Technical University of Crete, Greece
Department of Electronic and Computer Engineering

Error-Correcting Encoding for Self-Assembly with DNA Tiles

Vasilis Papadimitriou

Thesis Committee
Assistant Professor Michail G. Lagoudakis (ECE)
Assistant Professor Vasilis Samoladas (ECE)
Assistant Professor Aggelos Bletsas (ECE)

Chania, February 2013

Πολυτεχνειο Κρητης

Τμημα Ηλεκτρονικων Μηχανικων και Μηχανικων Υπολογιστων

Κωδικοποίηση Διόρθωσης Σφαλμάτων για Αυτο-Συναρμολόγηση με Πλακίδια DNA

Βασίλης Παπαδημητρίου

Εξεταστική Επιτροπή

Επίκουρος Καθηγητής Μιχαήλ Γ. Λαγουδάκης (ΗΜΜΥ)

Επίκουρος Καθηγητής Βασίλειος Σαμολαδάς (ΗΜΜΥ)

Επίκουρος Καθηγητής Άγγελος Μπλέτσας (ΗΜΜΥ)

Χανιά, Φεβρουάριος 2013

Abstract

As silicon-based computer technology approaches its limits, new computing paradigms, such as DNA and quantum computing, gain more and more attention from the research community in light of their potential for solving NP-hard problems by pushing the exponential time dimension into space. DNA computing investigates the possibility of encoding data and algorithmic procedures in synthetic DNA strands and exploiting the complementarity and massive parallelism properties of DNA to perform computations. Self-assembly of DNA tiles focuses on tiling computations, whereby tiles can attach to each other through carefully-designed sticky ends to create complex structures that encode desired computations. However, current DNA tiling technology is limited by the lack of reliability and robustness, meaning that during laboratory experiments several errors occur in the pairing of DNA strands/tiles, ultimately ruining the desired computation. This thesis focuses on a DNA tiling design for EXclusive-OR (XOR)-based periodic ribbon patterns, which is susceptible to errors that propagate throughout the entire assembly, and proposes two alternative encodings/designs which implement error self-correction with different trade-offs between complexity and efficiency. Both designs rely on introducing redundancy in information representation for detecting errors and additional levels of tiles for correcting errors. The proposed designs result in a drastic decrease of the probability that an error will propagate and corrupt the desired ribbon pattern. The downside of the proposed designs is the larger tile library and the increased overall size of the assembly lattice compared to the original design. Our simulated assembly results under the proposed designs verify the property of self-correction for many types of errors and indicate a clear reduction of the probability of error propagation in the assembly lattice.

Περίληψη

Καθώς η τεχνολογία υπολογιστών που βασίζεται στο πυρίτιο πλησιάζει τα όριά της, νέα πρότυπα υπολογισμού, όπως οι DNA και οι κβαντικοί υπολογιστές, αποκτούν όλο και περισσότερη προσοχή από την ερευνητική κοινότητα υπό το πρίσμα των δυνατοτήτων τους για επίλυση NP-δύσκολων προβλημάτων σπρώχνοντας την εκθετική χρονική διάσταση στο χώρο. Ο υπολογισμός με DNA διερευνά τη δυνατότητα της κωδικοποίησης δεδομένων και αλγοριθμικών διαδικασιών σε συνθετικές έλικες DNA και την αξιοποίηση των ιδιοτήτων συμπληρωματικότητας και μαζικού παραλληλισμού του DNA για την εκτέλεση υπολογισμών. Η αυτο-συναρμολόγηση πλακιδίων DNA επικεντρώνεται σε υπολογισμούς πλακόστρωσης, όπου τα πλακίδια μπορούν να προσαρτώνται το ένα στο άλλο μέσω προσεκτικά σχεδιασμένων απολήξεων για να δημιουργήσουν πολύπλοκες δομές που κωδικοποιούν επιθυμητούς υπολογισμούς. Ωστόσο, η τρέχουσα τεχνολογία αυτο-συναρμολόγησης πλακιδίων DNA περιορίζεται από την έλλειψη αξιοπιστίας και ανοχής σε σφάλματα, πράγμα που σημαίνει ότι κατά τη διάρκεια εργαστηριακών πειραμάτων εμφανίζονται πολλά σφάλματα στην αντιστοίχιση αλυσίδων και πλακιδίων DNA, καταστρέφοντας τελικά τον επιθυμητό υπολογισμό. Η παρούσα διπλωματική εργασία επικεντρώνεται σε μια σχεδίαση αυτο-συναρμολόγησης πλακιδίων DNA για περιοδικά πρότυπα/μοτίβα βασισμένα σε υπολογισμούς EXclusive-OR (XOR), η οποία είναι επιρρεπής σε σφάλματα που διαδίδονται σε όλη την συναρμολόγηση, και προτείνει δύο εναλλακτικές κωδικοποιήσεις/σχεδιάσεις που υλοποιούν αυτο-διόρθωση σφαλμάτων με διαφορετικές αναλογίες μεταξύ πολυπλοκότητας και αποτελεσματικότητας. Και οι δύο σχεδιάσεις βασίζονται στην εισαγωγή πλεονασμού στην αναπαράσταση πληροφορίας για τον εντοπισμό σφαλμάτων και πρόσθετων επιπέδων πλακιδίων για την διόρθωση σφαλμάτων. Οι προτεινόμενες σχεδιάσεις έχουν ως αποτέλεσμα τη δραστική μείωση της πιθανότητας ότι ένα σφάλμα θα διαδοθεί και θα αλλοιώσει το επιθυμητό πρότυπο/μοτίβο. Το μειονέκτημα των προτεινόμενων σχεδιάσεων είναι η μεγαλύτερη βιβλιοθήκη πλακιδίων και το αυξημένο συνολικό μέγεθος του πλέγματος συναρμολόγησης σε σχέση με τον αρχικό σχεδιασμό. Τα αποτελέσματα των προσομοιωμένων συναρμολογήσεων σύμφωνα με τις προτεινόμενες σχεδιάσεις επαληθεύουν την ιδιότητα της αυτο-διόρθωσης πολλών τύπων σφαλμάτων και δείχνουν μια σαφή μείωση της πιθανότητας διάδοσης σφαλμάτων στο πλέγμα της συναρμολόγησης.

Acknowledgements

Θα ήθελα γράψω αυτό το κομμάτι στα ελληνικά γιατί στα αγγλικά είναι πολύ πιθανό να μην το διάβαζε κανένας από αυτούς που θα αναφέρω. Αρχικά θα ήθελα να ευχαριστήσω την οικογένεια μου για την οικονομική άλλα και ψυχολογική υποστήριξη και κατανόηση που έδειξαν όλα αυτά τα χρονιά, γιατί χωρίς αυτούς ούτε σε αυτή την σχολή θα ήμουν ούτε θα είχα καταφέρει να την τελειώσω. Επίσης θέλω να ευχαριστήσω τον καθηγητή μου, για την υποστήριξη και βοήθεια του και στην επιλογή του θέματος άλλα και στην υλοποίηση του, καθώς και για τις πολλές και δύσκολες ώρες δουλειάς τις τελευταίες μέρες. Τελευταίους και καλύτερους θέλω να ευχαριστήσω αυτά τα τρελοκομεία (Μάνο, Δήμο, Στάθη, Τσιάμαλο, Μπουζού, Κον, Λελέ, Γκμοχ και τον Μεθενίτη εκεί στα ξένα) που έμπλεξα για παρέα αυτά τα χρονιά. Αφού καταφέραμε να επιβιώσουμε αρτιμελής αυτά τα έξι και χρόνια δεν φοβάμαι τίποτα. Ευχαριστώ και όσους ξέχασα να γράψω και θα μου κάνουν παράπονα!

Contents

List of Figures

Chapter 1

Introduction

Moore's Law states that silicon microprocessors double in complexity roughly every two years. One day this will no longer hold true, when miniaturization limits are reached. So, a successor to silicon is required. Fortunately, new technologies rise, such as DNA and quantum computers, to push computation boundaries further than silicon-based computers. A quantum computer [9] is a computational device that makes direct use of quantum mechanical phenomena, such as superposition and entanglement, to perform massive parallel operations on data. In this thesis, we are going to have a closer look at DNA computers. DNA computing [10] is a form of computing which exploits DNA and technologies from biochemistry and molecular biology, instead of silicon and the traditional hardware technologies. There are several properties of DNA that make it suitable for a computational element, such as its size which allows extremely dense information storage and enormous parallelism, its extraordinary energy efficiency, and primarily its complementarity. The complementarity of DNA allows to perform computations through self-assembly. Adleman's first experiment [10] used DNA strands to solve the Directed Hamiltonian Path problem, a well-known NP-complete problem. Using DNA strands and the properties of DNA self-assembly we can compose complex structures and use them as building units for computational elements. Such assemblies are known as DNA tiles [3] and, in general, offer more flexibility and robustness compared to plain DNA strands. DNA strands are still being investigated as substrates for solving computational problems, however DNA tiles seem to be more promising as candidates for replacing silicon. DNA tiles are boxed-shaped DNA assemblies with single DNA strands sticking out of its

vertices. Those single strands are called sticky ends [3]. If two tiles have complementary sticky ends, they can attach to each other. With the proper encoding of the sticky ends we can generate very complex patterns and encode computations. Although DNA computing is very promising, we are still far from replacing our silicon-based computers. This is because there are several problems to be solved, one of them being reliability and robustness, meaning that occasionally errors occur in the pairing of DNA strands/tiles.

1.1 Thesis Contribution

In this thesis, based on a DNA tiling design introduced for ribbon assemblies [8], we propose an error-correcting encoding/design for ribbon assemblies with DNA tiles. The original ribbon assembly design was selected, because it has been experimentally tested and an error probability per tile is known. We propose two different designs which implement error-correction with different trade-offs between complexity and efficiency. In both designs, in order to achieve error correction, first we ensure redundancy of information, so that we can detect errors. Then, by introducing new (correction) tiles, which are interwoven in the original design, we can correct the detected errors, if any. The proposed designs result in a drastic decrease of the probability that an error will propagate and corrupt the desired ribbon pattern. The downside of our design is the increased number of different tiles needed for the same computation as the original design and the increased overall size of the assembly lattice compared to the original design. Since experimental testing of our designs is beyond our capabilities, we implemented a simulation of the proposed self-assembly. The simulation stress test results of our designs were very promising, clearly indicating a reduction of the probability of error from 1.4% (original design) to 0.94% (first design) and 0.21% (second design).

