any
LEGO brick from another.
CHAPTER 1. INTRODUCTION

Figure 1.1: Winnie the Pooh [12] (left) and its corresponding LEGO sculpture [13] (right).

1.1 Thesis outline

Chapter 2 provides an overview of the LEGO construction problem and then presents existing techniques to solve the problem. In Chapter 3 we discuss how the real-world object is represented as input for the LEGO construction problem and how the input can be constructed. Chapter 4 presents the current standard method to solve the LEGO construction problem, namely, a beam search. In Chapter 5 we discuss our new approach to solving the LEGO construction problem using cellular automata with cell clustering. We compare the results of the beam search and the cellular automata methods in Chapter 6. Finally, we conclude and discuss possible future work in Chapter 7.
Chapter 2

Literature overview

2.1 The LEGO construction problem

The LEGO construction problem concerns itself with the development of a computer application that, given any real-world object, generates the LEGO building instructions for that object.

The traditional approach to solving the LEGO construction problem is to virtually cut a digital representation of the 3D object into horizontal two-dimensional (2D) layers. The problem then reduces to a series of 2D solutions which can be joined together to produce the final 3D LEGO sculpture.

The main aspects of the problem can be summarised \([20, 34]\) by the following:

- The application must take as input a *legolised* representation of the real-world object. The *legolised* representation is a matrix containing ones in the places where a generic brick can be placed and zeros where there must be empty spaces (see Figure 3.1, page 28). A generic brick is the smallest possible square brick, 7.9mm long and 1.11mm high and contains one stud to connect to other bricks (see Figure 2.1, page 5).
The set of LEGO bricks that may be used to reconstruct the real-world object is typically restricted to the “family” LEGO brick set. These LEGO bricks are integer multiples of the dimensions of the generic brick. Thus each of these blocks can be replaced by several generic bricks stacked together. The larger DUPLO bricks have also been included in the set of allowed bricks to reduce the overall cost of the sculpture (a few large bricks cost less than many small bricks). The set of allowed bricks and their dimensions are given in Appendix C, Table C.1 and Table C.2, on page 114.

- The LEGO sculpture must be one connected object when built.

- To save money and time when building with actual LEGO bricks, the inside of the sculpture should be kept hollow as far as possible. The recommended width from the outside to the inside of the model should be kept to approximately four generic bricks.

- An acceptable solution should be given within a reasonable time period.

- The application must be able to reconstruct large objects, since the original purpose of the program was to help develop LEGO sculptures for the LEGO theme parks. If large objects are ignored, the problem would be vastly simplified, as the running time and space required would decrease substantially.

- The application can ignore the colour of the real-world object and only produce instructions to build a monochromatic LEGO sculpture. However, the application must be extendable to incorporate the use of colour. Note that if the colour of the
real-world object is ignored, larger bricks can be used on the outside of the LEGO sculpture, since the colour boundaries can be ignored (see Figure 2.2).

![Image of LEGO sculpture](image)

**Figure 2.2:** If the colour of the LEGO bricks can be ignored, larger bricks can potentially be used, which will lower the number of bricks used and will therefore increase the stability and connectedness of the LEGO sculpture.

The LEGO construction problem is a large and complicated problem. The problem was therefore later simplified by Grower et al [20] to

- not include DUPLO bricks. DUPLO bricks over-complicate the problem by only being able to connect to a limited subset of the “family” LEGO bricks. They can only be connected to bricks that consist of an even number of generic bricks in its length and width, and other bricks must also have a height of at least three generic bricks to be able to connect to a DUPLO brick;

- allow only “family” LEGO bricks of height one, the height of the generic brick. Allowing bricks with different heights would add to the complexity of the algorithm that has to be developed, since one would have to keep track of bricks spanning multiple layers; and

- prohibit the interactive alteration of the *legolised* representation. By allowing the algorithm to alter the inside of the *legolised* representation, larger bricks could potentially be used to lower the number of bricks used and strengthen the sculpture, effectively lowering the cost of the sculpture. However, to determine when to allow
these changes and what their global effect would be, would increase the complexity of the algorithm. Even though a change could have a positive influence at the current construction area, it could have a negative influence on another part of the sculpture that will be constructed later, by not leaving it enough freedom to change the *legolised* representation.

All the techniques that we will present in this chapter will focus on solving the simplified LEGO construction problem rather than the original problem. The simplified set of “family” bricks is known as the standard LEGO brick set and is shown in Figure 2.3.

![Figure 2.3: The list of standard LEGO bricks](image)

### 2.2 Definitions

Before we discuss the simplified LEGO construction problem in more detail, we introduce some terminology.

**Attached brick:** A LEGO brick is attached if it is joined to at least one other LEGO brick in the LEGO sculpture.

**Neighbouring bricks:** The neighbouring bricks for any given brick are those bricks which touch at least one of the sides of the brick directly, within the same layer (see Figure 2.4, page 8).

**Brick direction:** We define a direction for each brick. For rectangular bricks, the direction is assigned depending on whether the brick is lying horizontally or vertically.
CHAPTER 2. LITERATURE OVERVIEW

Figure 2.4: The blue bricks are all the neighbouring bricks of the yellow brick. The red bricks, although touching the corners of the yellow brick, are not neighbouring bricks. when looked at from the top (see Figure 2.5). We say that the direction is either horizontal or vertical. The square bricks and the L-shaped bricks are considered to be both horizontal and vertical in direction.

Figure 2.5: The red brick has a vertical direction and the blue brick has a horizontal direction.

Parallel and perpendicular bricks: Bricks are parallel to each other if they have the same direction and perpendicular if they have opposite directions.

Brick boundaries: The sides of each brick create a border, when looking from the top of the brick, that makes it possible to distinguish it from its neighbouring bricks. The lines that define the border are called the (vertical) boundaries [20] of the brick (see Figure 2.6, page 9).
Given the definitions above, we now consider the general classes of optimisation problems [28]: discrete, combinatorial, and continuous optimisation problems. Discrete optimisation problems are optimisation problems where the solution space is finite. In discrete optimisation problems, the variables in the objective function are restricted to a set of discrete values (normally integer values). These problems can be solved in polynomial time.

A combinatorial optimisation problem is an optimisation problem where the solution space is extremely large, but finite. These problems are known to be NP-hard problems. Although small instances of the problem could possibly be solved in polynomial time, there exists no polynomial time algorithm to solve the problem exactly in general.

A continuous optimisation problem is an optimisation problem where the solution space is infinite. The variables in the objective function are not restricted and are usually real values. These problems are in general difficult to solve, and are also NP-hard problems.

When one builds a LEGO sculpture, there are numerous different ways of placing the bricks and choosing which bricks to use. The solution space is extremely large even for small LEGO sculptures. Although the solution space is extremely large, it is still finite and hence the LEGO construction problem is classified as a combinatorial optimisation problem.
2.3 LEGO construction as a combinatorial optimisation problem

Grower et al [20] first formulated the LEGO construction problem as a combinatorial optimisation problem, which makes use of heuristics to compute a cost function for the LEGO sculpture. In this section, we discuss their approach as a basis for a solution to the LEGO construction problem. We first consider the heuristics that help to ensure a stable and connected LEGO sculpture:

**Heuristic 1** A high percentage of the area of each brick should be covered, from above and below, by other bricks. This will help to increase the stability of the LEGO sculpture and will ensure that there are as few as possible bricks that are not attached to the sculpture.

**Heuristic 2** Larger bricks should be preferred over smaller bricks, since this results in a better overall stability for the sculpture. Using larger bricks also influences the cost of the LEGO sculpture, as a few larger bricks cost less than many smaller bricks.

**Heuristic 3** Bricks in consecutive layers should have alternating directionality. This will help yield a stronger support for the overall stability of the LEGO sculpture, as a brick is more likely to cover several bricks in the previous layer (see Figure 2.7, page 11).

**Heuristic 4** A high percentage of the vertical boundaries of each brick should be covered by bricks in the layers above and below. This helps to prevent a brick from being placed such that its boundaries match that of a brick in the previous layer, and potentially resulting in an unconnected LEGO sculpture.

**Heuristic 5** A brick must be placed such that, if either the short or the long side of the brick forms a T-shaped boundary with its neighbouring bricks, the middle of the side should be at the boundary defined by the neighbouring bricks (see Figure 2.8, page 11).
Figure 2.7: The arrows show the direction of the corresponding coloured bricks. The orange bricks are perpendicular to the green bricks, increasing the stability and strength of the LEGO sculpture.

Figure 2.8: The two red bricks show how a brick must be placed if it should encounter a boundary as indicated by the dashed line. The boundary is formed by the blue and yellow bricks and forms a T-shaped boundary when either of the red bricks is placed.

Heuristic 6 If a brick covers a vertical boundary in the previous layer, it should be centered on the boundary (see Figure 2.9, page 12).
Given the heuristics above, Grower et al recommended using a cost function of the form

\[ P = C_1 P_1 + C_2 P_2 + C_3 P_3 + C_4 P_4, \]  

(1)

where the \(C_i\)'s are weight constants and

- \(P_1\) relates to the alternating directionality of the bricks in consecutive layers. This function is used to penalise bricks which do not adhere to Heuristic 3 above;

- \(P_2\) corresponds to how well the consecutive layers cover the vertical boundaries. The function penalises the placement of bricks which does not adhere to Heuristic 4 above;

- \(P_3\) directly represents Heuristic 5; and

- \(P_4\) was added to explicitly encourage the use of larger bricks. \(P_2\) already tends to favour larger bricks, as they have less boundary per unit of area.

Grower et al recommended optimising the cost function given in Eq. (1) using either a local search method [14] or a simulated annealing technique [36]. Using a local search
method to solve the optimisation problem is almost certainly the most intuitive, as this
corresponds to what one would naturally do when building the LEGO sculpture by hand.
However, there are various other optimisation techniques that one could use to solve the
LEGO construction problem.

In general there are two main classes of optimisation techniques [28]: deterministic tech-
niques and stochastic techniques. Deterministic techniques, also known as exact tech-
niques, are used to find the global optimal solution when the solution space is relatively
small. These techniques require a clear relation between the characteristics of the so-
lution space and characteristics of the problem, in order to search the solution space
efficiently. Deterministic methods make use of efficient state space search methods [29],
brANCH-and-bound methods [17], or algebraic methods [28] to find the optimal solution.
When no relation between the quality of neighbouring solutions can be found, or when
the solution space is too large, the deterministic techniques cannot be used. This would
require an exhaustive search through the solution space, which is not feasible.

Stochastic techniques focus on optimisation problems where the solution space is ex-
tremely large and the global optimal solution is not always required. These techniques
sacrifice optimality for finding good solutions in a reasonable amount of time. These tech-
niques use heuristics and probability theory to guide the search. Stochastic optimisation
techniques include hill-climbing (local search) [29], simulated annealing [36], evolutionary
algorithms [16, 18], tabu search [14, 19], and iterated local search [14] techniques.

Deterministic optimisation techniques cannot be used to solve the LEGO construction
problem as the solution space is extremely large and there is no clear relation that can be
used to guide the search from one possible solution to another. Therefore, we will focus
on stochastic optimisation techniques that focus on solving combinatorial optimisation
problems.

In the next section we will briefly discuss a few of these optimisation techniques and how
they can or have been applied to the LEGO construction problem.
2.4 Optimisation techniques

2.4.1 Local search

Grower et al [20] briefly described how one could use a local search technique to solve the LEGO construction problem. The method we describe is our own interpretation using their method as a basis (for the original description, see [20]).

In a local search technique, one considers a small subregion at a time and search through a wide range of possible brick placements, for the best brick placement to fill the subregion. Instead of using all the bricks from the best brick placement, one or more bricks are selected from it and permanently placed into the layer. The subregion then moves or changes so that the next subregion overlaps the previous subregion. Therefore, the subregion can be seen as a sliding window.

The process is repeated until the entire layer is filled.

The small subregion is used to help predict how the bricks being placed will effect the global solution. If no subregion is used and the best possible brick is just always added, one could be forced to add small bricks at the end to fill up the holes. It is exactly to avoid that situation that we make use of the small subregion to help predict what bricks would lead to a better final solution.

The quality of each layer depends on the size of the subregion and the number of possible brick placements examined. If the size of the subregion is too small, the information gained about the global influence will be negligible. If a larger subregion is used, the number of possible brick placements can increase substantially, which would increase the execution time needed to find the best possible brick placement. Finding the optimal size for the subregion can be a difficult task, as the size and characteristics of real-world 3D objects can differ greatly from another. For example, sections of the object which are not supported from below, such as a person’s arms, could potentially be disconnected from the main sculpture if the size of the subregion is too small to allow enough bricks
to attach the section to the rest of the sculpture.

In summary, a local search method uses a small subregion (sliding window) to predict how brick placements will influence the final brick layout. The method finds the best possible brick layout for a given subregion and then places one or more bricks from the subregion into the layer permanently. The size of the subregion will determine the effectiveness of the method, as a too small subregion will result in negligible predictions, while a too large subregion will increase the execution time substantially.

2.4.2 Simulated annealing

Simulated annealing is a variant of the hill-climbing technique [29]. Hill-climbing is a greedy strategy where the algorithm computes all possible successor states for the current state, and then selects the best successor. A common problem with hill-climbing is that a local optimum can be found instead of the global optimum.

Simulated annealing selects a successor state from all the possible successors states at random. If the randomly selected successor state is an improvement over the current state, it becomes the new current state. If the successor state is not an improvement, it will be set as the new current state with a probability of less than one. This allows potentially “bad” search decisions to be made which would hopefully lead to a better global solution. The probability of the state being accepted as the new current state depends on how drastically the quality of the current state will change, should the new successor be accepted. If the quality will change drastically, the probability that the successor state is accepted will be lower. As the search progresses, the probability of accepting “bad” successor states are decreased to make them less likely to be accepted. This allows a broader search early on when one would want to find a good starting point and then to restrict the algorithm when possibly near the global optimal solution.

Simulated annealing can be applied to the LEGO construction problem [20] by dividing each layer into smaller sized subregions. Each subregion is then filled with an arbitrary
CHAPTER 2. LITERATURE OVERVIEW

initial placement of bricks, yielding the starting state. At each search step all the possible successor states are generated in parallel by considering each subregion separately, removing a small number of bricks, and then replacing them by new bricks. Successor states are selected randomly until a successor is accepted as the new current state. The number of search steps can be predefined, or the algorithm can stop when no more improvements are possible or an acceptable quality has been reached.

The success of this approach depends largely on the speed with which the cost function can be computed for each possible successor, and the size of the subregions being used. Small subregions will limit the feasibility of the approach, since the number of bricks in each subregion that can be replaced at every search step will become too small to have a positive global effect. Larger subregions will deliver better results, but then the number of search steps required increases quadratically with the problem size (see [20] for details).

2.4.3 Library pattern filling

Grower et al also suggested the use of a pattern filling dictionary. The dictionary would contain the optimal brick placements for known shapes and features. The reasoning behind this is that most legolised real-world objects will have layers which resemble primitive shapes. Therefore, if the shape can be identified, the optimal solution can be found in the dictionary. Hence, the problem is changed to finding a match in the dictionary instead of searching through possible layouts.

The problem with this technique is that the shape of a layer does not uniquely identify how a given layer should be built. One could for instance have two square shaped layers, but which are different in size. Generally both square shaped layers will have LEGO bricks of different size and could even use different brick placements (patterns). It would be ideal if we could use the same brick placement or pattern and only scale the size of the bricks being used. However, this will not always be possible as the size of the bricks
are restricted, and therefore a different brick placement will be required. Even if scaling is possible, the scaled brick placement would most likely not be optimal for the given shape size.

Another problem with this technique is that consecutive layers which have the same shape and size cannot be built in the same way. This would make the LEGO sculpture unstable and possibly disconnected. Therefore, multiple different optimal layouts would have to be stored in the dictionary to ensure that consecutive layers are constructed to increase the overall stability of the LEGO sculpture.

Although the method could be seen as a possible alternative, we feel that the size of the dictionary and the time needed to detect the shape of each layer would make the technique impractical.

2.4.4 Evolutionary algorithms

Evolutionary algorithms [16, 26] can be applied to solve optimisation problems, by making use of mechanisms inspired by the principles of evolution found in nature. In particular, optimisation problems are solved by simulating the biological processes of natural selection, reproduction, mutation, and the survival of the fittest. There are numerous different evolutionary algorithms, but each differs only in their implementation detail, while all following the same evolution process. For optimisation problems the most popular evolutionary algorithm is called a genetic algorithm.

Evolutionary algorithms start with an initial population. The population is normally a randomly generated set of candidate solutions to the problem being solved. Each candidate solution can be seen as an individual in the population. The algorithm uses a fitness function to evaluate the fitness of each individual in the population. A selection method is used to select parents for reproduction, which corresponds to natural selection. The selection method selects the parents for reproduction from the population with a bias towards higher fitness.
The selected parents reproduce by either recombination or mutation. The recombination operator, also known as the crossover operator, acts on two selected parents and creates one or two new candidate solutions. The mutation operator acts on a single parent and creates only one new candidate. Both the recombination and mutation operators are used to generate a set of candidate solutions known as the offspring. The offspring compete with the old population for their place in the new generation by limiting the size of the population, which corresponds to survival of the fittest. The evolution process is repeated until either a candidate with a sufficient quality has been found or until a fixed number of iterations has been executed.

The speed and success of the evolutionary algorithm greatly depends on how the candidate solutions are represented. A good representation will allow recombination and mutation operators to generate new candidate solutions faster. If these operators can be applied faster, a larger population can be used to generate candidate solutions, broadening the optimisation search. If the recombination and mutation operators are too slow, only a few candidate solutions can be generated, which would create a narrow search space. The candidate solutions can either be represented directly or indirectly. In a direct representation, the decision variables and problem functions are used directly whereas, in an indirect representation the problem is encoded in a series of bit strings that are manipulated by the algorithm. The indirect representation is generally more efficient, but more difficult to formulate.

Evolutionary algorithms were first used to solve the LEGO construction problem by Petrovic [30], who implemented an application that successfully generates brick layouts. The results, however, show that the method is relatively slow and requires significant execution time to construct even small sculptures.

Petrovic used evolutionary algorithms to construct the LEGO sculpture layer by layer. Each layer is evolved a predefined number of times and the best solution found is taken as the final layout for that layer. Petrovic used a direct representation of the candidate solutions, where each candidate solution was represented by an unordered list of triples.
Each triple stores the row and column positions, and the brick type, for each brick.

Petrovic experimented with two algorithms to generate the initial population. The first algorithm fills the ones in the *legolised* representation from left to right, and from top to bottom. Each time the algorithm finds an unfilled one, it probabilistically selects one of the possible bricks that fits into the layout. Larger bricks have a greater probability of being selected. The second algorithm first fills the ones that form the edges of the layer and then fills the remaining ones inside in a random order. The experiments showed that the second, edge first, algorithm produced better individuals. Petrovic therefore used only the second algorithm in his final experiments.

Once the initial population has been generated, the evolution process begins by evaluating each individual in the population. The fitness function used to evaluate each individual is similar to that of [20] and is given by:

\[
\text{Fitness} = C_{\text{numbricks}} \times \text{numbricks} + C_{\text{perpend}} \times \text{perpend} \\
+ C_{\text{edge}} \times \text{edge} + C_{\text{uncovered}} \times \text{uncovered} \\
+ C_{\text{otherbricks}} \times \text{otherbricks} + C_{\text{neighbour}} \times \text{neighbour},
\]

where the \( C \)'s are weight constants and

- the \( \text{numbricks} \) variable is the number of bricks in the sculpture. Note that Petrovic assumes that all bricks have the same cost, and hence larger bricks will implicitly be used where possible;

- the \( \text{perpend} \) variable corresponds to the directionality of the bricks in consecutive layers. This corresponds to Heuristic 3 in Section 2.3, page 10;

- the \( \text{edge} \) variable represents the number of edges of each brick which lies at the same location as that of bricks from the previous layer;

- the \( \text{uncovered} \) variable describes the area of each brick which is not covered by
bricks in the previous and following layers;

- for a given brick, the otherbricks variable represents the number of bricks in the previous layer, covered by this brick. By using the otherbricks variable, bricks are placed so that they cover as many bricks as possible in the previous layer, and therefore increase the overall stability of the sculpture; and

- the neighbour variable corresponds directly to Heuristic 5 in Section 2.3, page 10, where T-shaped brick boundaries are favoured.

After the population has been evaluated, a set of parents is selected. Petrovic experimented with two selection methods, namely, steady-state selection and roulette-wheel selection.

In steady-state selection, the $k$ best individuals are selected as parents. In roulette-wheel selection, each individual is assigned a pie shaped piece of a roulette wheel. The size of the piece is proportional to the fitness of the individual. For example, suppose that there are three individuals $x_1, x_2,$ and $x_3$, with respective fitnesses 2, 3, and 5. The total sum of the fitnesses is 10. Each individual is then assigned $\frac{2}{10}$ of the roulette wheel. Therefore, $x_1$ will be assigned $\frac{1}{5}$ of the roulette wheel and $x_2$ and $x_3$ will be assigned $\frac{3}{10}$ and $\frac{1}{2}$ of the roulette wheel respectively (see Figure 2.10, page 21). A single parent is selected by spinning the wheel similar to a real roulette wheel. When the roulette wheel stops, the parent corresponding to the section containing the “ball” is selected. Therefore, to select $N$ parents the roulette wheel will be spun $N$ times. This method is known as a global selection method, as the fitness of each individual is compared to the total sum of the fitnesses of all the individuals. The method is therefore biased towards individuals with a higher fitness.

Once the parents are selected, recombination or mutation operators must be used to produce new candidate solutions. The recombination operator takes two parents and selects a random rectangular subregion in one of the two parents. The offspring is composed of all the bricks inside the rectangle from the one parent, and all the bricks
Figure 2.10: Three individuals $x_1, x_2, x_3$ with respective fitnesses 2, 3, and 5, allocated pieces of the roulette wheel proportional to their fitness.

from the other parent which do not conflict\footnote{A brick conflicts with an already placed brick, if the brick cannot be placed correctly without removing the other brick. Both bricks want to occupy the same area.} with bricks already placed. The offspring can have unfilled ones due to conflicting bricks. These unfilled ones are filled by selecting bricks probabilistically, with larger bricks having a greater chance of being selected.

The mutation operator probabilistically selects one of seven different mutation operations and mutates a given parent to create one new candidate solution. The parent is mutated in one of the following ways:

- A single brick is replaced by another randomly selected brick. Bricks that conflict with the new brick are removed. Therefore, the new candidate solution could have unfilled ones.

- A new brick is placed at a randomly selected unfilled one, and all bricks that conflict with the new brick are removed.

- A randomly selected brick is shifted by the length of one generic brick in one of the four possible directions (that is, up, down, left, or right). Bricks that conflict with the placement of the new brick are removed.

- A single randomly selected brick is removed from the layout.
• A randomly selected brick is extended by the length of one generic brick in any of the four possible directions and all conflicting bricks are removed.

• All the bricks in a randomly selected rectangular region are removed and replaced by randomly selected new bricks, with larger bricks being selected with a higher probability.

• A whole new layout is generated by using one of the two algorithms used to generate the initial population. This is therefore not a direct mutation from parent to candidate solution, as the parent solution is not used in any way. This option allows for some random candidate solutions to be generated in order to broaden the search and to help prevent the method from settling at a local optimum.

Since the mutation operator can generate candidate solutions with unfilled ones, each mutated candidate is first filled to represent a valid solution before its cost can be evaluated.

The evolution process is repeated a fixed number of iterations, and the best solution found is then taken as the final layout for the layer. When all layers have been evolved, the LEGO sculpture has been constructed and no further changes are made to the layers.

Petrovic performed numerous experiments using so-called single-population and multiple-population techniques. In the multiple-population technique, several populations are evolved in parallel and occasionally several individual solutions are exchanged between the populations. In his multiple-population technique, the populations are arranged in a directed cycle, and the individuals can only move to their neighbouring population in the cycle. This method is also known as a stepping-stone migration scheme. By moving individuals between populations, the algorithm can explore different paths, which prevents the search area from being too narrow.

Petrovic used both steady-state and roulette-wheel selection methods for the single-population experiments. The experiments showed that the steady-state selection method
is more continuous and converges significantly faster (an order of magnitude faster) to the optimum solution than that of the roulette selection method. Petrovic therefore used only the steady-state selection method for the multiple-population experiments.

The multiple-population experiments were compared using 5, 10, and 20 populations. Each population consisted of 500 individuals. The number of individuals moved between the populations were varied between 5, 10, and 20 individuals. A single-population experiment was also executed, where the population size was equal to the sum of the sizes of all the populations, for all three different population numbers.

In all the above experiments, the single-population technique required significantly fewer generations to reach its optimum solution than the multiple-population technique. The single-population required on average only a 100 generations, whereas the multiple-population techniques required around a 1000 generations to reach a similar result. When comparing only the different multiple-population sizes, Petrovic found that there was no significant difference between using 10 and 20 populations, but five populations always performed the worst, almost never reaching the same quality results. The number of individuals moved between the populations had no significant influence on the results produced.

Petrovic later extended his work by implementing an improved genetic algorithm. The improved algorithm uses a combination of a direct and an indirect representation of the solution, instead of only a direct representation. The new representation allows the bricks to be shifted and duplicated to form building patterns (see Figure 2.11, page 24). The candidate solutions are now represented by a list of 5-tuples which store the row and column positions of each brick, the type of brick used, the number of rows (measured in generic brick lengths) and the number of columns used to shift the brick to form a building pattern, and the number of times the brick was shifted to create the pattern. Note that this representation only allows the brick pattern to extend in one direction.

Na [27] later extended the indirect representation of Petrovic by allowing the building patterns to extend in two directions instead of one (see Figure 2.12, page 24). The
results showed a noticeable increase in performance over the previous implementations of Petrovic, but the method still requires a significant amount of execution time to construct small LEGO sculptures. A $50 \times 50 \times 30$ hollow cylinder required 52 hours to construct\textsuperscript{2}. As the LEGO sculptures built by LEGO employees are far more complex and significantly larger, in our opinion this method does not meet the requirements for a practical solution to the LEGO construction problem.

\textbf{Figure 2.11}: Brick patterns constructed by shifting a brick by a number of columns and rows.

\textbf{Figure 2.12}: Brick patterns constructed by shifting a brick by a number of columns and rows in two directions.

\textsuperscript{2}In comparison, the beam search method takes about 500 seconds to construct the same sculpture, with similar quality.
2.4.5 Beam search

At the 2005 Brickfest conference, Winkler [39] presented a beam search technique to solve the LEGO construction problem.

A beam search is similar to a best-first search algorithm [29]. In a best-first search algorithm, all the possible successors to the current state are generated, and all are evaluated using a cost function. The successor with the best cost is then selected as the new current state. The algorithm therefore always chooses the best local solution, and hence is a hill-climbing technique.

A beam search works in a similar way, but instead of only using the best successor, it uses the best $k$ successors. At each step, all the parent states generate all their successors. The best $k$ successors over all the successors generated are selected and added to their respective parents. If some parent states do not have any successor states, the parent can be pruned from the search tree. Therefore, the technique bounds the width of the search tree by $k$, where each layer in the tree contains the best $k$ possible successors.

One problem with the technique is that it can occasionally focus on a too narrow search space, which in turn can result in bad solutions. One possible improvement [29] to the method is to, instead of always selecting the $k$ best successors, to select the $k$ successors probabilistically with a higher probability of selecting the lower cost successors. This will create a broader search space.

Winkler made use of 3D geometric models to represent the real-world object. The 3D geometric model is sliced into layers by using ray-tracing techniques included in the POV-Ray [9] application. The POV-Ray application slices the model into layers by moving two parallel planes through the 3D geometric model. Only the intersection of the model and the planes are rendered and saved as a monochrome image. The images are similar to medical CT scans. The images of each layer are then converted to a binary grid, which forms one layer in the *legolised* representation of the real-world object.
Winkler made use of a beam search to construct a lattice of maximum $k$ possible solution layouts, for a given layer. The layers are built consecutively. By searching through the lattice, the best possible brick placement can be found.

The algorithm works by first numbering all the ones (in the *legolised* representation) that need to be filled. Then, at each step, the lowest numbered unfilled one is filled using the $k$ best brick placements. The algorithm continues to fill the ones until all the ones have been filled. Note that by filling a one with a brick, not only that one is filled, but also all the ones that are covered by the brick. Therefore, the ones will not necessarily be filled in order. By numbering the ones, the maximum depth of the lattice is bounded by the number of unfilled ones, since in the worst case each one could be filled by a single generic brick.

Depending on the size of $k$, the lattice could grow extremely large. Some techniques that can be used to keep the size of the lattice manageable, include:

- the lattice can be generated on-the-fly by discarding nodes far in the past (nodes higher up in the lattice);
- keeping the LEGO sculpture hollow greatly reduces the number of squares that must be filled and will substantially decrease the number of possible brick placements that can be made; and
- to save memory, a bit packing method can be used to keep track of the squares which have been filled at each node in the tree.

Winkler gave no experimental results for his method. However, in chapter 4 we discuss our own implementation of his method, and show that it delivers extremely good results.
2.5 Related work

We note that there are other efforts related to the LEGO construction problem, albeit not direct attempts to solve it. For example, Lambrecht [25] developed a software application called LSculpt, to help the user build LEGO sculptures with improved detail. The application uses specialised LEGO bricks such that the LEGO brick plates can be orientated according to any of the three standard axes (the $x$, $y$ and $z$ axis). The method improves the detail of the entire LEGO sculpture by orientating small bricks plates in the directions which will deliver the most detail. The application does not produce LEGO building instructions, but it produces an LDraw [22] file consisting of $1 \times 1$ brick plates. The brick plates are in no way connected to each other and will therefore not produce a connected sculpture. It is left to the user to select and place the specialised bricks and plates, such that the different regions are connected together. Since this method gives an interesting perspective on LEGO sculpture construction, we include a brief discussion in Appendix B, page 109.

Other applications that have been developed are PicToBrick [7] which generates LEGO mosaics from digital images, and the various CAD packages such as LDraw [22], ML-CAD [24], and LEGO Digital Designer [6] to design LEGO models.

In this chapter we discussed the LEGO construction problem as stated by the LEGO company and showed how the LEGO construction problem has been simplified by Grower et al [20]. We discussed various optimisation methods that have been implemented or suggested by other researchers in order to solve the simplified LEGO construction problem. In the next chapter we will discuss how a real-world object can be represented as input for the LEGO construction problem and we will discuss various techniques that can be used to generate the input.
Chapter 3

3D real-world object representation

In the original LEGO construction problem [34], the LEGO company engineers specified that the application must accept a legolised representation as input. The legolised representation is a 3D matrix consisting of zeros and ones, representing the real-world 3D object. A one represents a part of the sculpture that has the size of one generic brick, whereas a zero represents an empty space of the same size (see Figure 3.1).

![Figure 3.1: A horizontal cut of a chess pawn legolised.](image)

Although the LEGO company engineers defined the legolised representation as the application input, they did not give any information on how to create it. To create the
legolised representation by hand is a cumbersome task even for small objects. Although one could possibly use expensive 3D scanners or 3D reconstruction cameras to construct the legolised representation from the real-world object, this will not be a practical solution for most LEGO building enthusiasts. Therefore, we set out to create an input method that would be freely available, user-friendly, and efficient, while still delivering good quality results.

Winkler [39] suggested the use of 3D geometric models to represent the real-world object. 3D modelling software such as Blender [2] is freely available, giving the user the ability to construct a wide range of 3D geometric models. These 3D geometric models store a mesh which contains all the vertices, polygons and normals of the 3D object. It can include colours, textures and even skeleton behaviour. These models can be large and contain vast amounts of detail, while using relatively small amounts of disk space. The added advantage of using 3D geometric models is that they can easily be scaled to any size. Therefore, users do not have to recreate the 3D geometric model each time a different sized sculpture is required. For more information on 3D representations and graphics terminology, the reader may consult [21].

We decided to focus on creating the legolised representation from 3D geometric models, as it accommodates most LEGO enthusiasts. Professional users can still use 3D scanners and 3D reconstruction methods to deliver a high quality 3D geometric model and then convert the 3D geometric model to a legolised representation.

We considered three different methods in our search to create a easy to use application that will deliver sufficient accuracy and speed. In the next section we will briefly discuss the first two methods that we implemented, which did not deliver the required results, and then we will explain in detail our final application which delivers excellent quality and is easy to use.
3.1 Exploratory implementations

3.1.1 Using a ray-tracer

The first method we implemented was to use a ray-tracer to convert a 3D geometric model to a "legolised" representation, as suggested by Winkler [39]. Ray-tracing [21] is a rendering technique that generates an image of a scene by simulating the way that rays of light travel in the real world. In the real world, rays of light are emitted from light sources and illuminate objects. These light rays can reflect or pass through objects depending on their properties. The reflected rays hit the eyes of the viewer and create the picture that is seen. Even though a scene contains a large number of light rays, only a few of these rays hit the eye. To trace all possible light rays in the scene, in order to find which rays hit the eye, is infeasible. Therefore, a ray-tracer works in reverse.

For each pixel in the final image, one or more viewing rays are shot from the eye or camera into the scene, to see if the ray intersects with any of the objects in the scene. These viewing rays therefore originate from the viewer. Every time the ray hits an object, the colour of the surface at that point is calculated. To ensure that the correct colour is found, the ray is traced back to each light source to determine the correct final colour. The final colour of the pixel depends on how the light is reflected and refracted until it reaches the viewer.

POV-Ray [9] is a free ray-tracer developed to create photo-realistic images from a 3D scene description. Winkler briefly explained how one could use POV-Ray to create a "legolised" representation: the 3D geometric model must first be converted to a POV-Ray compatible format. This can be done using PoseRay [8], a freely available 3D geometric model converter. PoseRay will convert the 3D geometric model into a complete 3D scene in POV-Ray format. POV-Ray can then be used to cut the 3D geometric model, or scene, into horizontal slices. Each of these slices are similar to medical CT scans (see Figure 3.2, page 31). These slices are created by using the built-in constructive solid...
geometry functions in POV-Ray. Constructive solid geometry functions find the union, intersection, or difference between two objects. In this case, the slices are generated by finding the intersection of the 3D geometric model and an infinite horizontal plane. Each of these slices can then be converted into a legolised representation by converting each black pixel to a zero and each white pixel to a one.

![Figure 3.2: The Stanford bunny 3D geometric model [11] cut into 15 horizontal slices by POV-Ray.](image)

In our experiments, we found that the method described above produced unwanted holes and artifacts with most of our 3D geometric models. This happens because a ray-tracer requires the 3D geometric model to be well defined in terms of both the inside and the outside of the model. Constructive solid geometry functions use the normal vectors of the polygons to calculate whether the ray is inside or outside of the object, in order to calculate the correct intersection between the plane and the 3D geometric model. 3D geometric models generally contain extra polygons on the inside of the model. Since the inside of the model is rarely seen by anyone, the model designer allows the use of larger polygons even if they cut into the inside of the model. These extra polygons make it impossible to determine whether the ray is inside or outside of the model, therefore resulting in holes or artifacts on the images.

Other disadvantages to using this method is that the user must have some understanding of POV-Ray, and the method requires the user to use numerous applications to convert the 3D geometric model to a legolised representation.
3.1.2 Using triangle-subdivision

In our second approach, we used the Jgeom [5] free geometry library developed for Java 3D [3]. It uses triangle-subdivision to calculate the constructive solid geometry of objects. The method requires the 3D geometric models to only consist of triangles. Therefore, if the 3D geometric model contains quadrangles, we convert them to triangles.