1.2 Thesis Overview

In Chapter 2 we offer the background needed in order to better understand the rest of the thesis. We describe definitions such as DNA, DNA tiles, self-assembly, sticky ends, etc. In Chapter 3 we describe in depth the original design and then we state the problem of incorrect tile assembly we address. In Chapter 4 we describe our designs, the main ideas behind them, the new (correction) tiles, their assembly properties, and their theoretical

analysis. In Chapter 5 we describe the MATLAB code that implements the simulation. In Chapter 6 we discuss the simulation results. Finally, in Chapter 7 we discuss the results of this thesis and we suggest some possible future research enhancements and directions. Appendix A provides the full tile library for our designs.

Chapter 2

Background

2.1 DNA

DeoxyriboNucleic Acid (DNA) is a nucleic acid containing the genetic instructions used in the development and functioning of living organisms. The DNA segments carrying this genetic information are called genes. DNA is a long polymer made from repeating units called nucleotides characterized by the nucleobases (Adenine, Thymine, Guanine, Cytosine) [1].

As first discovered by James D. Watson and Francis Crick [11], the structure of DNA of all species comprises two helical strands each coiled round the same axis with a pitch of 3.4 nanometers and a radius of 1.0 nanometers (see Figure 2.1). Although each individual repeating unit is very small, DNA polymers can be very large molecules containing millions of nucleotides. For instance, the largest human chromosome, chromosome number 1, is approximately 220 million base pairs long.

In a DNA double helix, each type of nucleobase on one strand normally interacts with just one type of nucleobase on the other strand. This is called complementary base pairing. Bases form hydrogen bonds between them, with Adenine bonding only to Thymine and Cytosine bonding only to Guanine. This arrangement of two nucleotides binding together across the double helix is called a base pair. As hydrogen bonds are not covalent, they can be broken and rejoined relatively easily. The two strands of DNA in a double helix can therefore be pulled apart like a zipper, either by mechanical force or high temperature. As a result of this complementarity, all the information in the double-stranded sequence of a DNA helix is duplicated on each strand, a property which is vital in

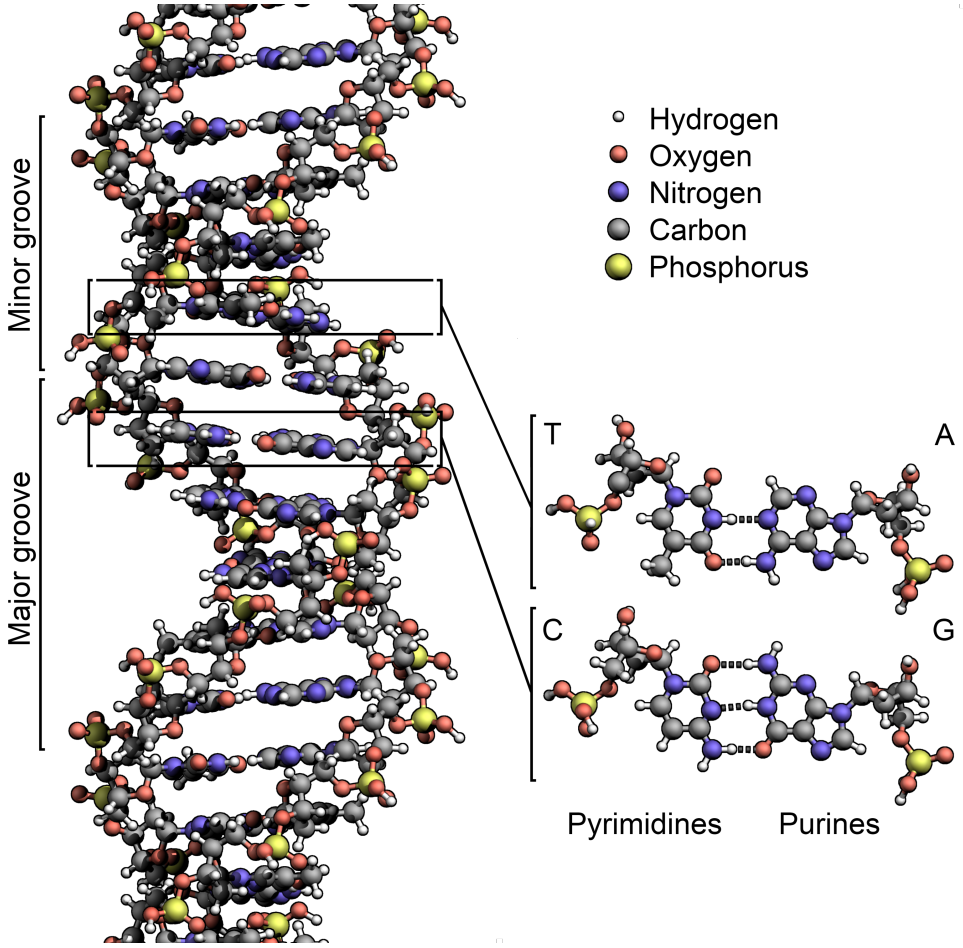


Figure 2.1: DNA double helix and base pairing (from [1])

DNA replication. Indeed, this reversible and specific interaction between complementary base pairs is critical for all the functions of DNA in living organisms [12].

2.2 DNA Computing

DNA computing [10] is a form of computing which exploits DNA and technologies from biochemistry and molecular biology, instead of silicon and the traditional hardware technologies. Adleman's first experiment [10] used DNA strands to solve the Directed Hamiltonian Path problem, a well-known NP-complete problem. DNA computing, or, more generally, biomolecular computing, is a fast developing interdisciplinary area. Research

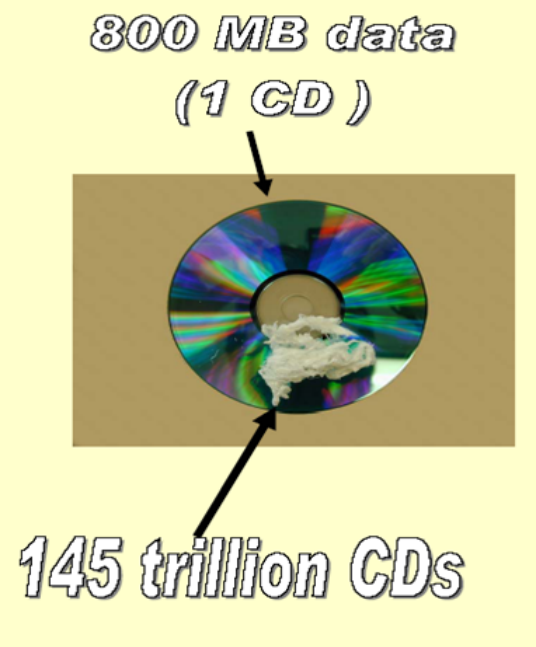


Figure 2.2: A comparison of CD and DNA densities

and development in this area concerns theory, experiments, and applications of DNA computing.

Moore's Law states that silicon microprocessors double in complexity roughly every two years. One day this will no longer hold true, when miniaturization limits are reached. Intel scientists say it will happen in about the year 2020 [13]. So, a successor to silicon is required.

DNA is a unique computational element for the following reasons. First, with DNA we can achieve extremely dense information storage. Figure 2.2 shows one gram of DNA on a compact disc (CD). The CD can hold about 800 MB of data. The one gram of DNA can hold about 1×10^{14} MB of data. The number of CDs required to hold the same amount of information, lined up edge to edge, would circle the Earth 375 times. With bases spaced at 0.35nm along DNA, data density is over a million Gbits per inch compared to 7 Gbits per inch in typical high performance hard disk drives (HDDs). One pound of DNA has the capability to store more information than all the electronic computers ever built. Second, with DNA we can achieve enormous parallelism. A test tube of DNA contains trillions of strands. Strands do not function sequentially over DNA. Each operation in a test tube of DNA is carried out on all strands in the tube in parallel!

Typically, $300,000,000,000,000$ (300 trillions) molecules are used at any single time step. Each molecule acts as a very simple CPU executing a specific computation. Third, DNA has extraordinary energy efficiency. Adleman estimated that his DNA computer was executing 2×10^{19} operations per joule [10]. Last, but not least, is the DNA's property of complementarity, which makes it suitable for computation. As mentioned before, DNA is encoded with four bases:

- A = Adenine
- T = Thymine
- G = Guanine
- C = Cytosine

These bases are like 0's and 1's used in silicon computers. DNA bases form two base-pairs: A–T and C–G. In nature, bases appear only in the form of complementary pairs. For example, the strand S=ATTACGCG is typically attached to the complementary strand $\bar{S}$=TAATGCGC.

2.3 DNA Tiles and DNA Self-assembly

A key property of DNA is the ability to assemble by itself thanks to the complementarity. *Self-assembly involves the spontaneous and autonomous organization of disorganized interacting components into an organized pattern without direct human or mechanical interference* [14]. The idea of self-assembly arose from three research fields:

1. DNA Computing [10]
2. Tiling Theory [15]
3. DNA Nanotechnology [16]

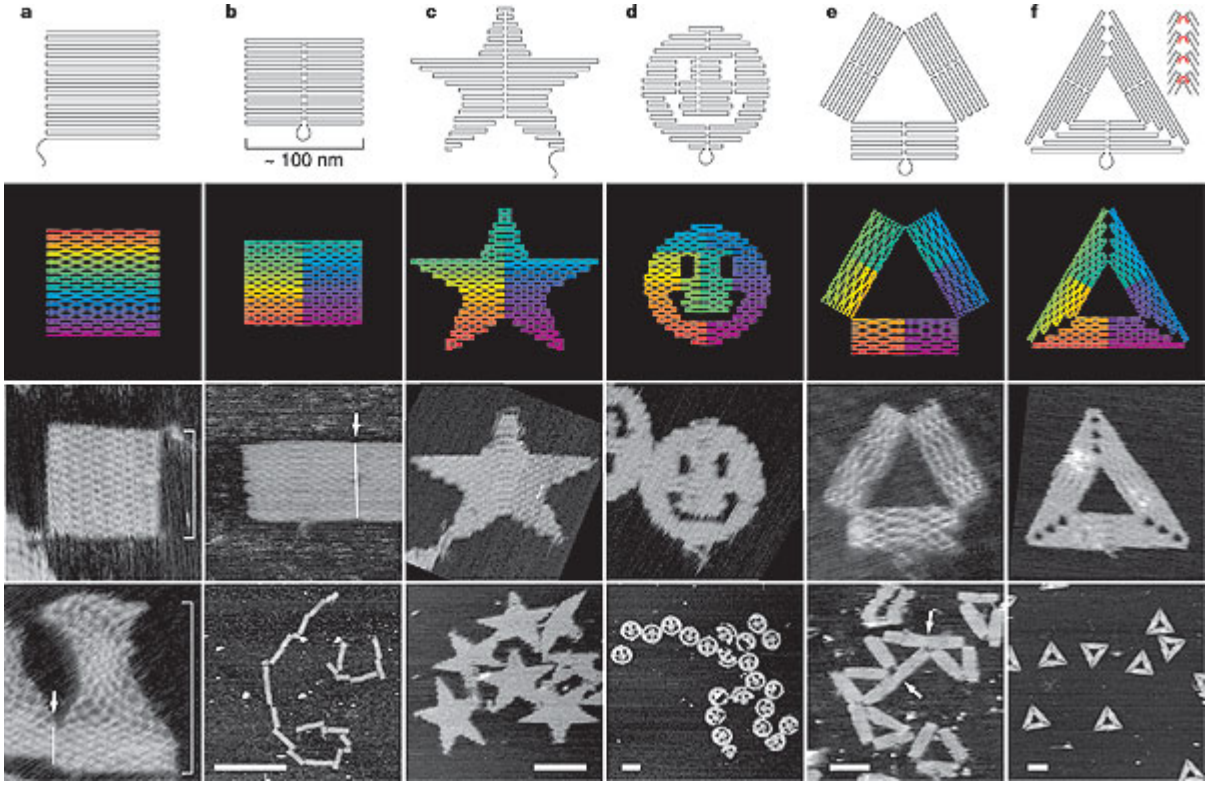


Figure 2.3: DNA origami (top: desired patterns, bottom: actual DNA patterns)(from [2])