We created a Java [4] application that uses the Jgeom geometry library to calculate the intersection of the 3D geometric model and a horizontal plane. This approach is therefore similar to our ray-tracing technique (see Section 3.1.1, page 30), except that we use a geometric approach to finding the intersection. This effectively eliminated the problem of determining the inside and outside of the 3D geometric model.

To find the final intersection, the geometric library first calculates which triangles intersect with the plane. It then uses the Catmull Clark subdivision algorithm [38] to subdivide the triangles until they do not intersect with the plane anymore. The Catmull Clark subdivision algorithm is generally used to create a smoother 3D geometric model by subdividing the polygons into smaller, more accurate, representations. After the triangles have been subdivided, the geometry library fills the hole representing the inside of the object with triangles.

Our experiments show that this method requires substantially more execution time than the ray-tracing method. This is due to the large number of triangles that must be subdivided for each slice. If the 3D geometric model contains a large number of triangles, the method is extremely slow. The method did deliver more accurate results than the ray-tracing method, although we still encountered small holes on the inside of the model. We believe that it is due to a problem in the algorithm which covers the hole representing the inside of the object. When the shape is complicated, the algorithm will fail to find a set of triangles to perfectly cover the inside of the object.

Since neither the ray-tracing nor the triangle-subdivision methods deliver results of sufficient quality, we decided to implement an application that will voxelise the 3D geometric
model, by using a triangle-cube intersection test. This technique was implemented to eliminate the problem of determining the inside and the outside of the object and to remove the slow process of finding the geometric intersection with the horizontal plane. The method also lends itself to a legolised representation as the voxelised model can directly be used as the legolised representation.

In the following section we will discuss our solution for creating a legolised representation from a 3D geometric model consisting of triangles.

### 3.2 Voxelisation

Voxelisation [33] is the process in which a 3D geometric model is transformed into a volumetric representation consisting of voxels. A voxel can be seen as a 3D cube of space. The 3D geometric model is divided into voxels by partitioning the minimum axis-aligned bounding cube into smaller equal sized cubes. The minimum axis-aligned bounding cube is the smallest cube that contains the 3D geometric model, and is orientated such that the normal vectors of each face of the cube are aligned with the standard $x$, $y$, and $z$ axes. Each voxel can be either filled or empty.

A voxel is said to be empty if it does not contain any triangles of the 3D geometric model and filled if it contains at least one triangle. The voxelised model is then used as an approximation to the original 3D geometric model. The resolution of the voxel grid partitioning is important, as a rough resolution using larger voxels will limit the amount of detail, whereas a finer grid partitioning using smaller voxels will give more detail, but also significantly increase the number of voxels. Note that since we are partitioning an axis-aligned bounding cube, the number of voxels in the height, width, and the depth will be the same. Figure 3.3, page 34, shows a chess pawn voxelised with a grid resolution of $64^3$ ($64 \times 64 \times 64$) voxels.
When the voxel grid resolution is fine, the number of voxels that we have to test for possible triangle-cube intersection increases significantly. Depending on the 3D geometric model, a large number of the voxels can be empty. Each of these voxels will still require the testing of all the triangles for possible intersection. Fortunately, there is an alternative method [32] to directly calculating the voxelised model, based on octrees.

An octree [32] is a tree-structured representation that can be used to describe a set of binary valued volumetric data enclosed by a bounding cube. Each node in the octree represents a cube. The size of the cube depends on its depth in the octree.

The octree is constructed by recursively subdividing each cube into eight sub-cubes, starting at the root node. The root node is defined as the smallest cube containing the entire 3D object (the axis-aligned bounding cube).

A node can be either black, white, or grey. A node is black if it lies completely outside of the object; white if it lies completely inside the object; and grey if it is partially inside and partially outside of the object. Only grey nodes therefore need to be subdivided. If a cube contains at least one triangle, it is marked as grey. Nodes will only be marked as white when the required voxel grid resolution has been reached. Black and white nodes will therefore always be leaf nodes and grey nodes will be interior nodes.

Given the octree organisation, it is clearly only necessary to test a subset of the total
triangles for possible intersection with each cube, as only the triangles contained in the parent node can possibly intersect with the child node. This is a significant advantage to that of directly computing all the voxels, where each voxel has to be tested against all the triangles. Another advantage is that black cubes group many empty voxels together, which saves a significant amount of computation and memory.

The resulting octree can then be converted to a legolised representation by using the leaf nodes. Each white leaf node corresponds to a one in the legolised representation. Therefore, by traversing the octree, one can easily fill in all the ones in the legolised representation.

3.2.1 Fast triangle-cube intersection test

We used a fast 3D triangle-box intersection method developed by Akenine-Möller [15]. The method was derived from the separating axis theorem (SAT) [15]. The theorem states that, given two convex objects, if one can find an axis along which the projection of the two objects does not overlap, then the objects themselves do not intersect. In 2D, the possible separation axes are only the axes parallel to the normals of each face. In 3D, the two convex objects do not intersect if they can be separated along an axis parallel to a normal of a face from either of the two objects, or along an axis formed by the cross product [21] of two edges, with one edge from each object. If the cross products are not used, certain edge-on-edge non-intersecting cases would be treated as intersecting. Figure 3.4, page 36, gives an example of SAT for two rectangles in 2D.

In implementations of the triangle-box intersection test, it is prudent to first ensure that the box is axis-aligned to be able to use this method. Otherwise a full polygon-triangle intersection test must be done, which would require significantly more tests.

The axis-aligned box (AAB) is defined by a center $c$, and a vector of half lengths $h$. A half length is similar to a radius, except that it is defined along a specific direction (the unit vectors), instead of along all directions. The triangle is defined by three vertices
(a) (b)  

Figure 3.4: There are two possible separating axes for the rectangles. Example (a) shows that the rectangles do not intersect, as the projected rectangles on the bottom axis do not overlap. In example (b) the projections on both axes overlap and therefore the rectangles are intersecting. [10]

$u_0$, $u_1$, $u_2$, a normal $n$, and has the three edges $f_1$, $f_2$, and $f_3$. To further simplify the projection tests, the AAB and triangle is translated so that the AAB is centered around the origin. To translate the triangle, the center of the AAB can simply be subtracted from each of its vertices: $v_i = u_i - c$. Figure 3.5, page 37, shows the notation used for the AAB and the triangle, and how the AAB and triangle are translated to the origin.

After the triangle and AAB have been translated, there are thirteen tests that have to be performed. These can be grouped into three groups:

1. Test the normals of the AAB as possible separating axes for triangle-box projection overlap. Even though there are six normals (one for each face of the AAB), the three unit vectors $e_0 = (1,0,0)$, $e_1 = (0,1,0)$, and $e_2 = (0,0,1)$ can be used instead, as they will represent the same possible separating axes. Akenine-Möller simplified this test by, instead of calculating the projections of the AAB faces and the triangle edges onto the axes, rather testing the AAB and triangle directly for the overlap. This is done by using a simple interval test. If any of the vertices lie inside the cube, they intersect. If all the vertices of the triangle are on one side of
the cube, they do not intersect. If vertices are on opposite sides of the AAB, they intersect, as they cut through the AAB. This group requires three tests in total.

2. Test the normal of the triangle as a possible separating axis. Akenine-Möller simplified this test by, instead of calculating the projections onto the axis given by the normal, making use of a fast plane-AAB overlap test. The test uses only the two diagonal vertices whose distance from the triangle is the minimum and maximum. By calculating the dot product of these vertices with the normal of the triangle, one can determine whether the AAB intersects the plane of the triangle. If the dot product of the minimum distance vertex is larger than zero, the box cannot intersect with the plane, as the box is above the triangle plane. If it is smaller or equal to zero, the box will intersect with the triangle only if the dot product of the maximum distance vertex and the normal of the triangle is greater of equal to zero (see Figure 3.6, page 38). This group hence only requires one test.

3. Test the cross product between each triangle edge and the unit vectors as a possible separating axis. This requires calculating the three triangle edges, calculating their cross products with the normal of the triangle, and then projecting the vertices of the triangle and the AAB onto the axis given by their cross products. The triangle
CHAPTER 3. 3D REAL-WORLD OBJECT REPRESENTATION

Figure 3.6: Test to see if AAB intersects with the triangle plane. Calculate the minimum ($v_{min}$) and maximum ($v_{max}$) distance vectors from any triangle vertex to the AAB. Calculate the dot product between each of these vectors and the normal of the triangle. If the $v_{min}$ gives a value greater than zero, the AAB will not intersect as it is above the triangle plane. If it is smaller or equal to zero, use the $v_{max}$ dot product to determine whether the AAB cuts the triangle plane.

edges are calculated by: $f_0 = v_1 - v_0$, $f_1 = v_2 - v_1$, and $f_2 = v_0 - v_2$. The cross products are calculated as: $a_{ij} = e_i \times f_j$, $i, j \in \{0, 1, 2\}$. All the triangle vertices are projected onto the plane by calculating the dot product between the corresponding cross product and the vertex. The AAB is projected onto the cross product by calculating a “radius”: $r = h_x|a_x| + h_y|a_y| + h_z|a_z|$. The minimum and maximum of the three projected triangle vertices are calculated. If the minimum of the projected vertices is greater than $r$, or the maximum of the projected vertices is less than $r$, the triangle and the AAB does not overlap. This group of requires nine tests in total.

If all the tests pass, the triangle intersects the AAB since no separating axis exists.

We implemented the 3D triangle-AAB intersection test method to determine which triangles intersect with an octree node. In all our experiments, the method achieved perfect accuracy in detecting all triangle-cube intersections. However, if the triangle has an area close to zero, the normal calculation of the triangle is not robust and would cause incorrect intersection results. Triangles with such small areas rarely exist in 3D geometric models and therefore, we feel that the significant speed increase over other intersection
methods available, greatly outweighs the possibility of misinterpreting a few triangles.

The voxelisation method constructs a hollow outer shell of the 3D geometric model. In most cases, this will cause the resulting LEGO sculpture to be disconnected and fragile. Therefore, we implemented two methods that increase the strength and connectedness of the LEGO sculpture by altering the legolised representation.

### 3.2.2 Solidifying the legolised representation

We implemented a method that fills the inside of the legolised representation to ensure a strong and connected legolised representation. The method simply runs through all the values and determines whether they represent a point inside or outside of the object. If the point represents a point inside the object, it is set to a one.

To test whether a point is on the inside or the outside of an object, we use an approach similar to ray-tracing. We cast rays horizontally and when they reach the object, we know that they have entered the object. For each subsequent zero value, we determine whether it is surrounded by ones in all six main directions in 3D (forward, backward, left, right, up, and down). If a value is surrounded, it is set to one. If the value is not surrounded, the ray has exited the inside of the object. The ray continues this process until it reaches the end of the legolised representation (see Figure 3.7, page 40).

This method successfully fills the interior of the legolised representation. It does, however, require that there are no holes in the legolised representation. This will only happen if the 3D geometric model did not form an enclosed object.

### 3.2.3 Hollowing out the legolised representation

Building a solid LEGO sculpture in general requires substantially more LEGO bricks than building a hollow LEGO sculpture. Therefore, we implemented a method to hollow out the solid legolised representation whilst ensuring that it stays connected.
Figure 3.7: Three rays used to test whether a voxel is inside or outside of the model. The two voxels marked with an ‘X’ is inside the model, as they are surrounded in all directions.

The method hollows out the solid legolised representation to deliver at least a minimum outer shell thickness specified by the user. This gives the user some control over the strength of the resulting LEGO sculpture, as a thicker outer shell will generally deliver a stronger LEGO sculpture. Some LEGO sculptures require wider outer shells, while others will be strong enough using only a thin outer shell. A sphere, for instance, would require a much wider shell thickness than a pyramid.

The method uses the same ray-tracing technique as in Section 3.2.2, page 39, to determine whether a point is inside the object. If the point is inside the object, the method determines whether it is surrounded by at least the minimum shell thickness in all six directions. If it is surrounded, we check whether the layers above and below will be connected to the current layer, with at least the minimum shell thickness, if we remove the point. If the layers will still be connected if the point is removed, the point is set to zero. If the object is smaller than the minimum outer shell thickness specified by the user, no changes will be made to the legolised representation.

3.2.4 Results

In this section we will only produce results for our complete voxelisation application (as described in Section 3.2, page 33). The other 3D geometric model conversion methods
produced too many unwanted holes and artifacts, to give any meaningful information. The method using the Jgeom [5] Java 3D geometric library required significant execution time, even for small models (15 layers). Models consisting of a large number of triangles could easily require an hour or more to execute. The ray-tracing method required the use of more than one application to produce the final legolised representation. Therefore, the time required to convert the 3D geometric model could not be measured accurately. To create the 2D images of each layer only required a few seconds to compute, even for relatively large models (128 layers).

Our voxelisation method is extremely fast, only requiring a few seconds to compute a high quality legolised representation from a 3D geometric model. When voxelising a 3D geometric model, we are interested in comparing different sculptures for the same number of layers. If we voxelise the minimum AAB with a given resolution, different models will not necessarily consist of the same number of layers. Some 3D geometric models can have a height that is smaller than its width or depth, which will result in the number of layers being less than the resolution used. For instance, it is possible to use a resolution of $64^3$ and only find 10 layers. The width or depth will consists of 64 voxels, but not the height. Therefore, in our voxelisation application, the user specifies the number of layers wanted and not the resolution of the voxel partitioning. The application will use the appropriate resolution to ensure that the specified number of layers is given. Note that the number of voxels in the width or depth of the model can therefore be significantly larger than the number of layers specified.

Tables 3.1 to 3.3, on pages 42 to 43, show the execution time required to voxelise a selection of 3D geometric models\(^1\) by specifying different numbers of layers, filling the inside of the model, and hollowing out the filled model. Tables 3.4 to 3.6, on pages 43 to 44, show the number of filled voxels that were created in all these voxelisations. Figure 3.8, page 45, demonstrates how the level of detail improves as the number of layers are increased for the Aphrodite statue 3D geometric model.

\(^1\)See Figure A.1, page 98, for images of these 3D geometric models.
CHAPTER 3. 3D REAL-WORLD OBJECT REPRESENTATION

<table>
<thead>
<tr>
<th>3D Model</th>
<th>Triangles</th>
<th>Number of layers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>Chess pawn</td>
<td>527</td>
<td>0.140</td>
</tr>
<tr>
<td>T-Rex</td>
<td>2839</td>
<td>0.078</td>
</tr>
<tr>
<td>Aphrodite statue</td>
<td>12989</td>
<td>0.250</td>
</tr>
<tr>
<td>Dragon</td>
<td>23979</td>
<td>0.579</td>
</tr>
<tr>
<td>Woman</td>
<td>78043</td>
<td>1.421</td>
</tr>
<tr>
<td>Bunny</td>
<td>83694</td>
<td>1.500</td>
</tr>
</tbody>
</table>

Table 3.1: The influence of the number of layers on the execution time (in seconds) for different 3D geometric models without altering the voxel model.

<table>
<thead>
<tr>
<th>3D Model</th>
<th>Triangles</th>
<th>Number of layers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>Chess pawn</td>
<td>527</td>
<td>0.172</td>
</tr>
<tr>
<td>T-Rex</td>
<td>2839</td>
<td>0.125</td>
</tr>
<tr>
<td>Aphrodite statue</td>
<td>12989</td>
<td>0.266</td>
</tr>
<tr>
<td>Dragon</td>
<td>23979</td>
<td>0.625</td>
</tr>
<tr>
<td>Woman</td>
<td>78043</td>
<td>1.469</td>
</tr>
<tr>
<td>Bunny</td>
<td>83694</td>
<td>1.593</td>
</tr>
</tbody>
</table>

Table 3.2: The influence of the number of layers on the execution time (in seconds) for different 3D geometric models when the legolised representation is filled inside.

These tests show that the voxelisation process is significantly faster than the other methods with which we experimented. Models containing a large number of triangles only required a few seconds to calculate. Filling or hollowing out the inside of the voxelised model significantly influences the execution time, with the larger models requiring several minutes rather than the seconds required without altering the voxelised model. These execution times are still relatively small when considering the size of the final LEGO sculpture that will be created. The typical LEGO enthusiast will seldom build LEGO sculptures of that magnitude. The large sculptures consisting of 512 layers is approximately four meters high when built in LEGO. Therefore, we feel that our voxelisation method is sufficient for our purpose.
### Table 3.3: The influence of the number of layers on the execution time (in seconds) for different 3D geometric models when the *legolised* representation is hollowed out to an outer shell thickness of four voxels.

<table>
<thead>
<tr>
<th>3D Model</th>
<th>Triangles</th>
<th>Number of layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess pawn</td>
<td>527</td>
<td>0.281 1.594 20.609 369.500</td>
</tr>
<tr>
<td>T-Rex</td>
<td>2839</td>
<td>0.171 0.891 8.000 103.437</td>
</tr>
<tr>
<td>Aphrodite statue</td>
<td>12989</td>
<td>0.266 0.547 2.750 29.187</td>
</tr>
<tr>
<td>Dragon</td>
<td>23979</td>
<td>0.656 1.547 8.235 78.500</td>
</tr>
<tr>
<td>Woman</td>
<td>78043</td>
<td>1.500 3.109 17.094 196.031</td>
</tr>
<tr>
<td>Bunny</td>
<td>83694</td>
<td>1.625 4.515 39.953 820.969</td>
</tr>
</tbody>
</table>

### Table 3.4: The total number of filled voxels created from different 3D geometric models for various numbers of layers without altering the voxel model.

<table>
<thead>
<tr>
<th>3D Model</th>
<th>Triangles</th>
<th>Number of layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess pawn</td>
<td>527</td>
<td>14463 59539 241131 969200</td>
</tr>
<tr>
<td>T-Rex</td>
<td>2839</td>
<td>7021 28560 115218 462800</td>
</tr>
<tr>
<td>Aphrodite statue</td>
<td>12989</td>
<td>11078 46422 192029 782902</td>
</tr>
<tr>
<td>Dragon</td>
<td>23979</td>
<td>13467 53956 215793 862840</td>
</tr>
<tr>
<td>Woman</td>
<td>78043</td>
<td>15099 61575 249049 1002853</td>
</tr>
<tr>
<td>Bunny</td>
<td>83694</td>
<td>4060 16974 68921 278633</td>
</tr>
</tbody>
</table>

### Table 3.5: The total number of filled voxels created from different 3D geometric models for various numbers of layers with the inside of the model filled.

<table>
<thead>
<tr>
<th>3D Model</th>
<th>Triangles</th>
<th>Number of layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess pawn</td>
<td>527</td>
<td>42508 314311 2407895 18853777</td>
</tr>
<tr>
<td>T-Rex</td>
<td>2839</td>
<td>15211 105912 786400 6052402</td>
</tr>
<tr>
<td>Aphrodite statue</td>
<td>12989</td>
<td>15829 96554 656737 4795694</td>
</tr>
<tr>
<td>Dragon</td>
<td>23979</td>
<td>55659 416294 3176770 24879121</td>
</tr>
<tr>
<td>Woman</td>
<td>78043</td>
<td>33571 235739 1758478 13561396</td>
</tr>
<tr>
<td>Bunny</td>
<td>83694</td>
<td>6031 39152 2796680 2071354</td>
</tr>
</tbody>
</table>
Table 3.6: The total number of filled voxels created from different 3D geometric models for various numbers of layers with the inside of the model hollowed out to an outer shell thickness of four voxels.

<table>
<thead>
<tr>
<th>3D Model</th>
<th>Triangles</th>
<th>Number of layers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>Chess pawn</td>
<td>527</td>
<td>27062</td>
</tr>
<tr>
<td>T-Rex</td>
<td>2839</td>
<td>12454</td>
</tr>
<tr>
<td>Aphrodite statue</td>
<td>12989</td>
<td>14263</td>
</tr>
<tr>
<td>Dragon</td>
<td>23979</td>
<td>28423</td>
</tr>
<tr>
<td>Woman</td>
<td>78043</td>
<td>25604</td>
</tr>
<tr>
<td>Bunny</td>
<td>83694</td>
<td>5790</td>
</tr>
</tbody>
</table>

In summary, we started this chapter by defining a legolised representation as a 3D matrix consisting of zeros and ones. Since 3D scanners and other 3D reconstructive equipment are expensive, we decided to use 3D geometric models to represent the real-world objects. 3D modelling software and 3D geometric models are freely available giving the user a wide range of objects to construct. The advantages of using 3D geometric models are that they require minimal disk space and can easily be scaled to any size.

We discussed two exploratory methods to convert the 3D geometric model to a legolised representation. The first method used ray-tracing, while the second method used triangle subdivision, to calculate the intersection between the 3D geometric model and a horizontal plane. Both methods did not provide sufficient accuracy, producing holes and unwanted artifacts. Both methods required a substantial amount of time when converting 3D geometric models containing a large number of triangles. Finally, we discussed our voxelisation application using a triangle-cube intersection test to partition the 3D geometric model, which is significantly faster than the other methods implemented and delivers extremely accurate results.

In the next chapter we will explain our implementation of a beam search method to solve the LEGO construction problem.
Figure 3.8: (a) A 3D geometric model of a statue; and the voxelised models with (b) 64, (c) 128, and (d) 256 layers.
Chapter 4

Beam search

At the 2005 Brickfest conference, Winkler [39] presented a beam search technique to solve the LEGO construction problem. In Section 2.4.5, page 25, we discussed his beam search method to create LEGO building instructions from 3D geometric models. The method seemed promising, but no results were given that could be directly compared to other implemented methods. Therefore, in this chapter we will present our beam search implementation using his method as a basis (for the original description, see [39]).

4.1 Method overview

We construct the LEGO sculpture layer-by-layer, with each layer using the layers directly above and below it to help ensure a connected and stable LEGO sculpture. Each layer in the legolised representation is built using a beam search tree. Each node in the tree corresponds to a possible brick layout for the layer being constructed. The layout can be empty, partially filled, or complete.

The beam search starts with one current node as the root of the tree. The root node is the empty layout for the layer. It then generates all the possible successor layouts to the current node. The successors are generated by calculating all the possible layouts that
can be constructed from the current node, by adding a single brick to it. A cost function is used to evaluate the quality of each successor. The $k$ best successors are then selected and added to the tree. These nodes form the new set of current nodes.

The method continues by generating all the successors to the new current nodes. The best $k$ successors over all the successor nodes generated are added to the tree. The beam search stops when no more successors can be generated.

Note that the beam search restricts the width of the tree to $k$ nodes. The depth of the search tree is limited by the number of ones in the layer that must be filled. In the worst case each one can be filled with a generic brick. If larger bricks are used, the depth will be less. Therefore, the worst case space complexity is $O(n \times k)$, where $n$ is the number of ones in the layer to fill, and $k$ is the maximum number of nodes per depth (width of the tree).

When the beam search is complete, all the completely filled leaf nodes must be considered. Due to the restricted tree width, it is possible that some of the leaf nodes are not completely filled. The best possible brick layout for the layer is the completely filled leaf node with the smallest cost.

Figure 4.1, page 48, is an example of a beam search tree constructed to fill a $3 \times 3$ layer, using the standard LEGO brick set (Figure 2.3, page 7). The width of the tree is limited to four nodes; that is, $k = 4$. The cost of the layouts have been omitted to simplify the example. The beam search method starts with only the root node, an empty layout. The method generates all the possible brick placements for the root node which fill the first unfilled cell (one). The four best brick placements are selected and added to the root node as children nodes. These layouts represent the placement of a $2 \times 3$, $3 \times 2$, $2 \times 2$, and a $1 \times 3$ brick respectively. In the next step all the possible brick placements for each of the new layouts in depth one are generated. The four best brick placements over all the brick placements are selected and added to their respective parent nodes. Note that due to the restricted width of tree, the $2 \times 2$ layout in depth one has no children nodes. Therefore, the node is an unfilled leaf node. Three of the four nodes in depth two are
completely filled layouts. Since there is only one possible brick placement for the last remaining unfilled node in depth two, it is simply added to the tree. The complete beam search tree has therefore been constructed. There are four completely filled leaf nodes and one unfilled leaf node.

![Beam search tree](image)

**Figure 4.1**: Beam search tree constructed to fill a $3 \times 3$ layer, using the standard LEGO brick set. The width of the tree was restricted to four nodes ($k = 4$).

### 4.2 Generation of successor nodes

Successor nodes are generated by calculating all possible layouts that can be constructed from the parent node layout, by adding a single brick to it. If the new brick can be placed anywhere in the layout, the number of possible brick layouts are extremely large. One
would have to test all possible placements for each brick type, which includes rotating the bricks as well.

Therefore, to limit the number of nodes generated, we number all the ones in the layer. When a successor is generated, we fill in the lowest numbered unfilled one. This significantly reduces the number of possible successors, as we only need to find which of the bricks can be placed into the layout such that it will cover the lowest numbered unfilled one.

The exact numbering of the ones is an important factor. Winkler [39] simply numbered the ones in order from top-to-bottom, left-to-right. The LEGO company engineers mentioned [34] that when they would construct a LEGO sculpture by hand, they would normally construct the outer edges first and then fill the inside of the sculpture. Therefore, we implemented both the top-to-bottom, left-to-right in order method, and an edge first method. The edge first method numbers the outer edges in clockwise order starting from the top left. The inside is then numbered from top-to-bottom, left-to-right. Figure 4.2 shows how a legolised layer of ones will be numbered using both numbering methods.

![Figure 4.2](image)

**Figure 4.2**: (a) Layer of ones and zeros to number. (b) Top-to-bottom, left-to-right in order numbering. (c) Edge first numbering.

It is important to note that the ones will not always be filled in the exact order they are numbered. When a brick is placed it will, if not a generic brick, cover multiple ones in the layer. Therefore, the numbering only serves as a guide.

Both numbering methods however, have a disadvantage of potentially forcing some bricks
to be disconnected from the sculpture. The problem occurs due to “dangerous” ones. Dangerous ones are ones that do not have any ones in the layers directly above or below it. Therefore, these ones have to be filled such that the resulting brick would connect to at least one of the layers above or below it. When using either of the numbering methods to guide the filling process, it can potentially force the dangerous ones to be disconnected by not leaving it enough unfilled ones to connect with one of the layers above or below.

Figure 4.3 gives an example of how both the numbering methods can potentially create disconnected bricks from the layouts in Figure 4.2, page 49. The dangerous ones are \{9,18\} for the in order numbering method and \{5,7\} for the edge first numbering method. For both numbering methods, larger bricks are used early in the building process. When the algorithm has to fill the dangerous ones, there is not enough space to connect the dangerous ones to the sculpture.

Although making the width of the tree larger would help, it is not a practical solution when constructing large sculptures. Therefore, we changed both methods to first number all the dangerous ones and then to number the remaining ones. This will give the dangerous ones a better chance to be connected to the sculpture. Figure 4.4, page 51, shows both numbering methods with the dangerous ones numbered first. It also shows two possible brick layouts that can be found by using the numbering methods. The exact layouts will depend on the width of the tree and the cost function used.
CHAPTER 4. BEAM SEARCH

Figure 4.4: Numbering methods with dangerous ones numbered first. (a) In order numbering. (b) Edge first numbering. Both images show possible brick layouts. The exact layout will depend on the cost function and tree width.

4.3 Cost function

We use a global cost function to determine the cost of a layout. The cost function corresponds to that used by Petrovic [30] (see Section 2.4.4, page 17), except that we did not use the *neighbour* heuristic. We required a cost function that could be calculated efficiently without the need to recalculate the entire layout cost. The speed of calculating the layout cost is extremely important, as the cost for all the possible successor nodes must be calculated in order to find the $k$ best successors. It is important to note that, although the width of the tree is limited to $k$ nodes, all possible successor nodes must still be calculated at each step.

The *neighbour* heuristic can only be used to calculate the cost of a completely filled layout, as it requires all neighbouring bricks to already be placed to give a meaningful quality measure. Although it could potentially make a difference to which nodes (layouts) are selected, we feel that it adds unnecessary computation, since we already restrict the way bricks are placed by using the layout numbering methods. The effect on brick placement will therefore be minimal.

By not adding the *neighbour* heuristic to the cost function, we have a single-brick cost function. Since each new brick only uses the layers directly above and below as a cost measure, we can simply add the cost of placing the new brick to the old layout cost. This significantly lowers the amount of computation that has to be performed.
The cost function we use is given by

\[
Cost = C_{\text{numbricks}} \times \text{numbricks} + C_{\text{perpend}} \times \text{perpend} \\
+ C_{\text{edge}} \times \text{edge} + C_{\text{uncovered}} \times \text{uncovered} \\
+ C_{\text{otherbricks}} \times \text{otherbricks},
\]

where the \( C \)'s are weight constants and

- the \text{numbricks} variable is the number of bricks in the sculpture. Note that all bricks have the same cost, and hence larger bricks will implicitly be used where possible;

- the \text{perpend} variable corresponds to the directionality of the bricks in consecutive layers. This corresponds to Heuristic 3 in Section 2.3, page 10;

- the \text{edge} variable represents how many of the edges of each brick lies at the same location as that of bricks from the previous layer;

- the \text{uncovered} variable describes how much of the area of each brick is not covered by bricks in the previous and following layers; and

- for a given brick, the \text{otherbricks} variable represents the number of bricks in the previous layer covered by this brick. By using this variable, bricks are placed to cover as many bricks as possible in the previous layer, and therefore increase the overall stability of the sculpture.

### 4.4 Pruning the tree

If one does not restrict the width of the tree, the tree will be complete with all the leaf nodes being completely filled layouts. However, by restricting the width, one can find leaf nodes which are not completely filled. Such a leaf node is created when none of its possible successor nodes are in the set of \( k \) best successors. The layout will therefore
not be explored any further and can be removed from the tree to save space. These leaf nodes are removed immediately after the $k$ successors for the depth have been added to their corresponding parents in the tree.

A node is pruned from the tree by recursively moving up the tree towards the root node. The node is removed from tree by removing it from its parent node. If the parent node has no more children, it has to be removed as well. This process is repeated until a parent node is found which has remaining children nodes. Figure 4.5 gives an example of a node that can be pruned due to having no children nodes in the following layer\(^1\). In the example the leaf node and its parent node is pruned from the tree, since its parent has no more children once the node has been removed.

![Figure 4.5](image)

**Figure 4.5**: A leaf node representing an unfilled layout can be pruned from the tree. (a) A beam search tree with an unfilled leaf node. (b) The beam search tree after pruning.

When constructing the beam search tree, one can find different brick layouts that fill the exact same ones in the layer. In other words, the layouts cover the exact same area. These duplicate layouts can differ only in terms of which bricks were used, and the placement of the bricks\(^2\). When the tree width is small, duplicate layouts would not necessarily occur. However, when the tree width is large, duplicate layouts can be expected.

When constructing the beam search tree without restricting the tree width, duplicate layouts can be seen as redundant, since only the lowest cost layout is needed. If two layouts cover the same area of ones, both layouts have exactly the same possible brick

---

\(^1\)We do not show the layouts of the nodes or the cost as this is not important to the pruning method.

\(^2\)Hence duplicate layouts can have different cost function values.
placements to fill the remaining area of the layer. Figure 4.6, page 55, shows two duplicate layouts covering the first two rows of a $3 \times 3$ layout, and the possible brick placements to fill the remainder of the layout. If the bricks used in the duplicate layouts are ignored, the remaining brick placements are exactly the same for both layouts. Therefore, due to the single-brick cost function, the cost of filling the remaining part of the layer will be the same for both duplicate layouts. As the best possible complete layout is wanted, only the successors for the layout with the best cost so far has to be explored. The other layout cannot deliver a better result. If the tree width is restricted, pruning duplicate nodes could have a negative effect as there are no guarantee that either of the nodes will reach a completely filled layout. It is possible that the lower cost node will be pruned from the tree due to the restricted width, whereas the higher cost node will not be pruned. However, we feel that it is reasonable to assume that we can still prune duplicate layouts from the restricted width beam search tree. Duplicate layouts will usually have a node which is higher up in the tree. Therefore, containing less bricks and thus should also have the lower cost. Since the node is still in the tree, it has a far greater chance of reaching a complete layout than the new duplicate node. Therefore, we prune duplicate layouts in our beam search tree.

We use a hash table to keep track of each unique layout in the tree. When we want to add one of the $k$ new successor nodes to the tree, we first use the hash table to check whether the layout already exists in the tree. If the layout does not exist in the tree, the node is simply added to the tree. If the layout already exists, we remove the layout node with the higher cost. Therefore, if the new successor layout cost is greater than the existing layout node in the tree, it is simply not added to the tree. If the successor cost is equal to the existing layout node, the successor is not added to the tree, since the existing layout node will consist of less bricks as it has a higher depth\textsuperscript{3} in the tree. If the successor has a lower cost, the existing layout node is pruned from the tree. This entails removing the entire subtree starting from the existing layout node. Note that, by removing the entire subtree, it is possible that some of the $k$ best successors in the

\textsuperscript{3}A node with a higher depth is closer to the root node of tree.
Figure 4.6: Two duplicate layouts and their possible brick placements to fill the remaining area of the layout. If the bricks in the duplicate layers are ignored, the possible brick placements are exactly the same for both layouts.

Current depth must also be removed. The parent node of the existing node must also be deleted if it has no more children.

Figure 4.7, page 56, gives an example of a duplicate layout with a lower cost than the existing layout node. The entire subtree starting from the existing layout node is pruned from the tree\(^4\). The duplicate nodes are indicated in red and blue. The blue node is the lower cost layout that has to be inserted into the tree. The subtree starting from the red node is removed from the tree. Its parent node is not removed from the tree as it has one remaining child node. The new successor node (blue) is then added to the tree.

By pruning the unfilled leaf nodes and the duplicate layouts from the beam search tree, the final beam search tree results in a linked list. This is due to keeping only one unique complete layout in the tree. The linked list represents the building instructions for the lowest cost complete layout.

\(^4\)We do not show the layouts of the nodes or the cost as this is not important to the pruning method.
4.5 Data structures used

In this section we briefly discuss the main data structures used in our beam search application.

The tree node structure contains:

- the current layout cost;
- a reference to its parent node to allow efficient pruning of nodes;
- a list of edges to children nodes. An edge contains the brick that was added to the current layout and a reference to the resulting child layout. Instead of keeping a list of all the bricks used to construct a layout in each node, we store the bricks in edges. This saves a significant amount of space, since each brick is only stored once for a given subtree. The list of edges is sorted from lowest to highest cost automatically, as the children are already sorted in the $k$ best successors; and
- a bitset which represents only the ones in the layer that has to be filled. The bitset corresponds directly to the numbering of the ones in the layer. Therefore, if a bit has a value of one, the correspondingly numbered one is filled, and otherwise it is
unfilled. By only representing the ones in the layer with the bitset, we save a significant amount of space, since sculptures are generally hollow and would therefore contain a large number of zeros. Large sculptures can typically contain more zeros than ones, since the edge thickness is small in comparison to the dimensions of the layer.