2.3.1 DNA Tiles

As stated by Seeman [17], *the Watsons-Crick complementarity of DNA molecules allows one to design not only simple double-stranded helices but also complicated woven structures consisting of many DNA strands.* Such assemblies are known as DNA tiles [3].

To create DNA tiles we use a process called DNA folding or DNA origami. DNA origami [2] is the nanoscale folding of long DNA strands to create arbitrary two and three dimensional rigid shapes at the nanoscale (see Figure 2.3). A DNA tile or a Double Crossover DNA molecule is composed of DNA strands folded into a block-shaped molecule with strands sticking out from its vertices, called sticky ends. Figure 2.4 shows a DNA tile composed by four interwoven strands (marked in different colors). The circled regions represent the crossover between the red/green and yellow/purple strands. The yellow and green strands sticking out on both sides (four, in total) are sticky ends. If two sticky ends of different tiles are complementary they can attach to each other creating a double

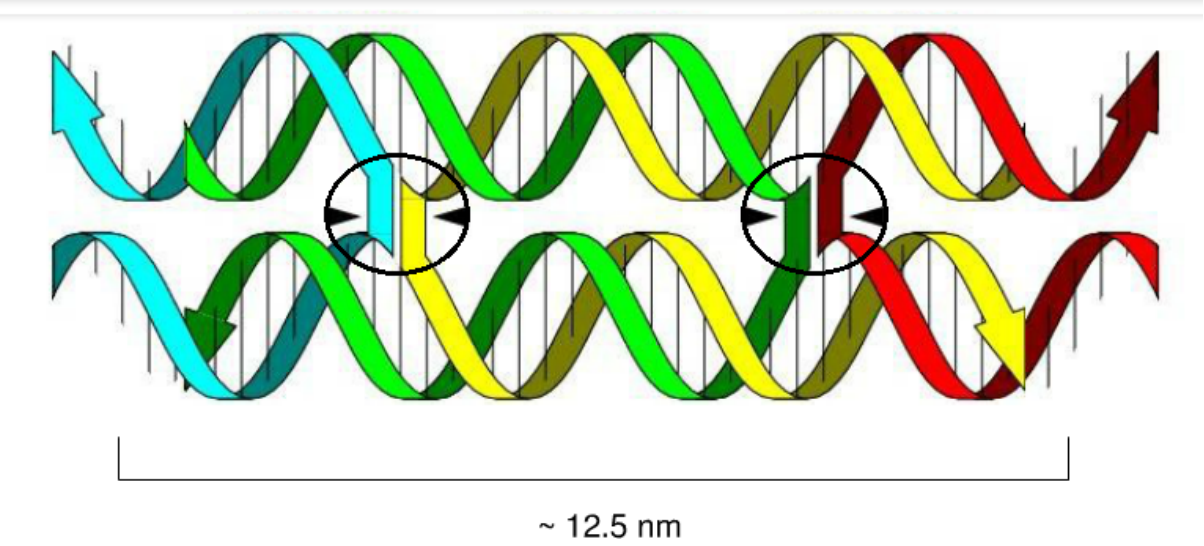


Figure 2.4: Example of a DNA tile composed of four DNA strands (from [3])

strand and resulting in an attachment of the two tiles at the corresponding vertices.

2.3.2 DNA Self-Assembly

Using DNA tiles as building blocks and the right encoding of the sticky ends we can make complex shapes and encode computations. The sticky ends must be encoded in a way that allow bonds to be created only between tiles with complementary sticky ends. Understanding of DNA self-assembly is easier with some simple examples. Figure 2.5 shows a simple assembly with two kind of DNA tiles (A and B). The upper right sticky end of A tiles is complementary to the bottom left sticky end of B tiles. Likewise, the upper left sticky end of A tiles is complementary to the bottom right sticky end of B tiles. In addition, the bottom left sticky end of A tiles is complementary to the upper right sticky end of B tiles. Finally, the bottom right sticky end of A tiles is complementary to the upper left sticky end of B tiles. Under this complementarity scheme (shown with matching colors), tiles A and B can only assemble in a two-dimensional lattice with alternating columns as shown in the figure.

Figure 2.6 shows a more complex assembly implementing a binary counter [4]. The counter has been implemented with the use of seven different tiles, three input tiles and four rule tiles. The different shapes at the edges of each tile represent different sticky

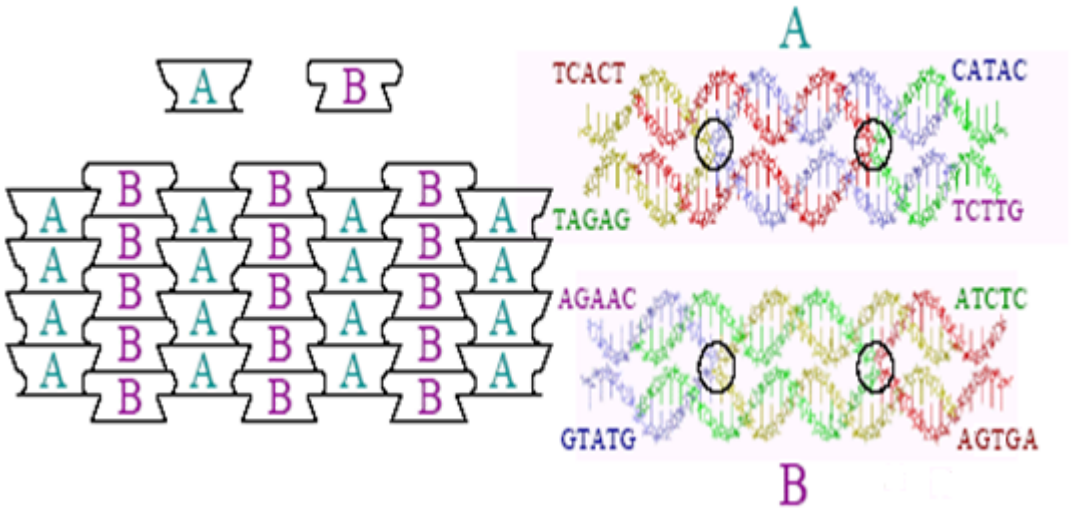


Figure 2.5: A example of a two-dimensional assembly with two DNA tiles (from [4])

ends and encode information used in the computation. Each tile stores a single bit and information about a roll-over bit, if any. Particularly, the top side of the tiles encodes the value of the bit stored in the tile and the left side carries the rollover bit value. The bottom and the right sides are used to enforce correct matching. The computation is triggered by a seed tile S and the border tiles offer the initial conditions for the computation. The width of the assembly (given by the border tiles at the bottom) determines the number of bits in the counter, while the height of the assembly (given by the border tiles on the right) determines the number up to which the counter counts. As the assembly grows, each row represents increasing numbers of the binary counter beginning with 1. The rule tiles abide to the following logic: if the rollover bit on the right side is `0`, then the current bit will simply copy the value of the bit at the bottom and will not transfer a rollover bit to its left; if the rollover bit on the right side is `1`, then the current bit will be added to the bit at the bottom, the result (modulo 2) will be stored at the current tile, and if there is any carry it will be transferred as a rollover bit to its left.

This design for binary counter has been experimentally tested [5] and yields the results shown in Figure 2.7. Although the desired pattern emerges, several errors (marked in red) occur due to incorrect attachment of rule tiles or merging of different lattices.

There are two main applications of DNA self-assembly. First, solving NP-hard problems, such as the Boolean Formula Satisfiability (SAT) problem. NP-complete problems

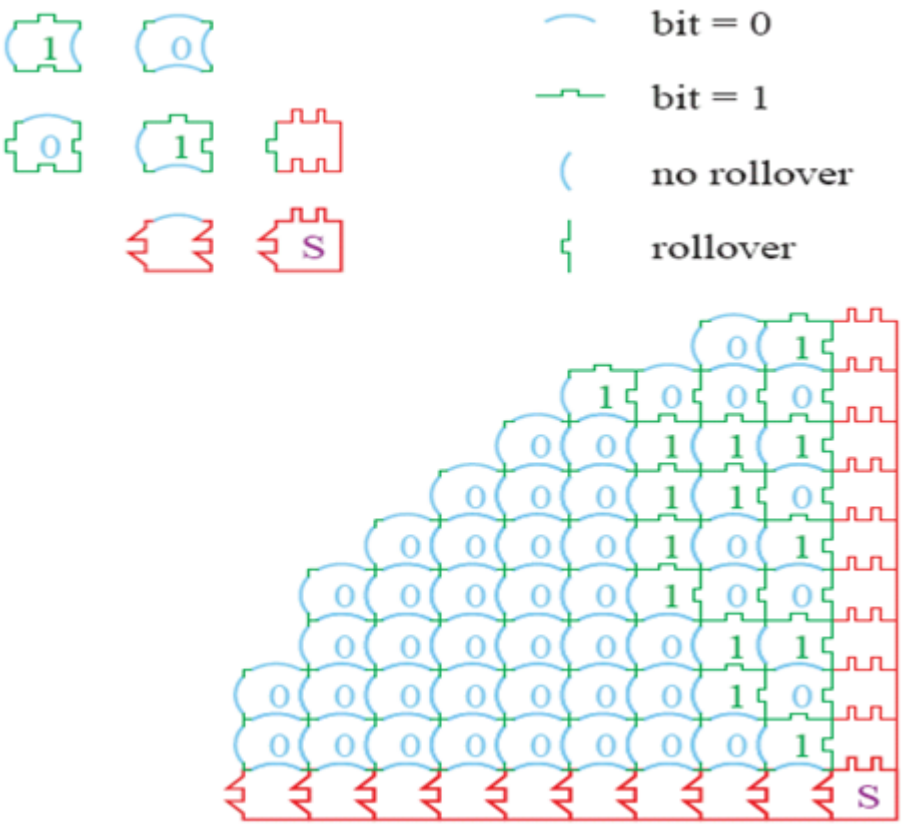


Figure 2.6: A binary counter implemented with DNA self-assembly (from [4])

have an exponential number of candidate solutions, therefore it is rather hard to find a correct solution, but it is typically easy to verify if a candidate solution is correct. With self-assembly we can generate all possible solutions and filter them out quickly using chemical methods and the parallelism of DNA. In fact, we are pushing the exponential dimension of the problem into the volume of the DNA (1 mL DNA = 2^{60} bits of information). Figure 2.8 shows an example of how a candidate solution is verified as the correct solution in an NP-hard problem. We generate input (the candidate solution) as an initial set of tiles in the first row. Based on the initial set of tiles, the assembly will end, when the YES or NO tile attaches. If the YES tile is attached, then the candidate solution is verified as a correct one, otherwise, if the NO tile is attached, the candidate solution is verified as incorrect. Alternatively, assemblies corresponding to incorrect solutions remain incomplete. Correct solutions can be read by extracting those assemblies which