In addition to the tree node and edge data structures we required the following data structures:

- two bitsets to represent the layers directly above and below the current layer being constructed. The bitsets are required to calculate the dangerous ones when numbering the ones in the current layer. It is also used to calculate the uncovered variable in the cost function;
- the list of the bricks used to construct the previous layer. This list is required to calculate the perpend variable in the cost function;
- a matrix representing the previous layer building instructions. The matrix can be seen as a grid, with each cell representing a zero or one from the original layer. Each cell contains a value corresponding to the brick covering it. If no brick covers the cell, it has a value of zero. Figure 4.8, page 58, gives an example of a layout of bricks constructed using the in order numbering method (with dangerous edges) and its corresponding matrix representation. The matrix is used to allow quick and easy cost calculation for the otherbricks and edge variables. If the matrix is not used, one would have to search through the list of bricks in the previous layer to find all the relevant bricks;
- a hash table to keep track of all unique layouts in the tree. We required a data structure that would allow fast duplicate layout detection and location. Although it is possible to search through the tree to determine whether a given layout already exists, it is impractical. One would potentially have to search through the entire tree, which requires $O(n)$ time, where $n$ is the number of nodes in the tree. By
CHAPTER 4. BEAM SEARCH

Figure 4.8: The matrix data structure representing the building instructions for the previous layer. (a) The bricks placed according to in order numbering method. (b) The cells in the matrix are numbered according to the brick covering it.

using a hash table we can detect whether a layout already exists and locate the corresponding node in $O(1)$ time. The hash table maps a given bitset, representing the ones filled in the layout, to the node in the tree that contains the same bitset. If there exist no such node in the tree, the hash table maps to nothing. Even though the hash table requires a substantial amount of space\(^5\), we feel that the significant speed advantage outweighs the need to minimise the space used;

- an array to store the order in which the ones must be filled. Therefore, if one must fill the bitset value at index $i$, one can find its $x$ and $y$ position in the layer by using the value stored at index $i$ in the array. The value stored at index $i$ is a single integer (combined position) value which represents the position as if the layer is numbered from top-to-bottom, left-to-right including all zeros. Note that this does not correspond to the numbering methods which only number ones. This method numbers every value in the layer. The $x$ and $y$ positions can be calculated by simply dividing the value by the dimensions of the layer. This gives us a $O(1)$ mapping of an index in the bitset to a position in the layer; and

- a hash table to map the $x$ and $y$ position of a one to an index in the filling order array. When testing whether a brick can be placed in a given layout, one has to make sure that the brick only covers unfilled ones. This test has to be fast, as it forms a significant part of generating successor nodes. To test whether a brick can

\(^5\)The hash table requires $O(n)$ space where $n$ is the number of unique layout bitsets in the tree.
be placed, one must test every position in the layer which the brick will cover. If any of the positions are zeros or filled ones, the brick cannot be placed. Therefore, we use a hash table to map the combined $x$ and $y$ position value to an index in the filling order array. If the position represents a zero in the layer, it will map to nothing and hence be rejected as an invalid brick placement. If the position represents a one, it will map to an index in the array. By using the index we check whether the corresponding bit in the layout bitset is set. If it is not set, the one is unfilled, and otherwise it is filled.

In the next chapter we will discuss our proposed method that can be used to solve the LEGO construction problem. The method uses cellular automata and local optimisation to construct a LEGO sculpture. We will therefore not give any results of the beam search method in this chapter. The results will be discussed in Chapter 6 where we compare the beam search method to our proposed cellular automata method.
Chapter 5

Cellular automata with cell clustering\textsuperscript{1}

In this chapter we propose an alternative method for solving the LEGO construction problem, by using cellular automata (CA). The method differs from previous methods by using local optimisation instead of global optimisation. The method was developed to improve on previous methods by: reducing the amount of memory required, reducing the execution time required, and allowing multicoloured LEGO sculptures to be constructed.

We briefly cover the main concepts of cellular automata in Section 5.1 and introduce some new terminology required by our method in Section 5.2. We then discuss in detail how the LEGO construction problem can be encoded as a cellular automaton.

5.1 Background

In this section we will briefly discuss the main concepts of cellular automata required in this chapter. For more in-depth information, see [40].

\textsuperscript{1}The work in this chapter has previously been published in [37].
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. 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 Figure 5.1). 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, $c_{22}(t = 1) = c_{12}(t = 0) \oplus c_{32}(t = 0) = c_{12}(t = 0) = 1$.

Our initial attempts to encode the LEGO construction problem as a CA was algorithmically awkward, as one brick can span more than one cell in a CA. We therefore defined a variant of CA which allows for the clustering of cells in CA.

### 5.2 CA with cell clustering

Generally, a CA has a given number of cells, and the number of cells stay constant 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. Therefore we need to define a cluster of cells.

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

**Definition 5.1.** 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 5.2.** Let $C$ be a 2D CA. A cluster $B$ in $C$ is a set of cells from $C$ such that any cell $c_{ij}$ in $B$ is adjacent to at least one other cell $c_{km}$ in $B$.

It is convenient to define disjointed clusters:

**Definition 5.3.** Two clusters $A$ and $B$ are disjoint if there is no cell $a_{ij}$ in $A$ such that $a_{ij}$ is also in $B$, and no cell $b_{ij}$ in $B$ such that $b_{ij}$ is in $A$.

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 also define adjacency of clusters:

**Definition 5.4.** Two clusters $A$ and $B$ are adjacent if there exists at least one cell $a_{ij}$ in $A$ and at least one cell $b_{km}$ in $B$ such that $a_{ij}$ is adjacent to $b_{km}$.

**Example 5.1.** In Figure 5.2, page 63, cell $c_{11}$ is adjacent to cell $c_{12}$, but $c_{11}$ is not adjacent to $c_{22}$. Also, the set of cells $\{c_{12}, c_{21}, c_{22}\}$ 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 Figure 5.3.

**Definition 5.5.** A valid LEGO brick is one with dimensions as defined in Figure 5.3.\(^2\)

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). Therefore, we define the meaning of the neighbourhood of a cluster:

**Definition 5.6.** The (Von Neumann) neighbourhood of a cluster $A$ is the collection of clusters adjacent to $A$.

Note that the above definition implies that a cluster may not necessarily have four neighbours in its Von Neumann neighbourhood as with a conventional CA. Also, the

\(^2\)The valid LEGO bricks include the various rotations of the bricks.
number of Von Neumann neighbours may vary between different clusters. For example, in Figure 5.2, page 63, 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 5.7.** Let $A$ and $B$ be two clusters in a binary 2D CA. Define a new cluster $C$ to contain every cell $a_{ij}$ in $A$ and every cell $b_{km}$ in $B$, and no other cells. Then $C$ is said to be the merge of $A$ and $B$.

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

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

We construct the LEGO sculpture layer-by-layer. Each layer in the legolised representation forms a two-dimensional grid of clusters. 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. It is possible for two clusters to merge if the resulting cluster will represent a (larger) valid LEGO brick. 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 new 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 the merging of clusters in more detail.
5.3.1 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). 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.

When the method starts building the first layer, all clusters will have neighbours with equal cost. This is due to all neighbours being $1 \times 1$ clusters. By selecting a random neighbour to merge with, different starting layouts can be found. This can have a noticeable effect on the final solution. For instance, in the first time step all the clusters can merge together and form a layout very close to the optimal layout, or it could possibly form a solution far from the optimal layout. The solution far from the optimum layout will require many more time steps to reach the optimum layout, as the layout will have to split and merge numerous times to improve the layout.

If there are no clusters with which the current cluster can merge, the cluster can split into smaller clusters (see Section 5.3.2, page 70). 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:

\[
\text{Cost} = W_{\text{perpend}} \times \text{perpend} + W_{\text{edge}} \times \text{edge} + W_{\text{uncovered}} \times \text{uncovered} + W_{\text{otherbricks}} \times \text{otherbricks},
\]

(1)

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 [30], 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 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.

In the first step each cluster is visited and the potential new clusters are calculated by using the best possible merge status information. A potential cluster is a set of clusters that want to be merged together. This process is sequential.

In the second step all the potential clusters are merged together 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 a randomly selected 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. Note that it will not always be possible to merge a cluster into one valid larger brick. This is due to the local neighbourhood used. We therefore merge the bricks so that they form valid bricks. We note that this method of merging the clusters will not necessarily always deliver the most 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 5.2.** Suppose that a $3 \times 3$ grid, with all values consisting of ones, must be constructed from the standard LEGO brick set (see Figure 5.3, page 63). 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 Figure 5.2(b), page 69). 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 potential new clusters (see Figure 5.2(c), page 69).

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 three clusters instead of two, since the second cluster cannot merge into one large valid brick (see Figure 5.2(d), page 69). 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 Figure 5.5, page 70), 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.
CHAPTER 5. CELLULAR AUTOMATA WITH CELL CLUSTERING

Figure 5.5: The resulting bricks after time step 2.

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.

5.3.2 Splitting

Our splitting method simply splits a given cluster into $1 \times 1$ clusters (see Figure 5.6). 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 dissolvement of the cluster.

Figure 5.6: 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. We assign a different splitting probability to every brick (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. Note that the splitting of clusters are done in phase 2, and can be done in parallel while the potential clusters are merged together.

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. If a cluster is not able to merge with neighbouring clusters, it can split into $1 \times 1$ clusters in phase 2.

In the next section we will explain how this method can be extended to incorporate multicoloured LEGO sculptures.

### 5.4 Multicoloured LEGO sculptures

Our CA method can easily be extended to incorporate multicoloured LEGO sculptures, by assigning colours to each of the starting $1 \times 1$ clusters, and then restricting clusters
Clusters on the outer edge of the sculpture (the parts that are visible when the sculpture has been built) are assigned the colour of the corresponding section in the real-world object. The remaining clusters representing the inside of the sculpture are assigned a wildcard colour.

Clusters are restricted to merge only with neighbours of the same colour or with a wildcard cluster. When a colour cluster merges with a wildcard cluster, the resulting cluster will have the colour of the coloured cluster. When clusters are split into $1 \times 1$ clusters, the edge clusters are set to their original starting colour and the clusters inside the sculpture are set to wildcard clusters. Note that wildcard clusters do not have to merge with colour clusters. They can merge with each other to form larger bricks. Wildcard bricks represent bricks where the colour of the brick is not important.

By allowing wildcard clusters, the method can connect different coloured parts of the sculpture on the inside of the sculpture, allowing otherwise disconnected sculptures to be connected. We give an example of how colour can be added to our method below.

**Example 5.3.** Suppose that we want to construct a wall consisting of two colours as shown in Figure 5.7.

![Figure 5.7](image-url)  
*Figure 5.7: A three layer wall consisting of two colours.*

Each layer is represented by a grid of clusters. The clusters on the outer edge of the sculpture are assigned the colours corresponding to the real-world 3D object. The remaining clusters are set to wildcard clusters. Figure 5.8, page 73, shows the starting grids of clusters for the three layers. Note that since the bottom and top layers only
consist of outer edges, they have no wildcard clusters.

\[ \text{Figure 5.8: The grid of clusters representing (a) the first and last layers and (b) the second layer. Wildcard clusters are indicated in grey.} \]

Therefore, the second layer is the only layer that can connect the two coloured parts of the sculpture. Figure 5.9, page 74, shows the final layout for each of the layers. In the first and last layers the clusters merge together to form the largest bricks possible without merging clusters of different colours. In the second layer the wildcard clusters in the middle (11, 12, 13, 14, 19, 20, 21, 22) merge together to connect the different coloured bricks in the previous layer. The other wildcard clusters in the second layer merge with neighbouring clusters to form larger coloured bricks.

\[ \square \]

Although simply adding colour to the clusters as discussed above can deliver good results, we added an additional heuristic to the local cost function Eq. (1), page 66, to help ensure a connected sculpture. The new term \( \text{coverColour} \) encourages bricks to cover as many bricks of different colour in the previous layer as possible. Therefore, the local cost function for multicoloured sculptures is:

\[
\text{Cost} = W_{\text{perpend}} \times \text{perpend} + W_{\text{edge}} \times \text{edge} + W_{\text{uncovered}} \times \text{uncovered} + W_{\text{otherbricks}} \times \text{otherbricks} + W_{\text{coverColour}} \times \text{coverColour}.
\]
5.5 Data structures used

In this section we briefly discuss the main data structures used in our CA with cell clustering method.

We use a 2-dimensional matrix to represent the grid of cells. Each cell in the matrix has a reference to the cluster that fills the cell in the layout. Therefore, all the cells in a given cluster will reference the same cluster. To save memory, we only create clusters for cells representing ones in the legolised representation. Hence, cells representing zeros in the legolised representation contain no reference. The matrix is used to quickly determine the neighbouring clusters for a given cluster, by simply finding all the unique clusters referenced by the surrounding cells in the matrix.

Our cluster contains the following information:

- the brick type it represents. At the start of each layer all clusters represent $1 \times 1$ bricks. As clusters merge, they are replaced by new clusters with the brick type
corresponding to the new cluster shape;

- an \( x \)- and \( y \)-position in the grid of cells, as a reference point for the top left corner of the brick. This is required to find the cells in the matrix that have to be tested for neighbouring clusters;

- the colour of the brick. This is required for multicoloured LEGO sculptures. In our implementation we allowed 256 colours including the wildcard colour. At the moment LEGO bricks are only made in 53 different colours, therefore we allow enough space for future extension;

- the splitting time delay. This value is decreased for each time step where the cluster cannot merge with any of its neighbouring clusters. Once the value reaches zero, the cluster attempts to split; and

- additional status information:
  - a reference to the best merge neighbour. This is the status information that is set in phase 1 of the method to indicate with which neighbouring cluster the cluster wants to merge; and
  - a value indicating in which direction the best merge neighbour is located (left, right, top, bottom). This speeds up the merging process in phase 2, as we do not have to retest where the neighbour is and what brick type the merge result should be.

Instead of visiting each cell in the grid in phase 1 to determine the best merge neighbours for each cluster, we keep a list of all the clusters. As the clusters start to merge, the size of the list will decrease, since the number of bricks used must be minimised. Therefore, we only run through the small list of clusters instead of running through all the cells in each time step.

When multicoloured sculptures are built, we use a hash table to map outer edge cells to their original colour. The hash table is used when a cluster is split into \( 1 \times 1 \) clusters
to determine the colour of each cluster. Only the outer edge cells are stored in the hash table. If a cell does not map to a value in the hash table, it is a wildcard cluster.

In this chapter we proposed a new method to solve the LEGO construction problem using CA with cell clustering. We started by briefly explaining cellular automata and then defined a new variant of CA, namely, CA with cell clustering. We showed how the LEGO construction problem can be encoded as a CA and how it can be adapted to incorporate multicoloured sculptures.

In the next chapter, we analyse the results obtained by our implementation of the CA with cell clustering method. We also compare the results to our implemented beam search method.
Chapter 6

Results

In Chapter 4 we discussed our implementation of a beam search using the method as described by Winkler [39] as a basis. In Chapter 5 we discussed our cellular automata with cell clustering method. In this chapter we will compare these two methods, and discuss their advantages and disadvantages. We also point out some interesting issues that occurred.

6.1 Measures of quality

In comparing the cellular automata with cell clustering method against the beam search method, one is in essence comparing a local optimisation method against a global optimisation method. Therefore, we had to define measures of quality for 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 (see Section 2.1, page 4);
• cost (as defined by the cost function): as the cost function contains all the parameters against which to optimise, a low cost function value represents a “good” model;

• execution time of the implementation on a specific model: the quicker a good layout can be reached, the better;

• the memory usage: if less memory is required, larger LEGO sculptures can be constructed;

• extending the solution to allow multicoloured sculptures: previous solutions to the LEGO construction problem assumed the input to be monochrome, which significantly limits the practical use of the implementation;

• different brick sets: if the application can restrict the brick set that must be used to construct the LEGO sculpture, a user can remove brick types that is not available;

• 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 software solution.

In the following section we will discuss each of these measures of quality by comparing the advantages and disadvantages of each of the methods.

6.2 Experiments

All experiments were performed on an Intel Quad Core Q6600 2.4GHz processor with 2048mb system memory. Since our implementation of the beam search method has not
yet been parallelized, we restricted the CA to using only one core and one thread, for a fair comparison.

To effectively compare the two methods, we had to use the same cost function weights for both methods. However, we found that the cost function weights, especially the brick cost, can have a substantial effect on which method is better for a given sculpture. Therefore, we decided to use two different sets of cost function weights – one in which the cost of the number of bricks is low and the other where the cost is high. Although the other weights can also have an influence, the selected cost function weights produce good overall building instructions, and was therefore left the same. Therefore, we selected weight constants \( C_{\text{perpend}} = -200, \ C_{\text{edge}} = 300, \ C_{\text{uncovered}} = 5000, \) and \( C_{\text{otherbricks}} = -200 \) for both methods. The low brick cost weight was selected as \( C_{\text{numbricks}} = 200 \) and the high brick cost weight was selected as \( C_{\text{numbricks}} = 500 \).

We set up a number of experiments to evaluate our measures of quality. Nine different 3D geometric models were selected. These models cover the general features one would normally want to construct when building a LEGO sculpture, such as the arms, fingers, and facial features of a person. Table 6.1, page 80, lists the different 3D geometric models with the number of layers, and the minimum edge thickness used. Note that the cube 3D geometric model has been included twice, using different numbers of layers, to represent simple structure sculptures. We used a wider minimum edge thickness with the larger 3D geometric models to improve the strength of the LEGO sculpture. For a visual image of each of the 3D geometric models see Appendix A, page 98.

A summarised view of the results is given in Table 6.2 and Table 6.3, on pages 81 and 82 respectively. For detailed results see Appendix A, page 99 to 108. Since the tree width and the number of time steps required to deliver good results depend on the sculpture being constructed, we summarised the results according to the cost function values of the two methods being comparable. This helps us to determine which method delivers

\footnote{These values are similar to the values suggested by [30]. The large integer values prevent underflow errors in the evaluation of the cost function.}
### 3D Geometric Models and Their Properties

<table>
<thead>
<tr>
<th>3D Geometric</th>
<th>Number of layers</th>
<th>Edge thickness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cube</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Chess</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>Bunny</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>Cube</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>Max</td>
<td>64</td>
<td>4</td>
</tr>
<tr>
<td>Dzilla</td>
<td>64</td>
<td>4</td>
</tr>
<tr>
<td>Dino Vel</td>
<td>128</td>
<td>4</td>
</tr>
<tr>
<td>Woman Seated</td>
<td>128</td>
<td>4</td>
</tr>
<tr>
<td>T-Rex</td>
<td>256</td>
<td>6</td>
</tr>
<tr>
<td>Aphrodite statue</td>
<td>256</td>
<td>6</td>
</tr>
</tbody>
</table>

**Table 6.1**: The 3D geometric models, the number of layers, and the minimum edge thickness used in our experiments.

The solution the quickest and uses the smallest number of bricks, for similar cost function values. Note that 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.

#### 6.2.1 Number of bricks used

When using the higher brick cost (see Table 6.3, page 82), we found that the CA method delivered substantially fewer bricks in significantly less time than that of the beam search method. The beam search method required a much larger tree width and therefore a significantly longer execution time to deliver a similar number of bricks. When using the low brick cost (see Table 6.2, page 81) the CA delivered fewer bricks than that of the beam search, but not necessarily with the same cost function value. The experiments showed that to reach similar cost function values, the CA method required a significantly longer execution time. However, if the cost function value was ignored, the CA method delivered substantially less bricks in significantly fewer execution time.

This can be expected, as the CA method forces two neighbouring bricks to merge whenever possible, so that fewer but larger bricks are used. Therefore, although it uses the
### Table 6.2: Summary of results when using a low brick cost of $C_{numbricks} = 200$.

<table>
<thead>
<tr>
<th>3D Geometric Model</th>
<th>Number of Bricks</th>
<th>Cost function value</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BS</td>
<td>CA</td>
<td>BS</td>
</tr>
<tr>
<td>Cube</td>
<td>35</td>
<td>37</td>
<td>5999</td>
</tr>
<tr>
<td>Chess</td>
<td>1205</td>
<td>981</td>
<td>384220</td>
</tr>
<tr>
<td>Stanford Bunny</td>
<td>1375</td>
<td>1138</td>
<td>263529</td>
</tr>
<tr>
<td>Cube 32</td>
<td>3102</td>
<td>2128</td>
<td>247673</td>
</tr>
<tr>
<td>Max</td>
<td>1718</td>
<td>1413</td>
<td>317700</td>
</tr>
<tr>
<td>Dzilla</td>
<td>4441</td>
<td>3524</td>
<td>817255</td>
</tr>
<tr>
<td>Dino</td>
<td>15170</td>
<td>11919</td>
<td>2819299</td>
</tr>
<tr>
<td>Woman seated</td>
<td>29096</td>
<td>23372</td>
<td>5911255</td>
</tr>
<tr>
<td>T-Rex</td>
<td>72026</td>
<td>62029</td>
<td>10876886</td>
</tr>
<tr>
<td>Aphrodite statue</td>
<td>39409</td>
<td>32222</td>
<td>5772139</td>
</tr>
</tbody>
</table>

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 can lead 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.

Both the beam search method and the CA method show that by increasing the number of time steps or the tree width used, the number of bricks will decrease unless the best optimal solution has already been reached.
CHAPTER 6. RESULTS

<table>
<thead>
<tr>
<th>3D Geometric Model</th>
<th>Number of Bricks</th>
<th>Cost function value</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>BS</td>
<td>CA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BS</td>
<td>CA</td>
</tr>
<tr>
<td>Cube</td>
<td>32</td>
<td>16210</td>
<td>17526</td>
</tr>
<tr>
<td>Chess</td>
<td>1118</td>
<td>678753</td>
<td>679701</td>
</tr>
<tr>
<td>Stanford Bunny</td>
<td>1436</td>
<td>714223</td>
<td>571806.9</td>
</tr>
<tr>
<td>Cube 32</td>
<td>2823</td>
<td>1100148</td>
<td>871269</td>
</tr>
<tr>
<td>Max</td>
<td>1765</td>
<td>870479</td>
<td>704289</td>
</tr>
<tr>
<td>Dzilla</td>
<td>4620</td>
<td>2292700</td>
<td>1879168</td>
</tr>
<tr>
<td>Dino</td>
<td>15469</td>
<td>7643511</td>
<td>7679398</td>
</tr>
<tr>
<td>Woman seated</td>
<td>29366</td>
<td>14818642</td>
<td>14837608</td>
</tr>
<tr>
<td>T-Rex</td>
<td>71878</td>
<td>32520530</td>
<td>32181952</td>
</tr>
<tr>
<td>Aphrodite statue</td>
<td>38452</td>
<td>16690366</td>
<td>16496130</td>
</tr>
</tbody>
</table>

Table 6.3: Summary of results when using a high brick cost of $C_{\text{numbricks}} = 500$.

6.2.2 Cost function value

The experiments show that the cost function constants have a substantial influence on which method is better. When the number of bricks must be minimised (a high brick cost), the CA method delivers the best results in significantly less time. However, when the cost of bricks are low, both methods deliver comparable results. This can be expected, as the CA always tries to minimise the number of bricks as explained in Section 6.2.1, page 80, and hence the overall cost will be lower.

The results showed that when models are small or simple (not too many curves and detail), such as the cube and chess models, the beam search delivers excellent results. This can be expected, as the tree width required to reach good results will be substantially smaller than in larger sculptures. Small and simple models only have a limited number of possible layouts which are worth considering.

The beam search has a definite advantage when a user wants a specific feature to be more prominent. For instance, if only alternate directionality should be taken into account, the beam search will use mostly $1 \times 2$ bricks to maximise the number of bricks with alternate directionality. Although the CA method will take alternate directionality into account, it will still merge bricks together to minimise the number of bricks used.
6.2.3 Execution time

Reaching a good layout with the CA method depends on the number of time steps that the CA executes, whereas to reach a good solution with the beam search method depends on the width of the search tree (and how much it is pruned). These two parameters are not necessarily easily comparable. Due to the random element in the merging phase of the CA method, it is possible that the best solution is found almost instantly.

When the brick cost is low, the execution time of both methods are comparable for small simple 3D geometric models, but with larger models the CA method delivers sculptures of reasonable quality in substantially less time. When the brick cost is high, the CA method is significantly faster in all models except the small cube.

When models are small, the beam search tree can easily cover most building possibilities. However, when larger sculptures are constructed, it requires a much larger tree width. This significantly increases the memory usage and execution time required. The CA method on small models will quickly merge into large bricks and deliver a good starting solution. The remaining time steps can then be used to improve the cost of the layer. We found that, when the number of bricks is considered as the only measure of quality, the CA method requires substantially fewer time steps and therefore significantly less execution time to reach a good solution. However, if the cost function value is important, the CA method requires substantially more time steps to improve the solution.

6.2.4 Memory usage

The CA method has a significant advantage over the beam search method, when it comes to memory requirements. The beam search requires a substantial amount of memory as the width of the search tree and the size of the sculptures increase. The CA method always requires only the memory needed to store the grid of cells and the list of clusters. Therefore, the CA method can easily be used on a basic home computer to construct relatively large sculptures, whereas the beam search method will only be able to construct
small sculptures, or large sculptures of weak quality.

6.2.5 Multicoloured sculptures

The CA method has a distinct advantage over beam search when it comes to the extension of the implementation to cater for multicoloured sculptures.

In the case of the CA method, the inclusion of colour has only a small effect on the merging phase, where merging of clusters is then restricted to clusters of the same colour only (as explained in Section 5.4, page 71). Our current implementation of the CA method already incorporates multicoloured models. The beam search method, however, is much more difficult to extend to coloured models.

Traditionally, when building monochrome LEGO sculptures, an edge 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 help connect the sculpture and improve 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 the tree pruning can lead to disconnected sculptures. Therefore, we did not implement the inclusion of colour for the beam search method.

6.2.6 Ease of building

We measured the ease of building LEGO sculptures by constructing a few models by hand using the instructions generated by both methods. Due to the fewer bricks used by the CA method, larger models felt easier and quicker to construct than that of the beam search method. However, when simple models (models containing layers of similar shape) are constructed, the beam search method produced patterns which correspond to how one
would construct a LEGO sculpture by hand. In these cases one would prefer the building instructions generated by the beam search method. Figure 6.1 shows the difference in a few of the layers created by the beam search method and the CA method for the small cube. The beam search delivers the typical solution a person would construct, whereas the CA method does not follow the same patterns. Although the layouts produced by the CA method is similar to that constructed by the beam search method, in larger sculptures the brick patterns produced by the beam search method is much easier to follow when building the LEGO sculpture.

![Figure 6.1: The first four layers of the cube constructed by the (a) beam search method and the (b) CA method. The beam search delivers patterns generally constructed by a person for simple models.](image)

Although the brick layouts created by the two methods could differ substantially, there was no major difference in their stability. Therefore, we feel that individual users may show a preference for any of the methods.

### 6.2.7 Ease of implementation

Although the number of lines of code for each of our implementation is roughly comparable, it was much simpler to implement the CA method – even with merging and
splitting – than it was to implement the search tree with its associated pruning in the beam search method. Due to the significant amount of memory required by the beam search tree, more complicated data structures and algorithms had to be used to minimise the amount of memory used. Therefore, we sometimes had to sacrifice speed to ensure a lower memory usage. Since the CA requires significantly less memory, we could focus on decreasing the execution time.

6.2.8 Restricted brick sets

The beam search has the advantage that it can use any brick set, although it is recommended that the generic brick is included. If the generic brick is not included in the brick set, holes can occur due to the numbering method used when filling a layer. Although the beam search method allows any brick set to be used, there is no guarantee that the brick set will deliver a connected LEGO sculpture. For example, if one would only use the generic brick, the entire sculpture will be disconnected.

The CA method was developed with the standard LEGO brick set in mind. Therefore, although one could potentially further restrict the brick set, special care must be taken as the bricks must be able to merge together to form larger bricks. If there is no intermediate brick into which smaller bricks can merge, larger bricks will not be able to form. The CA method, by the definition of cells, has to include the generic brick.

Since the L-shaped brick is not common in LEGO brick sets, we have added the ability to remove it from the brick set in both methods. This will not have a significantly influence on the CA method, as there are numerous ways to merge smaller bricks into larger bricks without the L-shaped brick.

In summary then, we conclude that the beam search method and the cellular automata method deliver comparable results. When the brick cost is low, the beam search method tends to find lower cost sculptures substantially quicker than the CA method. However, when the cost of bricks are high, the CA method produces better overall results, using
fewer bricks, less execution time, and delivering lower cost sculptures. The only distinct advantage the beam search has over the CA method is that a user can specify a specific feature to be prominent, whereas the CA method will always merge bricks to minimise the total number of bricks used. The distinct advantages of the CA method are the ease of implementation, the significantly smaller memory usage, and its trivial extension to construct multicoloured LEGO sculptures.
Chapter 7

Conclusion and future work

7.1 Conclusion

The goal of this thesis was to develop a new improved approach to solving the LEGO construction problem and to create a complete LEGO sculpture construction package for use by LEGO enthusiasts.

In Chapter 2 we discussed the LEGO construction problem as stated by the LEGO company and showed how the LEGO construction problem has been simplified by Grower et al [20]. We briefly discussed some LEGO building terminology which we used throughout the thesis. We discussed various optimisation techniques that have been implemented or suggested by other researchers in order to solve the simplified LEGO construction problem.

In developing a complete LEGO sculpture construction package, the first aspect we addressed was how to create the input data required by the LEGO construction problem. Since 3D scanners and reconstruction equipment are expensive, we decided to use 3D geometric models to represent the real-world objects. 3D modeling software and 3D geometric models are freely available giving the user a wide range of objects to construct. The advantages of using 3D geometric models are that they require minimal disk space
and can easily be scaled to any size.

In Chapter 3 we discussed our implemented voxelisation application that converts a 3D geometric model into a legolised model. The application uses a fast triangle-box intersection test to partition the 3D geometric model into small cubes. The voxelised model is then directly converted to a legolised model. The voxelisation method eliminates the problem of determining the inside and the outside of a 3D geometric model, which produced unwanted holes and artifacts in our exploratory methods. The voxelisation application is significantly faster than the exploratory methods implemented and delivers extremely accurate results. Therefore, we successfully developed an application that allows users to create a legolised model for any real-world object by using 3D geometric models.

In Chapter 4 we discussed our implementation of a beam search method previously implemented by Winkler [39], to solve the LEGO construction problem. The method seemed promising, but no results were given by Winkler that could be directly compared to other previously implemented methods. Therefore, we implemented our own version using his method as a basis.

In Chapter 5 we discussed our new approach to solving the LEGO construction problem by using cellular automata with cell clustering. Previously implemented methods, including the beam search method, used global optimisation to create the LEGO building instructions. Our CA method makes use of local optimisation to find the optimal solution. The method models the LEGO construction problem by defining cells as LEGO bricks. Cells merge with neighbouring cells to form larger clusters (bricks) by using a local cost function. Since the number of cells in CA generally stays constant throughout execution, we had to define a new variant of CA with cell clustering. CA was used to reduce the extremely large memory usage of previous methods, and to significantly reduce the complexity of incorporating multicoloured sculptures.

Our experiments in Chapter 6 showed that both the beam search method and the CA method produce excellent results in reasonable time. The results for both methods were
comparable and therefore, we feel that individual users may show a preference towards any of the methods. The only advantage the beam search has over the CA method is that a user can specify a specific feature to be prominent, whereas the CA method, due to local optimisation, will always merge bricks to minimise the total number of bricks used. Since most users would want to minimise the total number of bricks used, we feel that this is not really a disadvantage to using the CA method. The advantages of the CA method are the ease of implementation, the significantly smaller memory usage to previously implemented methods, and its trivial extension to construct multicoloured LEGO sculptures which were previously too complex to construct.

We have therefore, in our opinion, succeeded in developing a new improved method to solving the LEGO construction problem, by using CA with cell clustering. We successfully developed a complete LEGO sculpture construction application and therefore accomplished our thesis goal. Our complete application is available as a sourceforge project, at http://bricksculpturer.sourceforge.net.

7.2 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.

We are planning to further parallelise the code of both methods 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 [23], we do not expect either method to improve upon the other by using parallelisation.

We have previously implemented a 3D CA with cell clustering, in order to construct all layers in the sculpture simultaneously, instead of layer-by-layer. The results were poorer than expected, but we intend to revisit this aspect in more detail. We believe the poor
results are due to our cost function being 2-dimensional. By using our 2D cost function, a layer tries to improve on the previous time step by using the layer below to measure its quality. When constructing the sculpture layer-by-layer this will be sufficient, as the previous layer does not change. Therefore, the layer has a constant reference point against which to judge its quality. However, when constructing all layers at once, the previous layer changes with each time step. Therefore, a layer uses outdated information to improve itself (CA always use the information in the previous time step), which could result in a weaker solution. The cost function should therefore be changed to incorporate transformations in the layers below and above.

Although we try to ensure that the final LEGO sculpture will be one connected object, by penalising bricks which are not connected to the sculpture, and by first filling the dangerous ones in the beam search method, it does not guarantee a connected sculpture. A sculpture can still contain bricks or sections that are not connected to the main sculpture. These parts might be impossible to connect due to the voxelisation resolution used or due to the restricted brick sizes available. Therefore, we feel further analysis must be done to determine beforehand whether it is possible to construct a connected sculpture from the legolised representation.

Another area that we feel need to be addressed, is to determine the structural strength and stability of the final LEGO sculpture. Due to the weight of LEGO bricks and the way that they are connected, it is possible that sections of the sculpture can break off due to its weight. For instance, the arms of a sculpture representing a man can break off due to the bricks connecting the arm to the torso being too small or too few to carry the weight of the arm. Although most LEGO enthusiasts use glue to ensure that the sculpture will not break, we feel it can be solved by strengthening the potential breaking points on the inside of the sculpture.

When considering the weight of the LEGO bricks, one has to consider whether the LEGO sculpture will be able to stand on its own without falling over. We have already done some experiments to try and determine whether a LEGO sculpture will be able to stand
and if not, how to improve its stability. Our method simply adds bricks to the inside of the LEGO sculpture to shift the center of mass towards the required position. The required position is approximated by finding the center of mass for only the first layer. Therefore, the center of mass for a man standing, would be between his two feet. For most LEGO sculptures this serves as a good approximation. Bricks are added to the sculpture using a greedy approach by placing a brick if they improve the overall center of mass. The bricks are placed as low as possible in the LEGO sculpture to lower the center of gravity, increasing its stability. Our experiments showed that although our method produces good results, it is far too slow for larger sculptures. Therefore, more research must be done in this area.

Currently the user must either download or create a 3D geometric model by hand using a 3D modeling application such as Blender [2]. Ideally the application should accept a series of synchronized photographs as input and generate the 3D geometric model on its own. We have not yet implemented such an application, but we did identify shape from silhouette [41] and shape from structured light [35] as two possible 3D reconstruction methods that should deliver the required level of detail.

Finally, we are planning to add the option of generating multicolour legolised representations to our voxelisation application. Currently multicoloured sculptures have to be constructed by hand, which is a cumbersome task. The colour of each voxel will be determined by calculating the average colour for all of the triangles that intersect the voxel. To ensure only true LEGO brick colours are used, these colours will be mapped to the closest colour in the LEGO brick colour palette.
Bibliography

[1] 3D meshes research database.


[3] Java 3D.


    https://jgeom.dev.java.net/.

    http://ldd.lego.com/.


    http://mysite.verizon.net/sfg0000/.


http://graphics.stanford.edu/data/3Dscanrep/.


http://www.brickfrenzy.com/?m=pooh.


http://citeseer.ist.psu.edu/714970.html.


http://www.idi.ntnu.no/grupper/ai/eval/brick/.

[22] Jessiman, J. *LDRAW, LEGO CAD software package.*  
http://www.ldraw.org/.

http://doi.acm.org/10.1145/293347.293353.

http://www.lm-software.com/mlcad/.

http://lego.bldesign.org/LSculpt/.


http://www idi.ntnu.no/grupper/ai/eval/brick/.


http://www idi.ntnu.no/grupper/ai/eval/brick/.

http://citeseer.ist.psu.edu/article/szeliski90realtime.html.


http://www.idi.ntnu.no/grupper/ai/eval/brick/.

www.prip.tuwien.ac.at/people/sab/papers/cvww02.pdf.


http://uncomp.uwe.ac.uk/automata2008/.