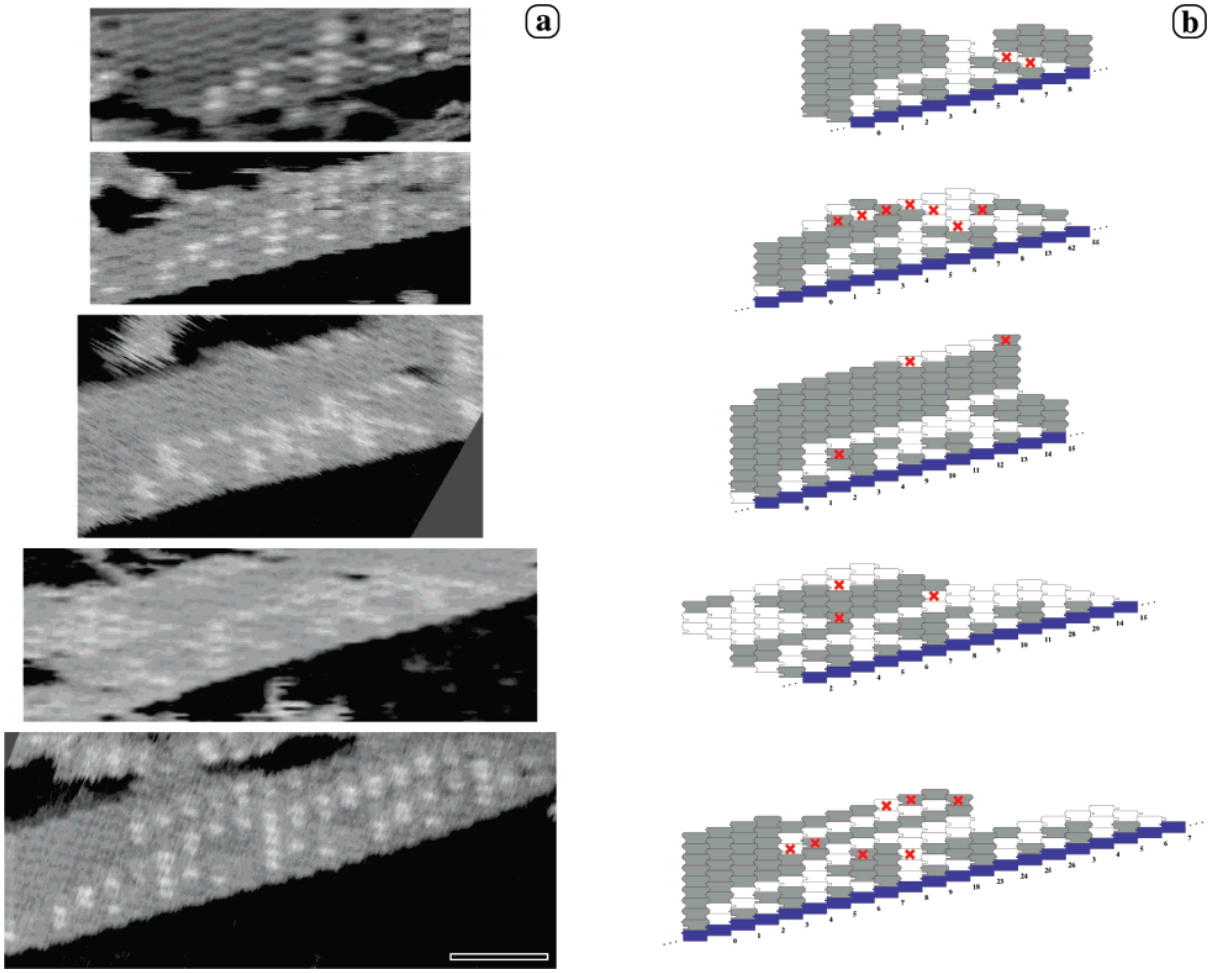


Figure 2.7: Experimental results of DNA self-assembly for a binary counter (from [5])

include a YES tile.

Figure 2.9 shows an example assembly for solution verification in the SAT problem [6]. In this design we have two initial set of tiles. The tiles in the lower left boundary encode the variables of the formula. In this example, there are only three variables: x_1, x_2, x_3. The tiles in lower right boundary encode the clauses of the formula that must be satisfied. Tile S is used to separate the clauses. Specifically, the clauses in this particular 3-SAT instance are:

$$(\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3)$$

The row of tiles next to the variables boundary (colored in cyan) encodes the candidate

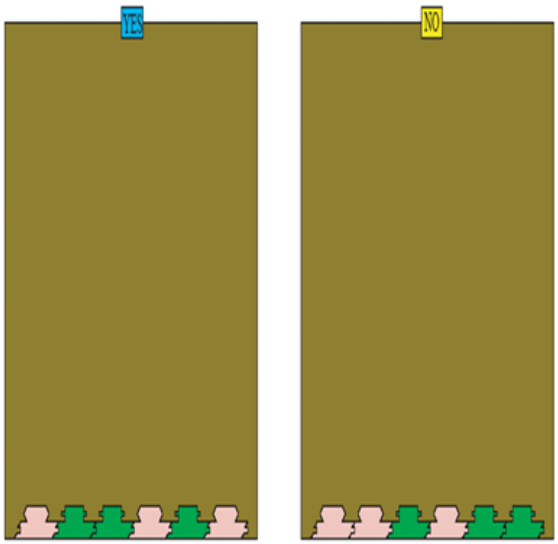


Figure 2.8: Solving an NP-hard problem using DNA self-assembly (from [4])

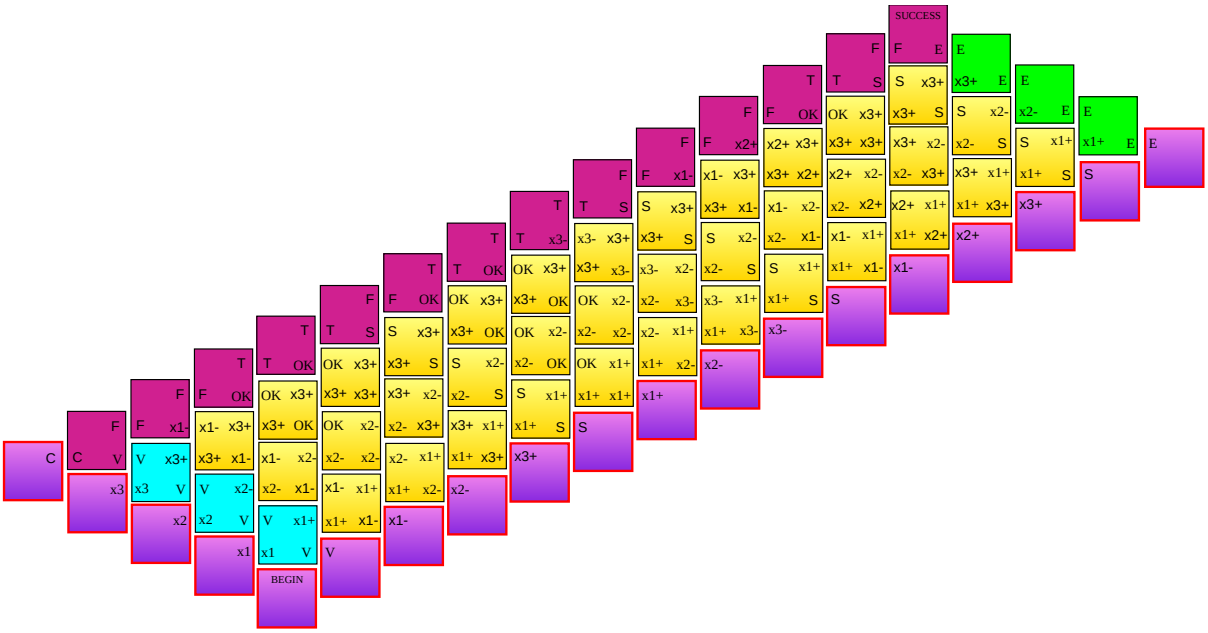


Figure 2.9: Solving the SAT problem using DNA self-assembly (from [6])

solution, a truth assignment to the variables. In this particular example, the encoded

candidate solution is the following truth assignment:

$$x_1 = True \quad x_2 = False \quad x_3 = True$$

The interior tiles in the assembly (colored in yellow) perform satisfiability checking; they check if some literal in each clause is satisfied by the chosen assignment. The findings of this check are propagated to the top left boundary, where the tiles summarize the satisfiability of each clause. If the candidate solution is a correct one (a satisfying assignment), then the assembly will complete and the tile marked with $SUCCESS$ will be attached at the very end, as shown in the figure. If the candidate solution is an incorrect one, the assembly will not be able to complete and the $SUCCESS$ tile will not be attached. Satisfying assignments can be read by extracting those assemblies which include a $SUCCESS$ tile.

The second application of DNA self-assembly is in programmable nanofabrication. Using self-assembly designs we can fabricate certain shapes, complex patterns, molecular electronic circuits, etc. An example of nanofabrication [7] is shown in Figure 2.10. Specifically, in this example we construct a RAM demultiplexer. In Figure 2.10 (a) we can see the required tile set for the implementation of the demultiplexer. The main property of these tiles is that they have simple logical gates attached on them in order to make an electronic circuit. Using the given tile set, we can create a band, shown in Figure 2.10 (b), that follows the design of the simple binary counter described above. With the help of two such bands as seed rows, the final fabrication for the RAM demultiplexer grows as a self-assembly, as shown in Figure 2.10 (c).

2.4 Properties of DNA Computers

The key advantage of DNA computers is the potential to supply massive computational power. A DNA computer can simulate parallel machines, where each processor's state is encoded by a set of DNA strands or DNA tiles. DNA computers can perform massively parallel computations by executing recombinant DNA operations that act on all the DNA molecules simultaneously. These recombinant DNA operations may be performed to execute massively parallel local memory read/write operations, logical operations, as well as basic operations on strings, such as parallel arithmetic, making them suitable for solving NP-hard problems by parallel verification of candidate solutions.

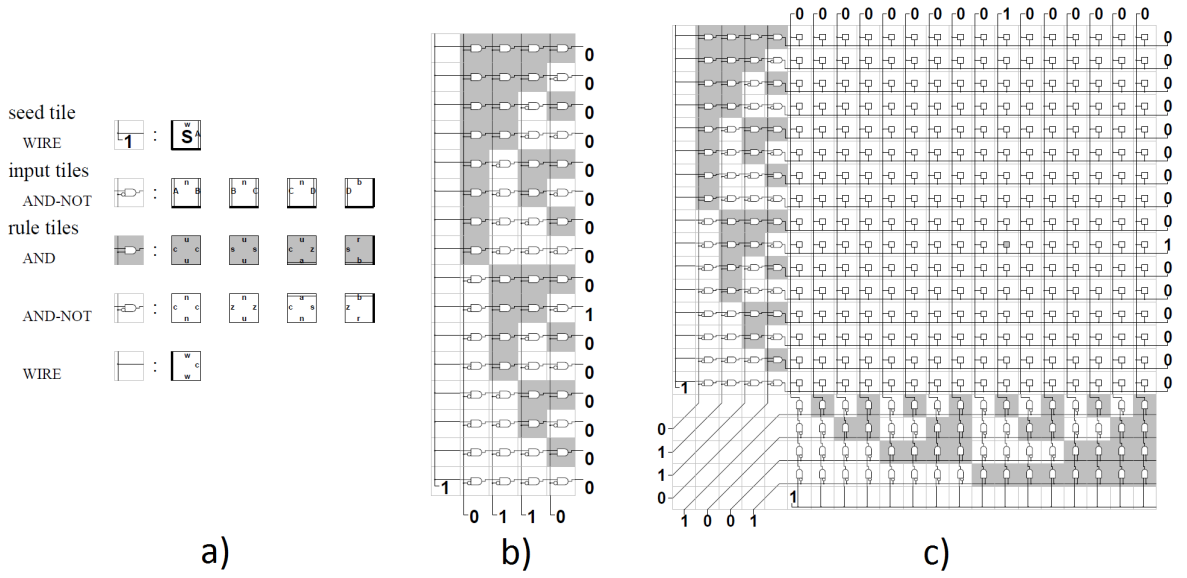


Figure 2.10: Constructing a RAM demultiplexer using DNA self-assembly (from [7])

Another advantage of DNA computers is the incredibly light weight of the actual material used. DNA in weak solution in one liter of water can encode the state of about 10^{18} processors and, since certain recombinant DNA operations can take several minutes, the overall potential for a massively parallel DNA computer is about $1,000$ tera-ops. The same solution can encode 10^7 to 10^8 tera-bytes, on which we can perform massively parallel associative searches [18].

Additionally, DNA computers consume extremely low power. The only power needed is to keep the DNA solution at the right temperature to prevent denaturation. Adleman estimated that his DNA computer was executing about 2×10^{19} operations per joule [10].

On the other hand, DNA computers suffer from high costs in time. For example, for a problem that an electronic computer needs three days to compute a solution, a DNA computer will need about 2 hours, but it may take a week of lab work for preparation. Thus, today simple problems are solved much faster on electronic computers. Also, with DNA computers, it may take longer to sort out the answer to a problem than it took to solve the problem itself. Therefore, DNA computers are generally slower than the electronic ones, mostly due to slow input/output operations.

Last, but not least, DNA computers have some reliability issues. Occasionally errors

occur in the pairing of DNA strands/tiles resulting in violations of the strict complementarity property and/or incomplete assemblies, which eventually yield wrong computations. This fact represents a major obstacle in advancing the state of the art in DNA computing and significant research efforts are dedicated in inventing chemical and/or algorithmic mechanisms that guarantee certain levels of reliability.

Chapter 3

Problem Statement

3.1 DNA Self-Assembly for Sierpinski Patterns

This thesis focuses on DNA self-assembly (tiling) for ribbon patterns. In order to fully understand how such a tiling works, we are going to describe a design for Sierpinski patterns in depth [8]. The Sierpinski triangle was chosen as a test pattern, because it requires only a small set of tiles, yet it involves all the major assembly mechanisms in which errors could occur. Each tile "computes" the eXclusive-OR (XOR) $\oplus$ function of its inputs ($0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$) in the sense that a unique tile will bind at a site corresponding to a particular input pair and will present two identical outputs representing the XOR of the inputs.

Figure 3.1 (a) shows such tile bindings, whereby the output (right side of the tiles) is the XOR operation on the inputs (left side of the tiles). A library of four abstract tiles, shown on the right side of Figure 3.1 (a), implements the correct XOR function for left-to-right growth. In addition to the rules implemented by the tiles, it is necessary to provide initial conditions for the assembly. A seed row sets the boundary conditions for growth by specifying the initial scaffolding where tiles can bind to. An initial row of `0`'s with a single `1` will produce the Sierpinski triangle pattern. Wherever both inputs match (cases pointed by black arrows), tiles may attach asynchronously to the seed row or to two adjacent tiles in the assembly. A single match (case pointed by a red arrow) is insufficient for attachment. When these assembly rules are executed without errors, these tiles grow from the seed row to produce a Sierpinski pattern, as shown in Figure 3.1 (b-top). If errors occur, such as those indicated by the two red marks in Figure 3.1 (b-bottom), the

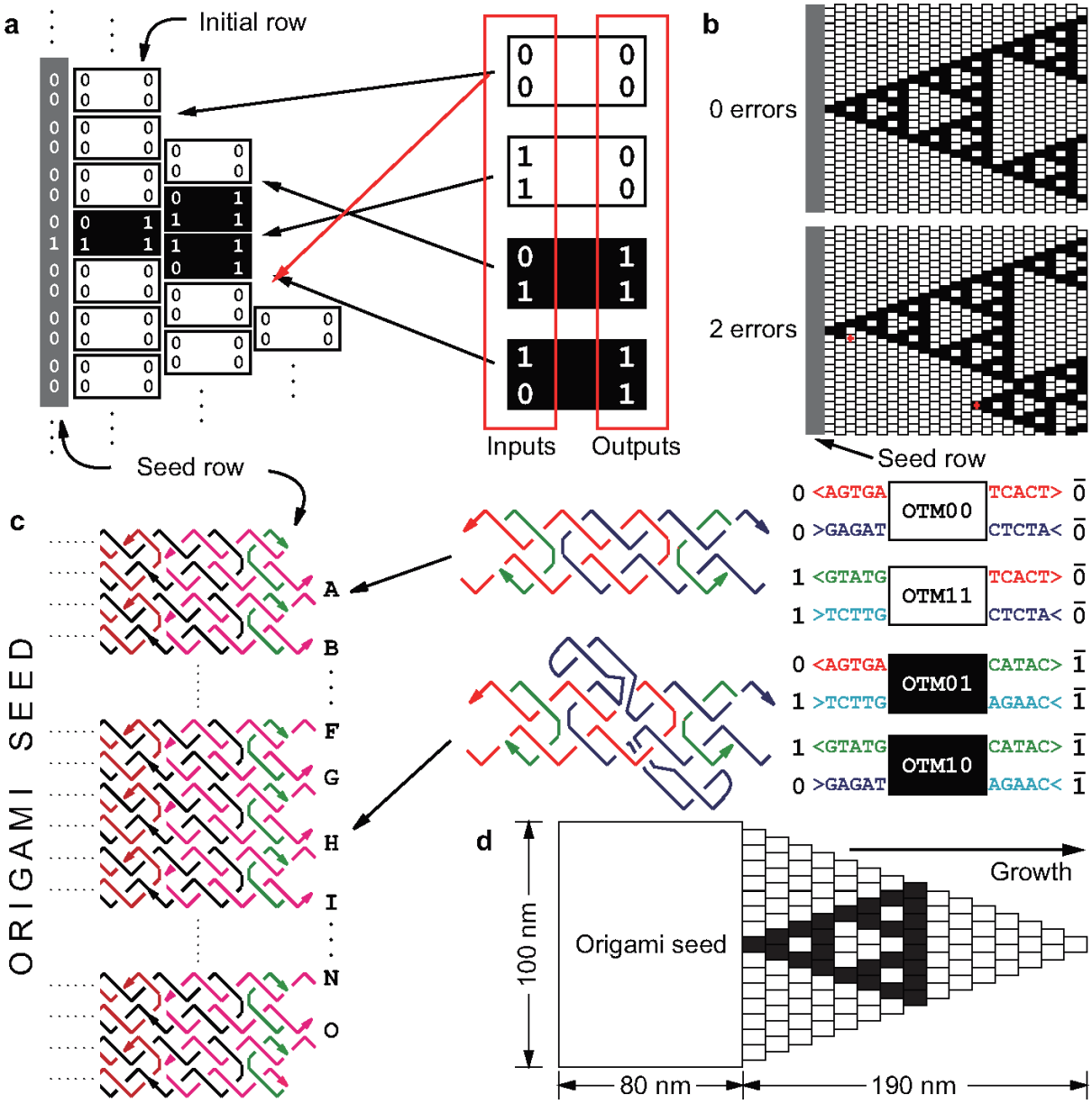


Figure 3.1: Design of DNA tiling for a Sierpinski triangle pattern (from [8])

pattern in the assembly is disrupted. A single error can result in a widespread change in the pattern, due to propagation of incorrect information.