Appendix A

Results

A.1 3D geometric models used

Figure A.1 shows all the 3D geometric models used in the thesis. All the models were downloaded from the 3D Meshes Research Database [1], except the cube and the Stanford bunny [11] models. The cube was created using the 3D modeling application Blender [2].

A.2 Complete test results

Tables A.1 to A.20 show all the results that we found for both the beam search method and the cellular automata with cell clustering method. Since the cellular automata method has a random element in the merging phase, the average values over a number of runs are given. We therefore add the standard deviation of the cost value over all runs. Note that the standard deviation decreases as the number of time steps are increased.
Figure A.1: Images of the 3D geometric models used in this thesis.
Table A.1: Results from all beam search experiments for the cube 3D geometric model with 8 layers and an edge thickness of 2.

<table>
<thead>
<tr>
<th>Tree width</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>29644</td>
<td>68</td>
<td>0</td>
<td>6612</td>
<td>63</td>
<td>0</td>
</tr>
<tr>
<td>500</td>
<td>19826</td>
<td>44</td>
<td>1</td>
<td>6537</td>
<td>47</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>20890</td>
<td>46</td>
<td>2</td>
<td>7058</td>
<td>51</td>
<td>2</td>
</tr>
<tr>
<td>2000</td>
<td>20702</td>
<td>45</td>
<td>4</td>
<td>7202</td>
<td>45</td>
<td>4</td>
</tr>
<tr>
<td>4000</td>
<td>20158</td>
<td>43</td>
<td>8</td>
<td>6236</td>
<td>40</td>
<td>8</td>
</tr>
<tr>
<td>10000</td>
<td>16210</td>
<td>32</td>
<td>19</td>
<td>5999</td>
<td>35</td>
<td>21</td>
</tr>
<tr>
<td>20000</td>
<td>16210</td>
<td>32</td>
<td>19</td>
<td>5999</td>
<td>35</td>
<td>49</td>
</tr>
</tbody>
</table>

Table A.2: Results from all cellular automata experiments for the cube 3D geometric model with 8 layers and an edge thickness of 2.

<table>
<thead>
<tr>
<th>Time steps</th>
<th>High brick cost $C_{numbricks} = 500$</th>
<th>Low brick cost $C_{numbricks} = 200$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average Cost</td>
<td>Bricks</td>
</tr>
<tr>
<td>100</td>
<td>30052</td>
<td>60</td>
</tr>
<tr>
<td>200</td>
<td>25984</td>
<td>52</td>
</tr>
<tr>
<td>500</td>
<td>23911</td>
<td>48</td>
</tr>
<tr>
<td>1000</td>
<td>20905</td>
<td>42</td>
</tr>
<tr>
<td>1500</td>
<td>20266</td>
<td>41</td>
</tr>
<tr>
<td>2000</td>
<td>20195</td>
<td>41</td>
</tr>
<tr>
<td>5000</td>
<td>18093</td>
<td>37</td>
</tr>
<tr>
<td>10000</td>
<td>17526</td>
<td>35</td>
</tr>
</tbody>
</table>
### Table A.3: Results from all beam search experiments for chess pawn 3D geometric model with 32 layers and a minimum edge thickness of 2.

<table>
<thead>
<tr>
<th>Tree width</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>824038</td>
<td>1270</td>
<td>6</td>
<td>450805</td>
<td>1278</td>
<td>6</td>
</tr>
<tr>
<td>500</td>
<td>760809</td>
<td>1226</td>
<td>29</td>
<td>385860</td>
<td>1220</td>
<td>30</td>
</tr>
<tr>
<td>1000</td>
<td>763123</td>
<td>1209</td>
<td>64</td>
<td>384220</td>
<td>1205</td>
<td>61</td>
</tr>
<tr>
<td>2000</td>
<td>729555</td>
<td>1188</td>
<td>126</td>
<td>378015</td>
<td>1209</td>
<td>118</td>
</tr>
<tr>
<td>4000</td>
<td>732426</td>
<td>1185</td>
<td>245</td>
<td>370720</td>
<td>1191</td>
<td>238</td>
</tr>
<tr>
<td>10000</td>
<td>722045</td>
<td>1175</td>
<td>680</td>
<td>362410</td>
<td>1153</td>
<td>716</td>
</tr>
<tr>
<td>20000</td>
<td>699237</td>
<td>1138</td>
<td>1735</td>
<td>348144</td>
<td>1169</td>
<td>1770</td>
</tr>
</tbody>
</table>

### Table A.4: Results from all cellular automata experiments for chess pawn 3D geometric model with 32 layers and a minimum edge thickness of 2.

<table>
<thead>
<tr>
<th>Time steps</th>
<th>Average Cost</th>
<th>Bricks</th>
<th>Average Time (s)</th>
<th>Std Dev.</th>
<th>Average Cost</th>
<th>Bricks</th>
<th>Average Time (s)</th>
<th>Std Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>864465</td>
<td>1130</td>
<td>2</td>
<td>18911</td>
<td>522146</td>
<td>1146</td>
<td>2</td>
<td>21802</td>
</tr>
<tr>
<td>200</td>
<td>808191</td>
<td>1071</td>
<td>3</td>
<td>11126</td>
<td>498737</td>
<td>1109</td>
<td>3</td>
<td>14316</td>
</tr>
<tr>
<td>500</td>
<td>763474</td>
<td>1022</td>
<td>7</td>
<td>14877</td>
<td>453097</td>
<td>1048</td>
<td>7</td>
<td>14535</td>
</tr>
<tr>
<td>1000</td>
<td>737177</td>
<td>994</td>
<td>14</td>
<td>12395</td>
<td>437471</td>
<td>1032</td>
<td>13</td>
<td>8356</td>
</tr>
<tr>
<td>1500</td>
<td>720120</td>
<td>972</td>
<td>20</td>
<td>12966</td>
<td>427001</td>
<td>1013</td>
<td>19</td>
<td>11864</td>
</tr>
<tr>
<td>2000</td>
<td>712276</td>
<td>966</td>
<td>26</td>
<td>9191</td>
<td>419661</td>
<td>1006</td>
<td>25</td>
<td>8353</td>
</tr>
<tr>
<td>5000</td>
<td>689755</td>
<td>940</td>
<td>64</td>
<td>7843</td>
<td>407800</td>
<td>992</td>
<td>63</td>
<td>6631</td>
</tr>
<tr>
<td>10000</td>
<td>679701</td>
<td>929</td>
<td>127</td>
<td>7656</td>
<td>396421</td>
<td>981</td>
<td>125</td>
<td>6998</td>
</tr>
</tbody>
</table>
Table A.5: Results from all beam search experiments for the Stanford bunny 3D geometric model with 32 layers with a minimum edge thickness of 4.

<table>
<thead>
<tr>
<th>Tree width</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>756886</td>
<td>1475</td>
<td>7</td>
<td>327181</td>
<td>1504</td>
<td>7</td>
</tr>
<tr>
<td>500</td>
<td>742061</td>
<td>1463</td>
<td>36</td>
<td>295807</td>
<td>1448</td>
<td>36</td>
</tr>
<tr>
<td>1000</td>
<td>707922</td>
<td>1411</td>
<td>72</td>
<td>292225</td>
<td>1469</td>
<td>74</td>
</tr>
<tr>
<td>2000</td>
<td>714327</td>
<td>1424</td>
<td>110</td>
<td>287794</td>
<td>1448</td>
<td>150</td>
</tr>
<tr>
<td>4000</td>
<td>704583</td>
<td>1414</td>
<td>150</td>
<td>272417</td>
<td>1421</td>
<td>312</td>
</tr>
<tr>
<td>10000</td>
<td>714223</td>
<td>1436</td>
<td>313</td>
<td>263529</td>
<td>1375</td>
<td>889</td>
</tr>
<tr>
<td>20000</td>
<td>661819</td>
<td>1351</td>
<td>2279</td>
<td>262396</td>
<td>1394</td>
<td>2388</td>
</tr>
</tbody>
</table>

Table A.6: Results from all cellular automata experiments for the Stanford bunny 3D geometric model with 32 layers with a minimum edge thickness of 4.

<table>
<thead>
<tr>
<th>Time steps</th>
<th>High brick cost $C_{numbricks} = 500$</th>
<th>Low brick cost $C_{numbricks} = 200$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average Cost</td>
<td>Bricks</td>
</tr>
<tr>
<td>100</td>
<td>736660</td>
<td>2</td>
</tr>
<tr>
<td>200</td>
<td>695176</td>
<td>1265</td>
</tr>
<tr>
<td>500</td>
<td>649540</td>
<td>1192</td>
</tr>
<tr>
<td>1000</td>
<td>625952</td>
<td>1156</td>
</tr>
<tr>
<td>1500</td>
<td>611012</td>
<td>1137</td>
</tr>
<tr>
<td>2000</td>
<td>605186</td>
<td>1127</td>
</tr>
<tr>
<td>5000</td>
<td>586609</td>
<td>1098</td>
</tr>
<tr>
<td>10000</td>
<td>571807</td>
<td>1077</td>
</tr>
</tbody>
</table>
Table A.7: Results from all beam search experiments for the cube 3D geometric model with 32 layers and a minimum edge thickness of 4.

<table>
<thead>
<tr>
<th>Tree width</th>
<th>High brick cost: ( C_{numbricks} = 500 )</th>
<th>Low brick cost ( C_{numbricks} = 200 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Bricks</td>
</tr>
<tr>
<td>100</td>
<td>1150829</td>
<td>2998</td>
</tr>
<tr>
<td>500</td>
<td>1187368</td>
<td>3058</td>
</tr>
<tr>
<td>1000</td>
<td>1126130</td>
<td>2920</td>
</tr>
<tr>
<td>2000</td>
<td>1089674</td>
<td>2814</td>
</tr>
<tr>
<td>4000</td>
<td>1096156</td>
<td>2829</td>
</tr>
<tr>
<td>10000</td>
<td>1100148</td>
<td>2823</td>
</tr>
</tbody>
</table>

Table A.8: Results from all cellular automata experiments for the cube 3D geometric model with 32 layers and a minimum edge thickness of 4.

<table>
<thead>
<tr>
<th>Time steps</th>
<th>High brick cost ( C_{numbricks} = 500 )</th>
<th>Low brick cost ( C_{numbricks} = 200 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average Cost</td>
<td>Bricks</td>
</tr>
<tr>
<td>100</td>
<td>1262079</td>
<td>2766</td>
</tr>
<tr>
<td>200</td>
<td>1152919</td>
<td>2552</td>
</tr>
<tr>
<td>500</td>
<td>1052925</td>
<td>2350</td>
</tr>
<tr>
<td>1000</td>
<td>999374</td>
<td>2242</td>
</tr>
<tr>
<td>1500</td>
<td>984590</td>
<td>2207</td>
</tr>
<tr>
<td>2000</td>
<td>962441</td>
<td>2166</td>
</tr>
<tr>
<td>5000</td>
<td>891320</td>
<td>1988</td>
</tr>
<tr>
<td>10000</td>
<td>871269</td>
<td>1945</td>
</tr>
</tbody>
</table>

Table A.7: Results from all beam search experiments for the cube 3D geometric model with 32 layers and a minimum edge thickness of 4.

Table A.8: Results from all cellular automata experiments for the cube 3D geometric model with 32 layers and a minimum edge thickness of 4.
### High brick cost: $C_{\text{numbricks}} = 500$

<table>
<thead>
<tr>
<th>Tree width</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>940524</td>
<td>1853</td>
<td>7</td>
<td>378112</td>
<td>1842</td>
<td>7</td>
</tr>
<tr>
<td>500</td>
<td>916857</td>
<td>1829</td>
<td>36</td>
<td>363140</td>
<td>1821</td>
<td>34</td>
</tr>
<tr>
<td>1000</td>
<td>879235</td>
<td>1772</td>
<td>69</td>
<td>341062</td>
<td>1782</td>
<td>69</td>
</tr>
<tr>
<td>2000</td>
<td>870479</td>
<td>1765</td>
<td>140</td>
<td>335752</td>
<td>1769</td>
<td>144</td>
</tr>
<tr>
<td>4000</td>
<td>868678</td>
<td>1756</td>
<td>269</td>
<td>334402</td>
<td>1773</td>
<td>274</td>
</tr>
<tr>
<td>10000</td>
<td>827217</td>
<td>1694</td>
<td>800</td>
<td>317700</td>
<td>1718</td>
<td>809</td>
</tr>
<tr>
<td>20000</td>
<td>817951</td>
<td>1681</td>
<td>1839</td>
<td>310026</td>
<td>1686</td>
<td>2022</td>
</tr>
</tbody>
</table>

### Low brick cost $C_{\text{numbricks}} = 200$

<table>
<thead>
<tr>
<th>Tree width</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>940524</td>
<td>1853</td>
<td>7</td>
<td>378112</td>
<td>1842</td>
<td>7</td>
</tr>
<tr>
<td>500</td>
<td>916857</td>
<td>1829</td>
<td>36</td>
<td>363140</td>
<td>1821</td>
<td>34</td>
</tr>
<tr>
<td>1000</td>
<td>879235</td>
<td>1772</td>
<td>69</td>
<td>341062</td>
<td>1782</td>
<td>69</td>
</tr>
<tr>
<td>2000</td>
<td>870479</td>
<td>1765</td>
<td>140</td>
<td>335752</td>
<td>1769</td>
<td>144</td>
</tr>
<tr>
<td>4000</td>
<td>868678</td>
<td>1756</td>
<td>269</td>
<td>334402</td>
<td>1773</td>
<td>274</td>
</tr>
<tr>
<td>10000</td>
<td>827217</td>
<td>1694</td>
<td>800</td>
<td>317700</td>
<td>1718</td>
<td>809</td>
</tr>
<tr>
<td>20000</td>
<td>817951</td>
<td>1681</td>
<td>1839</td>
<td>310026</td>
<td>1686</td>
<td>2022</td>
</tr>
</tbody>
</table>

**Table A.9**: Results from all beam search experiments for the Max 3D geometric model with 64 layers and a minimum edge thickness of 4.

### High brick cost $C_{\text{numbricks}} = 500$

<table>
<thead>
<tr>
<th>Time steps</th>
<th>Average Cost</th>
<th>Bricks</th>
<th>Average Time (s)</th>
<th>Std Dev.</th>
<th>Average Cost</th>
<th>Bricks</th>
<th>Average Time (s)</th>
<th>Std Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>867380</td>
<td>1571</td>
<td>3</td>
<td>9893</td>
<td>39436</td>
<td>1604</td>
<td>3</td>
<td>5145</td>
</tr>
<tr>
<td>200</td>
<td>819219</td>
<td>1501</td>
<td>6</td>
<td>10432</td>
<td>364697</td>
<td>1534</td>
<td>5</td>
<td>7997</td>
</tr>
<tr>
<td>500</td>
<td>777182</td>
<td>1433</td>
<td>12</td>
<td>7429</td>
<td>340318</td>
<td>1472</td>
<td>12</td>
<td>5455</td>
</tr>
<tr>
<td>1000</td>
<td>743571</td>
<td>1379</td>
<td>23</td>
<td>8038</td>
<td>326523</td>
<td>1433</td>
<td>22</td>
<td>4968</td>
</tr>
<tr>
<td>1500</td>
<td>734425</td>
<td>1364</td>
<td>34</td>
<td>6224</td>
<td>321376</td>
<td>1424</td>
<td>33</td>
<td>5855</td>
</tr>
<tr>
<td>2000</td>
<td>725589</td>
<td>1349</td>
<td>44</td>
<td>5947</td>
<td>317257</td>
<td>1413</td>
<td>43</td>
<td>3657</td>
</tr>
<tr>
<td>5000</td>
<td>704289</td>
<td>1319</td>
<td>106</td>
<td>4727</td>
<td>303005</td>
<td>1378</td>
<td>105</td>
<td>3386</td>
</tr>
<tr>
<td>10000</td>
<td>692771</td>
<td>1298</td>
<td>211</td>
<td>4439</td>
<td>295415</td>
<td>1373</td>
<td>209</td>
<td>3617</td>
</tr>
</tbody>
</table>

**Table A.10**: Results from all cellular automata experiments for the Max 3D geometric model with 64 layers and a minimum edge thickness of 4.
Table A.11: Results from all beam search experiments for the Dzilla 3D geometric model with 64 layers and a minimum edge thickness of 4.

<table>
<thead>
<tr>
<th>Tree width</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2435744</td>
<td>4756</td>
<td>19</td>
<td>1022909</td>
<td>4856</td>
<td>22</td>
</tr>
<tr>
<td>500</td>
<td>2287963</td>
<td>4592</td>
<td>99</td>
<td>898711</td>
<td>4633</td>
<td>106</td>
</tr>
<tr>
<td>1000</td>
<td>2292700</td>
<td>4620</td>
<td>206</td>
<td>903056</td>
<td>4628</td>
<td>212</td>
</tr>
<tr>
<td>2000</td>
<td>2227825</td>
<td>4528</td>
<td>417</td>
<td>882811</td>
<td>4563</td>
<td>420</td>
</tr>
<tr>
<td>4000</td>
<td>2219955</td>
<td>4522</td>
<td>855</td>
<td>835446</td>
<td>4472</td>
<td>873</td>
</tr>
<tr>
<td>10000</td>
<td>2158557</td>
<td>4435</td>
<td>2495</td>
<td>817255</td>
<td>4441</td>
<td>2414</td>
</tr>
</tbody>
</table>

Table A.12: Results from all cellular automata experiments for the Dzilla 3D geometric model with 64 layers and a minimum edge thickness of 4.