Figure 3.1 (c) shows a complete DNA design of the XOR tiles, using a double-crossover motif [3], and part of the structure of the DNA origami seed implementing the initial conditions. The sticky ends of the four XOR tiles are 5-bases long and complementary

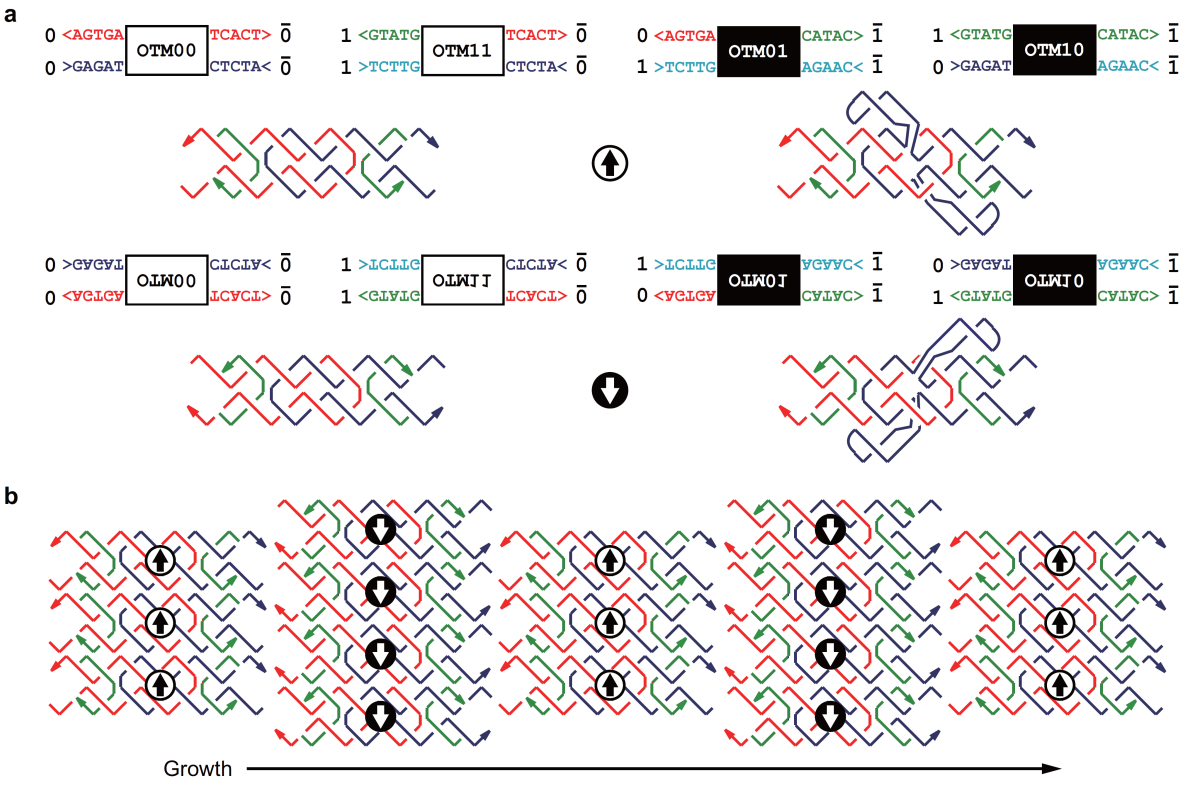


Figure 3.2: Design of DNA tiling for a Sierpinski triangle pattern (from [8])

ends are marked with the same color. To fully understand how tiles attach to each other through the sticky ends, one must consider that in alternating rows tiles are mirrored (flipped upside-down) for correct attachment, as shown in Figure 3.2, which also illustrates the structural differences between tiles with output `0` and tiles with output `1`. Finally, Figure 3.1 (d) shows the cone-shaped Sierpinski assembly produced by error-free growth from an origami seed specifying the initial row `0000000010000000` along with the actual sizes of the lattice.

In order to allow for larger assemblies, the tile set is augmented with boundary tiles, so that the assembly can grow as a fixed-width ribbon. The boundary tiles provide the necessary boundary conditions for the correct growth of the pattern, as shown in Figure 3.3 (a). This assembly begins with a seed row encoding the input `0101010101010` and results in a pattern with a 28-rows period, containing a total of 406 tiles per repeat. The boundary tiles consist of two types of single tile and one type of double tile for each of the two boundary sides (top and bottom), as shown in Figure 3.3 (b). Single

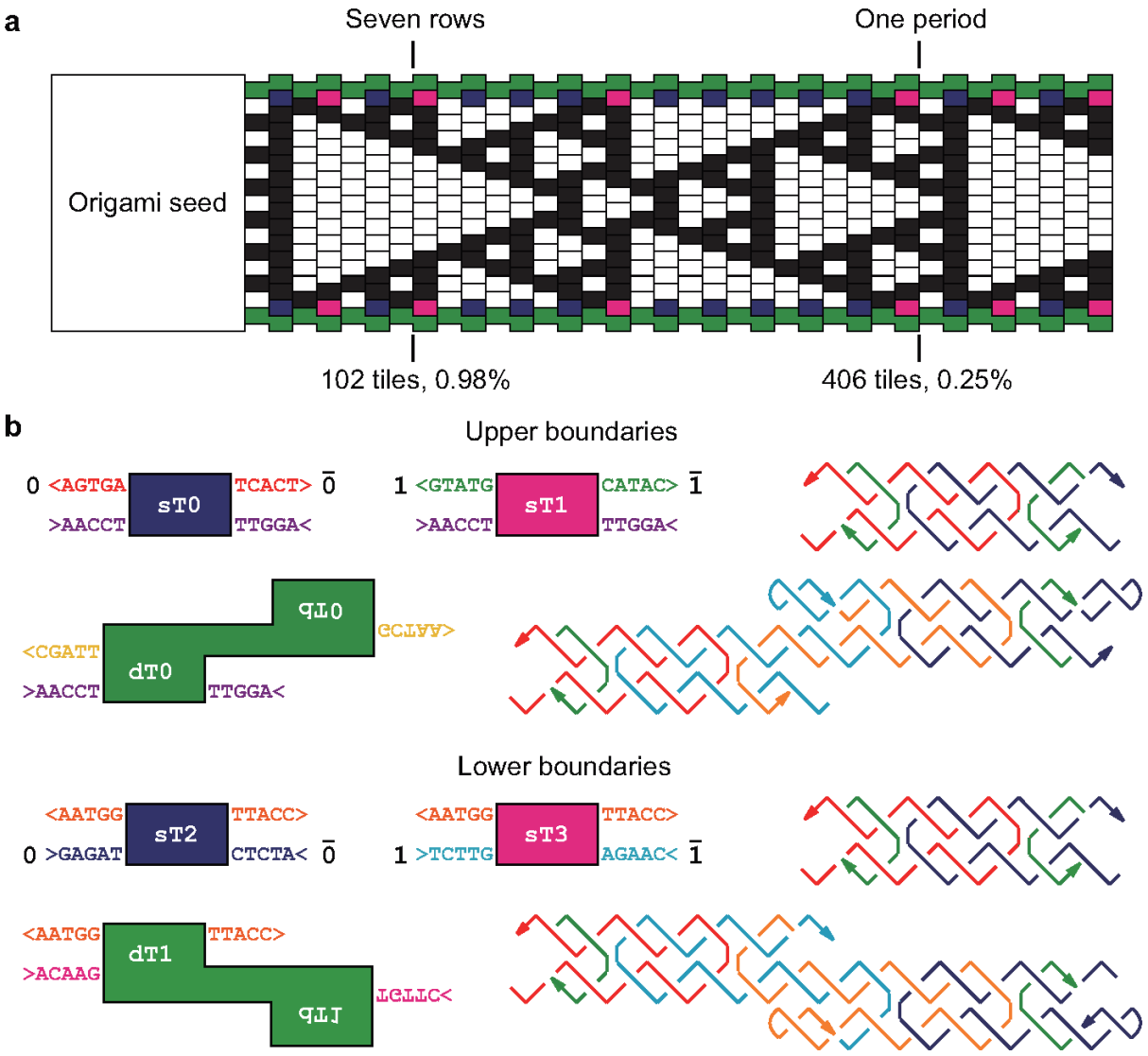


Figure 3.3: Design of DNA tiling for a Sierpinski ribbon pattern (from [8])

boundary tiles must be viewed flipped upside-down to realize how they attach to double boundary tiles. The logic is that each single boundary tile simply copies to the right the information provided by the non-boundary tile input found on its left in the assembly. Double boundary tiles on each side simply attach to each other to host the single boundary tiles and extend the assembly.

3.2 Self-Assembly Errors

The constraint required for correct growth of the ribbon assembly is that a tile may attach to a site, if and only if it matches both inputs. This is essential for growth of the correct pattern; nevertheless, errors may occur. This fact has been verified by numerous experiments in the lab with actual recombinant DNA tiles, as depicted in Figure 3.4. In fact, the longer the ribbon, the higher the probability of an error.

A first type of error shows up when a tile becomes attached by just one matching input. Such a mismatched tile will most likely only transiently attach to a growing assembly and will eventually be replaced by a correct one. However, occasionally it will become permanently embedded, especially if other tiles subsequently attach to it and grow around it. This kind of error is called a *growth error* and results in a disruption of pattern formation due to the incorporation and propagation of incorrect information.

A related second type of error results when a tile attaches by a single sticky end, despite the absence of a tile to provide the other input and then gets locked in place by subsequent lattice growth. These are called *facet nucleation errors* [19, 20]. A single facet nucleation error on a cone-shaped assembly would allow both forward and backward growth of additional layer of tiles. These new tiles are also likely to contain and propagate incorrect information, nevertheless this type of error is eliminated by the introduction of the boundary tiles and the ribbon structure of the assembly, since there are no orphan sticky ends hanging out of the assembly.

The third type of error, a *nucleation error*, occurs when several tiles come together to form a small assembly that initiates further growth in the absence of a seed DNA origami. Lacking the correct boundary conditions, such assemblies tend to be ill-formed and therefore pose no severe problems in the formation of correct lattices.

It is clear that growth errors pose the most severe threats to correct assemblies. Previous experiments on algorithmic self-assembly [21, 22] reported growth error rates between 1% and 10% per tile, although these estimates were imprecise due to highly-variable crystal growth and selective imaging. More recent statistical experimental data indicate that the probability of a single error is about 1.4% within the first 15 rows of the assembly [8]. These findings indicate that, in practice, perfect growth of Sierpinski patterns is difficult to achieve as a single error may destroy the entire pattern.

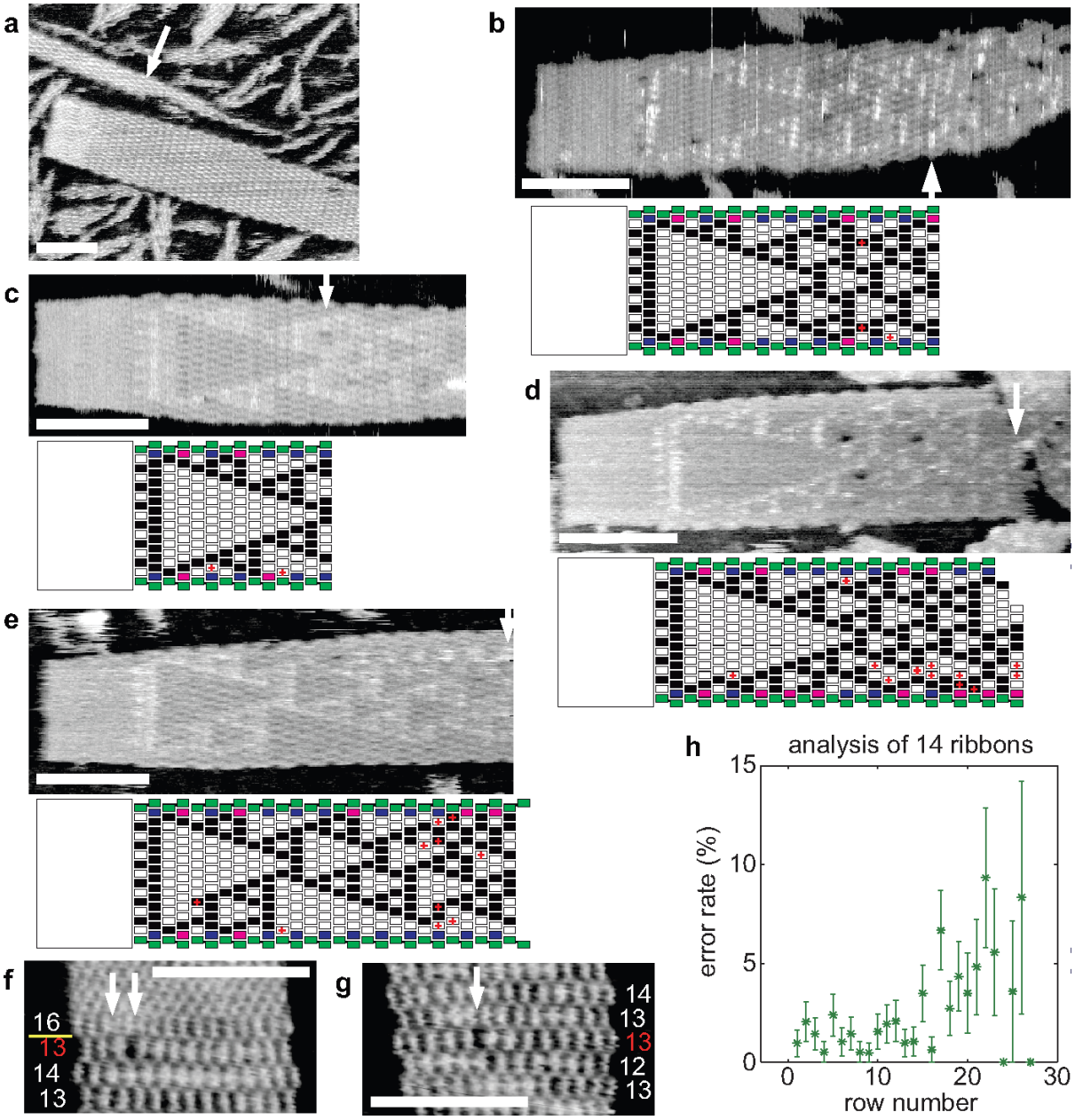


Figure 3.4: Actual DNA tilings for Sierpinski ribbon pattern and detected errors (from [8])

3.3 Seeking a Self-Correcting Design

Motivated by the severity of growth errors in DNA self-assembly for ribbon patterns, we decided to study the design of the tiles to make the assembly robust against such errors.

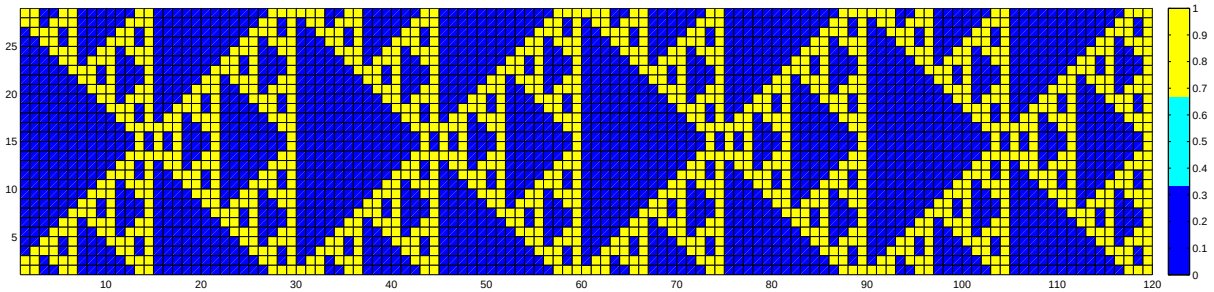


Figure 3.5: Assembly with the original design without errors (four periods)

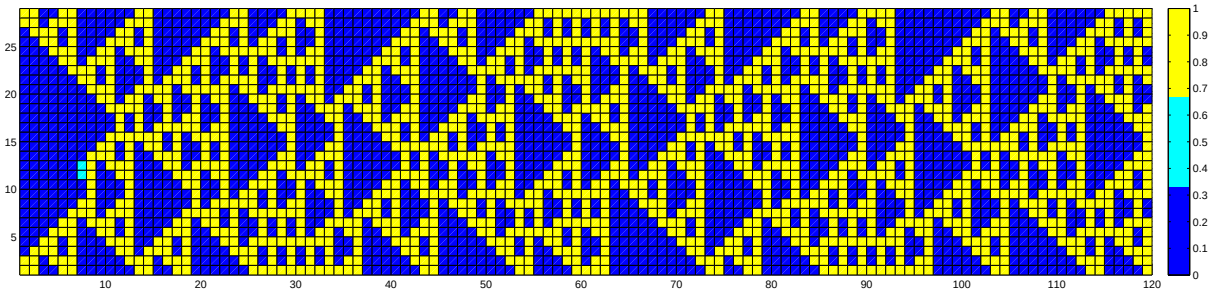


Figure 3.6: Assembly with the original design with a single error (marked in cyan)

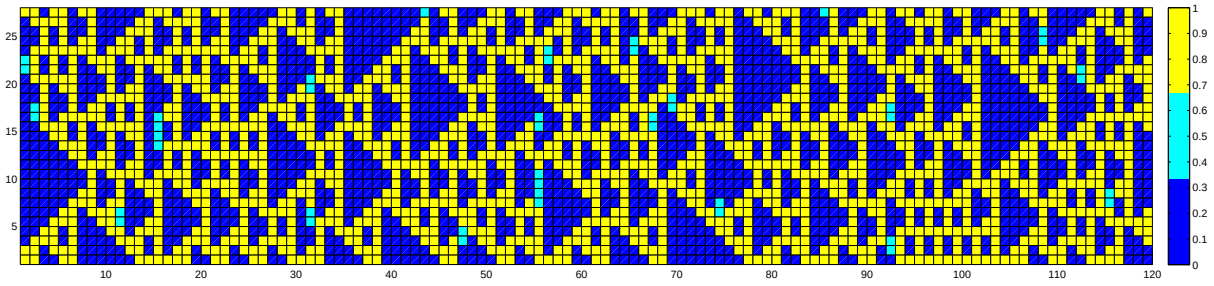


Figure 3.7: Assembly with the original design with random errors (probability=1.4%)

To illustrate the severity of the problem we conducted a number of simulations of the assembly with and without errors. Figure 3.5 shows a correct assembly of a Sierpinski ribbon pattern without errors up to four periods. Figure 3.6 shows the same assembly with just one error introduced near the beginning; notice how both the structure and the periodicity of the pattern are corrupted. A new pattern now emerges with a larger period and there is no hope of recovery. Finally, Figure 3.7 shows the same assembly with multiple errors, whereby the probability of an error occurring at each tile was taken equal to the experimentally estimated value 1.4%. In this case, no clear pattern emerges

and the resulting ribbon nowhere resembles the desired one.

The key question we try to address in this thesis is the following: Is there a different tile design which allows for recovery from errors? In other words, is it possible to restrict the influence of a single error locally, leaving only some local scars in the assembly? It is clear that such a self-correcting property cannot come for free. Some degree of redundancy will be required and some growth in the size of the assembly is unavoidable. Therefore, a few related questions are posed: What kind of redundancy is needed? How complex are the required tiles? How bigger is the required tile library? What is the influence of the required extra tiles on the size of the assembly? Is there a gain in reliability/robustness of the assembly in exchange of the required extra space? Starting from the original design and the experimentally verified error rate, we seek a new design with error correction properties in order to significantly reduce the overall error in the assembly.

Chapter 4

Our Approach

4.1 Redundancy

In our quest towards an error-correcting design we realized that we need to employ redundancy, that is to store some extra information in our tiles, so that we can detect if any kind of error occurred. We explored several options in order to achieve this kind of redundancy. For example, we considered using parity bits inside each tile, however the occurrence of an error would also lead to parity bits matching the wrong data, because of the design of the tiles. Thus, no error could be detected with just parity information.

We concluded that only some scheme that duplicates correct information would fit our purpose. We started duplicating the information contained in the original tiles by making the new tiles larger with more sticky ends (Figure 4.1). This design reduces the error rate, because for an error to occur two (not just one) sticky ends must attach incorrectly. Even so, however, the introduction of a single wrong tile will propagate incorrect information and eventually will alter the entire lattice. The next step was to duplicate the tiles themselves. So, instead of a single XOR tile for each operation, we employed two identical tiles (Figure 4.2).

This way if an error would occur at a single tile, the redundant information would remain intact. But the problem isn't solved yet, because such a scheme would simply signal the presence of an error in the best case, but we would not be able to tell how the correct the error. Fixing the error would boil down to a coin-flip situation to decide which one of the two tiles is the correct one and which is the incorrect.