<table>
<thead>
<tr>
<th>Time steps</th>
<th>Average Cost</th>
<th>Bricks</th>
<th>Average Time (s)</th>
<th>Std Dev.</th>
<th>Average Cost</th>
<th>Bricks</th>
<th>Average Time (s)</th>
<th>Std Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2236012</td>
<td>4036</td>
<td>5</td>
<td>16116</td>
<td>1025799</td>
<td>4101</td>
<td>5</td>
<td>9904</td>
</tr>
<tr>
<td>200</td>
<td>2124731</td>
<td>3852</td>
<td>9</td>
<td>16845</td>
<td>954085</td>
<td>3902</td>
<td>9</td>
<td>10058</td>
</tr>
<tr>
<td>500</td>
<td>1997618</td>
<td>3663</td>
<td>19</td>
<td>17080</td>
<td>894096</td>
<td>3765</td>
<td>19</td>
<td>4277</td>
</tr>
<tr>
<td>1000</td>
<td>1930189</td>
<td>3552</td>
<td>35</td>
<td>11180</td>
<td>863488</td>
<td>3657</td>
<td>35</td>
<td>7346</td>
</tr>
<tr>
<td>1500</td>
<td>1903955</td>
<td>3501</td>
<td>50</td>
<td>12502</td>
<td>846548</td>
<td>3620</td>
<td>51</td>
<td>6676</td>
</tr>
<tr>
<td>2000</td>
<td>1879168</td>
<td>3470</td>
<td>70</td>
<td>9609</td>
<td>833863</td>
<td>3592</td>
<td>67</td>
<td>6485</td>
</tr>
<tr>
<td>5000</td>
<td>1828526</td>
<td>3390</td>
<td>169</td>
<td>7944</td>
<td>798869</td>
<td>3524</td>
<td>166</td>
<td>7398</td>
</tr>
<tr>
<td>10000</td>
<td>1798646</td>
<td>3351</td>
<td>334</td>
<td>8174</td>
<td>783190</td>
<td>3489</td>
<td>328</td>
<td>2967</td>
</tr>
</tbody>
</table>
**Table A.13:** Results from all beam search experiments for the Dino 3D geometric model with 128 layers and a minimum edge thickness of 4.

<table>
<thead>
<tr>
<th>Tree width</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>8261246</td>
<td>16106</td>
<td>61</td>
<td>3494395</td>
<td>16338</td>
<td>65</td>
</tr>
<tr>
<td>500</td>
<td>7966498</td>
<td>15872</td>
<td>333</td>
<td>3207925</td>
<td>15819</td>
<td>342</td>
</tr>
<tr>
<td>1000</td>
<td>7851432</td>
<td>15763</td>
<td>646</td>
<td>3071865</td>
<td>15662</td>
<td>668</td>
</tr>
<tr>
<td>2000</td>
<td>7643511</td>
<td>15469</td>
<td>1325</td>
<td>3009737</td>
<td>15491</td>
<td>1344</td>
</tr>
<tr>
<td>4000</td>
<td>7529188</td>
<td>15328</td>
<td>2726</td>
<td>2919232</td>
<td>15301</td>
<td>2799</td>
</tr>
<tr>
<td>10000</td>
<td>7405899</td>
<td>15200</td>
<td>8279</td>
<td>2819299</td>
<td>15170</td>
<td>8273</td>
</tr>
</tbody>
</table>

**Table A.14:** Results from all cellular automata experiments for the Dino 3D geometric model with 128 layers and a minimum edge thickness of 4.

<table>
<thead>
<tr>
<th>Time steps</th>
<th>High brick cost $C_{numbricks} = 500$</th>
<th>Low brick cost $C_{numbricks} = 200$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average Cost</td>
<td>Bricks</td>
</tr>
<tr>
<td>100</td>
<td>7679398</td>
<td>13594</td>
</tr>
<tr>
<td>200</td>
<td>7312031</td>
<td>13022</td>
</tr>
<tr>
<td>500</td>
<td>6941753</td>
<td>12438</td>
</tr>
<tr>
<td>1000</td>
<td>6743764</td>
<td>12123</td>
</tr>
<tr>
<td>1500</td>
<td>6666441</td>
<td>12003</td>
</tr>
<tr>
<td>2000</td>
<td>6594524</td>
<td>11901</td>
</tr>
<tr>
<td>5000</td>
<td>6456010</td>
<td>11704</td>
</tr>
<tr>
<td>10000</td>
<td>6360444</td>
<td>11569</td>
</tr>
</tbody>
</table>
APPENDIX A. RESULTS

<table>
<thead>
<tr>
<th>Tree width</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>14818642</td>
<td>29366</td>
<td>106</td>
<td>5911255</td>
<td>29096</td>
<td>111</td>
</tr>
<tr>
<td>500</td>
<td>14107819</td>
<td>28609</td>
<td>559</td>
<td>5465520</td>
<td>28454</td>
<td>583</td>
</tr>
<tr>
<td>1000</td>
<td>13835489</td>
<td>28248</td>
<td>1080</td>
<td>5290628</td>
<td>28177</td>
<td>1190</td>
</tr>
<tr>
<td>2000</td>
<td>13573812</td>
<td>27933</td>
<td>2271</td>
<td>5146792</td>
<td>27909</td>
<td>2312</td>
</tr>
<tr>
<td>4000</td>
<td>13300676</td>
<td>27540</td>
<td>4543</td>
<td>4967198</td>
<td>27590</td>
<td>4826</td>
</tr>
</tbody>
</table>

Table A.15: Results from all beam search experiments for the Woman seated 3D geometric model with 128 layers and a minimum edge thickness of 4.

<table>
<thead>
<tr>
<th>Time steps</th>
<th>Average Cost</th>
<th>Bricks</th>
<th>Average Time (s)</th>
<th>Std Dev.</th>
<th>Average Cost</th>
<th>Bricks</th>
<th>Average Time (s)</th>
<th>Std Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>14837608</td>
<td>26068</td>
<td>19</td>
<td>50571</td>
<td>7029613</td>
<td>26280</td>
<td>19</td>
<td>35857</td>
</tr>
<tr>
<td>200</td>
<td>14188730</td>
<td>25080</td>
<td>33</td>
<td>40589</td>
<td>6665558</td>
<td>25306</td>
<td>32</td>
<td>26430</td>
</tr>
<tr>
<td>500</td>
<td>13552389</td>
<td>24082</td>
<td>72</td>
<td>36069</td>
<td>6315101</td>
<td>24380</td>
<td>68</td>
<td>19258</td>
</tr>
<tr>
<td>1000</td>
<td>13260577</td>
<td>23618</td>
<td>134</td>
<td>37228</td>
<td>6149614</td>
<td>23925</td>
<td>128</td>
<td>32604</td>
</tr>
<tr>
<td>1500</td>
<td>13098754</td>
<td>23383</td>
<td>196</td>
<td>42196</td>
<td>6069916</td>
<td>23761</td>
<td>188</td>
<td>15194</td>
</tr>
<tr>
<td>2000</td>
<td>13022284</td>
<td>23272</td>
<td>256</td>
<td>35214</td>
<td>6011396</td>
<td>23642</td>
<td>246</td>
<td>14614</td>
</tr>
</tbody>
</table>

Table A.16: Results from all cellular automata experiments for the Woman seated 3D geometric model with 128 layers and a minimum edge thickness of 4.
Table A.17: Results from all beam search experiments for the T-Rex 3D geometric model with 128 layers and a minimum edge thickness of 6.

<table>
<thead>
<tr>
<th>Tree width</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>32520530</td>
<td>71878</td>
<td>355</td>
<td>10876886</td>
<td>72026</td>
<td>387</td>
</tr>
<tr>
<td>500</td>
<td>31735408</td>
<td>71620</td>
<td>1904</td>
<td>10164021</td>
<td>71333</td>
<td>2040</td>
</tr>
<tr>
<td>1000</td>
<td>30941036</td>
<td>70376</td>
<td>3784</td>
<td>9707818</td>
<td>70162</td>
<td>3925</td>
</tr>
<tr>
<td>2000</td>
<td>30488804</td>
<td>69679</td>
<td>8366</td>
<td>9534406</td>
<td>70043</td>
<td>8199</td>
</tr>
</tbody>
</table>

Table A.18: Results from all cellular automata experiments for the T-Rex 3D geometric model with 256 layers and a minimum edge thickness of 6.

<table>
<thead>
<tr>
<th>Time steps</th>
<th>Average Cost</th>
<th>Bricks</th>
<th>Average Time (s)</th>
<th>Std Dev.</th>
<th>Average Cost</th>
<th>Bricks</th>
<th>Average Time (s)</th>
<th>Std Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>35730416</td>
<td>69364</td>
<td>50</td>
<td>134378</td>
<td>14934024</td>
<td>69949</td>
<td>50</td>
<td>75736</td>
</tr>
<tr>
<td>200</td>
<td>33833500</td>
<td>66245</td>
<td>84</td>
<td>61770</td>
<td>13942437</td>
<td>66827</td>
<td>85</td>
<td>65940</td>
</tr>
<tr>
<td>500</td>
<td>32181952</td>
<td>63424</td>
<td>181</td>
<td>84878</td>
<td>13146455</td>
<td>64311</td>
<td>186</td>
<td>45713</td>
</tr>
<tr>
<td>1000</td>
<td>31339098</td>
<td>61968</td>
<td>337</td>
<td>78264</td>
<td>12712862</td>
<td>62944</td>
<td>343</td>
<td>50922</td>
</tr>
<tr>
<td>1500</td>
<td>30937072</td>
<td>61293</td>
<td>501</td>
<td>59541</td>
<td>12518416</td>
<td>62370</td>
<td>503</td>
<td>43285</td>
</tr>
<tr>
<td>2000</td>
<td>30742324</td>
<td>60967</td>
<td>664</td>
<td>50097</td>
<td>12401480</td>
<td>62029</td>
<td>655</td>
<td>29964</td>
</tr>
</tbody>
</table>
### Table A.19

<table>
<thead>
<tr>
<th>Tree width</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
<th>Cost</th>
<th>Bricks</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>17917702</td>
<td>40027</td>
<td>226</td>
<td>5772139</td>
<td>39409</td>
<td>216</td>
</tr>
<tr>
<td>500</td>
<td>17401044</td>
<td>39306</td>
<td>1115</td>
<td>5484592</td>
<td>39269</td>
<td>1134</td>
</tr>
<tr>
<td>1000</td>
<td>17122390</td>
<td>39313</td>
<td>2401</td>
<td>5328639</td>
<td>39153</td>
<td>2292</td>
</tr>
<tr>
<td>2000</td>
<td>16690366</td>
<td>38452</td>
<td>4660</td>
<td>5020676</td>
<td>38123</td>
<td>4731</td>
</tr>
<tr>
<td>4000</td>
<td>16705394</td>
<td>38619</td>
<td>9978</td>
<td>4955946</td>
<td>38326</td>
<td>9956</td>
</tr>
</tbody>
</table>

Table A.19: Results from all beam search experiments for the Aphrodite statue 3D geometric model with 256 layers and a minimum edge thickness of 6.

### Table A.20

<table>
<thead>
<tr>
<th>Time steps</th>
<th>Average Cost</th>
<th>Bricks</th>
<th>Average Time (s)</th>
<th>Std Dev.</th>
<th>Average Cost</th>
<th>Bricks</th>
<th>Average Time (s)</th>
<th>Std Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>18516018</td>
<td>37082</td>
<td>35</td>
<td>74273</td>
<td>7396320</td>
<td>37529</td>
<td>37</td>
<td>46374</td>
</tr>
<tr>
<td>200</td>
<td>17463984</td>
<td>35271</td>
<td>58</td>
<td>63064</td>
<td>6881509</td>
<td>35770</td>
<td>65</td>
<td>34539</td>
</tr>
<tr>
<td>500</td>
<td>16496130</td>
<td>33545</td>
<td>122</td>
<td>55009</td>
<td>6414759</td>
<td>34209</td>
<td>146</td>
<td>28010</td>
</tr>
<tr>
<td>1000</td>
<td>15980918</td>
<td>32631</td>
<td>229</td>
<td>43896</td>
<td>6165839</td>
<td>33326</td>
<td>279</td>
<td>31460</td>
</tr>
<tr>
<td>1500</td>
<td>15738552</td>
<td>32174</td>
<td>335</td>
<td>39938</td>
<td>6025943</td>
<td>32907</td>
<td>413</td>
<td>21866</td>
</tr>
<tr>
<td>2000</td>
<td>15575064</td>
<td>31910</td>
<td>440</td>
<td>40264</td>
<td>5971186</td>
<td>32770</td>
<td>548</td>
<td>19649</td>
</tr>
</tbody>
</table>

Table A.20: Results from all cellular automata experiments for the Aphrodite statue 3D geometric model with 256 layers and a minimum edge thickness of 6.
Appendix B

LSculpt

The work presented in this appendix is not a direct attempt to solve the LEGO construction problem. However, it is related to building detailed LEGO sculptures using a computer application. Since it gives an interesting perspective on LEGO sculpture construction, we decided to include it in this thesis.

B.1 Voxelisation of boundary representation using orientated LEGO plates.

LEGO sculptures are generally constructed by using the standard LEGO brick set (Figure 2.3, page 7) with all the studs pointing up. Due to the shape and size of the LEGO bricks, small LEGO sculptures appear blocky and coarse. The resolution of the sculpture can be improved by using thinner LEGO brick plates, instead of the standard LEGO bricks, to increase the number of layers in the sculpture. However, this will only improve the resolution in the vertical direction as the brick plates still have the same width as the standard LEGO bricks.
Lambrecht [25] developed a method, using specialised LEGO building bricks (see Figure B.1), to build LEGO sculptures such that the LEGO plates can be orientated according to any of the three standard axes (the x, y, and z axis). The detail of the entire LEGO sculpture can be improved by orientating the small brick plates in the directions which will give the most detail. Figure B.2 shows the difference in detail between the standard studs-up method and the alternative orientated brick plates method developed by Lambrecht.

**Figure B.1:** Some of the specialised bricks used by Lambrecht to orientate LEGO brick plates in any direction.

**Figure B.2:** The sculpture on the left uses the standard studs-up method and the sculpture on the right uses Lambrecht’s method of orientating the brick plates. The level of detail found when using the alternative method of orientating the brick plates is significantly better.

Lambrecht developed a software application called LSculpt [25], which converts a triangle mesh into a LEGO model in LDraw\(^1\) format. A triangle mesh represents a 3D object, by only using triangles. The triangle mesh generally only represents the outer shell of

---

\(^1\)LDraw [22] is a computer aided design application for LEGO structures.
the object, keeping the inside hollow, to reduce the amount of information required to represent the 3D object. The application does not provide any building instructions. It produces an LDraw file consisting only of $1 \times 1$ brick plates. These brick plates are in no way connected to each other and will therefore not produce a connected sculpture. It is left to the user to select and place the specialised bricks and plates, such that the different regions are connected together. This can be a tedious task [25]. For example, it took six hours to build the bunny sculpture shown in Figure B.2, page 110.

The method developed by Lambrecht starts off by partitioning the triangle mesh into voxels. A voxel can be seen as a cube of space in a 3D grid, either empty or filled. A cube is filled if it contains or intersects any of the triangle faces. Figure B.3 gives an example of a 3D teapot which has been voxelised. Note that, due to the use of a cube, the level of detail has been reduced. The inside of the voxelised model is kept hollow to minimize the number of LEGO pieces required to build the sculpture.

![Figure B.3: A triangle mesh representing a teapot voxelised.](image)

Each resulting voxel is assigned an orientation according to the average normal vector of all the faces contained in the voxel. The average normal is used to orientate the LEGO plates that will represent the surface. Once the orientation is found, ray-casting is used to improve the detail of the model. This is done by dividing the cube used to voxelise the 3D mesh into 20 small $1 \times 1$ brick plates (see Figure B.4, page 112).

Each cube is then refined to only contain the brick plates that intersect or are inside of the triangle mesh. To help ease the building process, Lambrecht tries to optimise the directions of neighbouring bricks plates. Neighbouring brick plates with the same
Figure B.4: The cube used to voxelise the 3D triangle mesh consists of 20 small $1 \times 1$ LEGO brick plates. This is the smallest possible cube that can be constructed from $1 \times 1$ LEGO brick plates.

direction are easier to connect. Therefore, bricks which have alternate directions to its neighbours are penalised. The total sculpture penalty is minimised by reorientating a few of the worst cubes at each iteration. This is done for a maximum number of iterations or until there are no cubes that can be improved. For exact details on how the optimisation is done please see [25].

Their results show that the voxelisation process is extremely fast, converting a 3D bunny consisting of 69,451 triangles in just 500 milliseconds. The method does however struggle with some 3D models as the inside and outside of the 3D model must be defined extremely well for the ray-casting and calculation of the cube orientation to work. The method uses the normal vectors of the triangles to determine whether the voxel is inside or outside of the 3D model. If the normals are not defined correctly unexpected results such as holes or artifacts can be expected.
Appendix C

The “family” LEGO brick set

In the original LEGO construction problem the set of bricks that may be used to reconstruct the real-world object is restricted to the “family” LEGO brick set. The set of LEGO bricks included in the “family” LEGO brick set and their dimensions are given in Tables C.1 and C.2, page 114. The dimensions are measured in generic bricks, since each of these bricks can be replaced by several generic bricks stacked together.
### Table C.1: The “family” LEGO brick dimensions relative to the generic brick.

<table>
<thead>
<tr>
<th>length</th>
<th>width</th>
<th>height</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1,3,15</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1,3,6,15</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1,3,15</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1,3</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1,3,15</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1,3</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1,3</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1,3,9</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1,3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1,3,9</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1,3,9</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>1,3</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>1,3</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

### Table C.2: The LEGO DUPLO bricks included in the “family” LEGO brick set with dimensions relative to the generic brick.

<table>
<thead>
<tr>
<th>length</th>
<th>width</th>
<th>height</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>6,12</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>6,12</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>