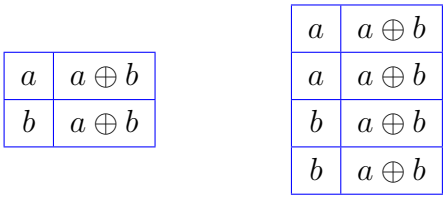


Figure 4.1: Original XOR tiles (left) and new XOR tiles with redundancy (right)

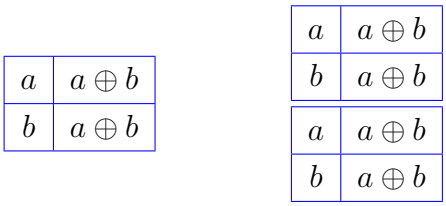


Figure 4.2: Original design (left) and new design with duplicated tiles (right)

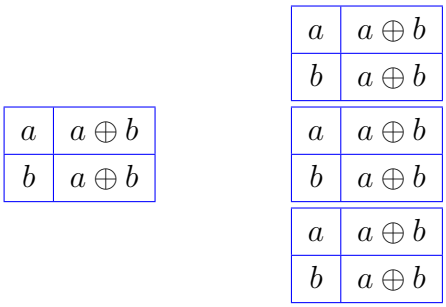


Figure 4.3: Original design (left) and new design with tripled tiles (right)

In order to avoid the coin-flip situation we tripled the tiles. This way, if an error occurs, we will have two correct and one incorrect tile. Thus, we can potentially fix the error. The only way incorrect information can propagate under such a scheme is the occurrence of two errors simultaneously at two of the three tiles in a group (Figure 4.3).

4.2 First Design

Our first design for a self-correcting assembly succeeds in correcting errors in regular XOR tiles, but suffers from certain errors occurring in the additional tiles it introduces.

4.2.1 Correction Tiles

Even now that we have achieved redundancy we still have two problems. Firstly, we need a way to detect and correct the error. The hardest part in DNA self-assembly is the correction, because, even if the incorrect tile is precisely detected, it cannot be just removed. This is true because tiles bind and unbind under precise thermodynamic control, so the change of environmental conditions in a specific parts of the assembly where an error occurred is practically impossible. Secondly, we need to make sure that the information contained in some row of tiles would be correctly propagated to the next row of tiles including the redundancy. An example will help us understand this kind of problem. Figure 4.4 (left) shows a simple example of information propagation between rows in the original design. In the first row, there are three XOR operations ($x = a \oplus b, y = c \oplus d, z = f \oplus g$), leading to two XOR operations in the second row ($e = x \oplus y, w = y \oplus z$). By design, the right sticky ends of the tiles in the first row match the left sticky ends of the tiles in the second row. This way information propagates correctly from one row to the next. Figure 4.4 (right) shows the same example along with the desired information propagation between rows under the new design with tripled tiles. Notice the requirement that the pattern of tripled tiles remains consistent between rows. The colored lines indicate which bits of information from the first row need to be transferred to the second row, as well as their exact destination. It is clear that the right sticky ends of the tiles in the first row cannot match the left sticky ends of the tiles in the second row. Therefore, in order to achieve this propagation scheme (and possibly correct errors, if any), we need to use additional intermediate levels of tiles.

In order to address these problems we are going to augment our tile library with a new set of tiles, called *correction tiles*, which are going to be inserted between the regular rows of XOR tiles. These new error-correcting tiles abide to the following template:

x	c
y	c
y	w
z	c
z	w
w	c

This template is a 6×2 tile with six sticky ends on each side and yields 16 distinct correction tiles. The left sticky ends are used to detect errors in the last row of XOR

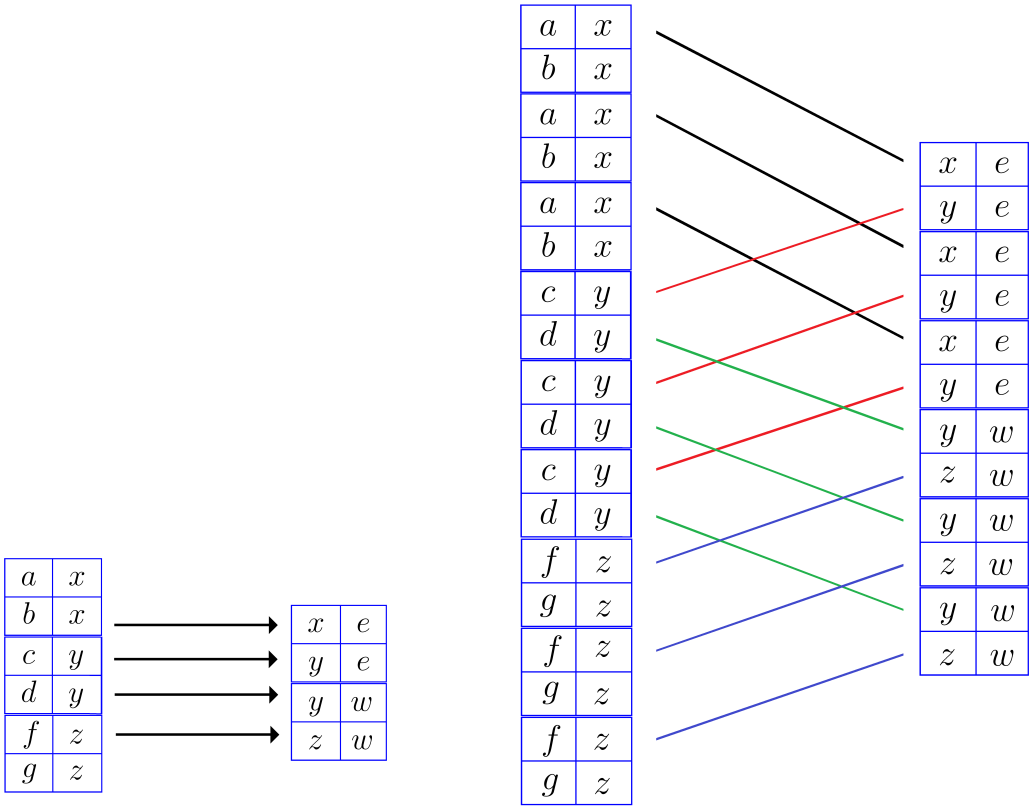


Figure 4.4: Information propagation in the original (left) and the new (right) design

tiles, while the right sticky ends are used to correct detected errors, if any, and arrange the outputs so that they match the desired arrangement in the next row of XOR tiles. The top five inputs come from the same triplet of redundant XOR tiles, therefore they should be identical, if there is not error. If not, the correct value is restored from the majority of them. More precisely, correction is implemented by the following rules:

if $(x = y = z)$ then there is no error and $c \leftarrow x$
else if $(x = y \neq z)$ then there is error in z and $c \leftarrow x$
else if $(x = z \neq y)$ then there is error in y and $c \leftarrow x$
else if $(y = z \neq x)$ then there is error in x and $c \leftarrow y$

Figure 4.5 illustrates information propagation under the new design. The assembly on the left with the original design consists of a DNA origami seed encoding the input `0101010101` and two rows of XOR tiles. The same assembly is shown in the middle with

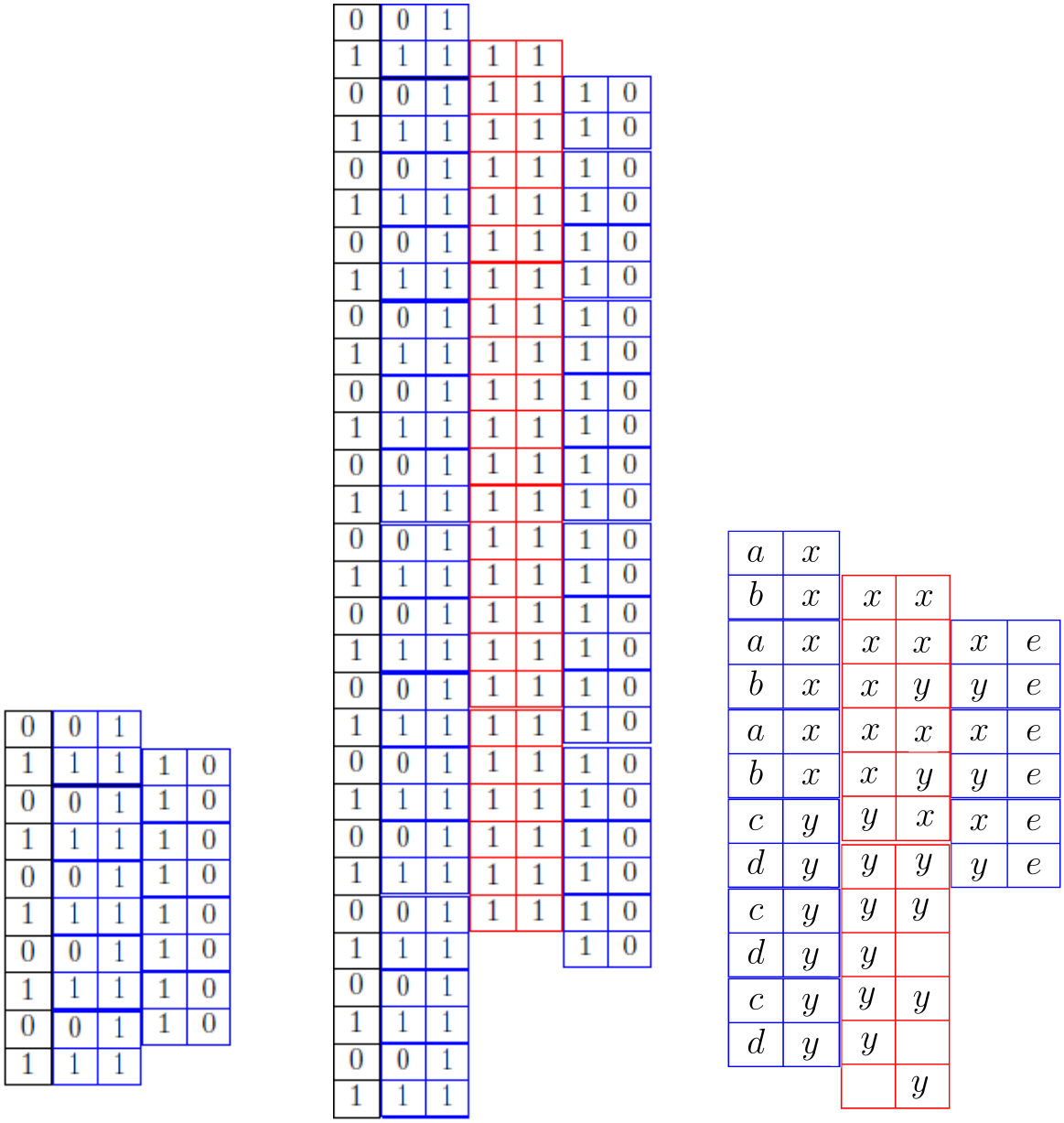


Figure 4.5: Error-free assembly: original design (left) and new design (middle and right)

the new design. The intermediate level of correction tiles is marked in red. Information propagates correctly from the first row of XOR tiles to the second one following the generic scheme shown on the right.

Figure 4.6 illustrates error correction under the new design. The three correspond to errors (marked in yellow) occurring in each of the three XOR tiles in a triplet. In the left

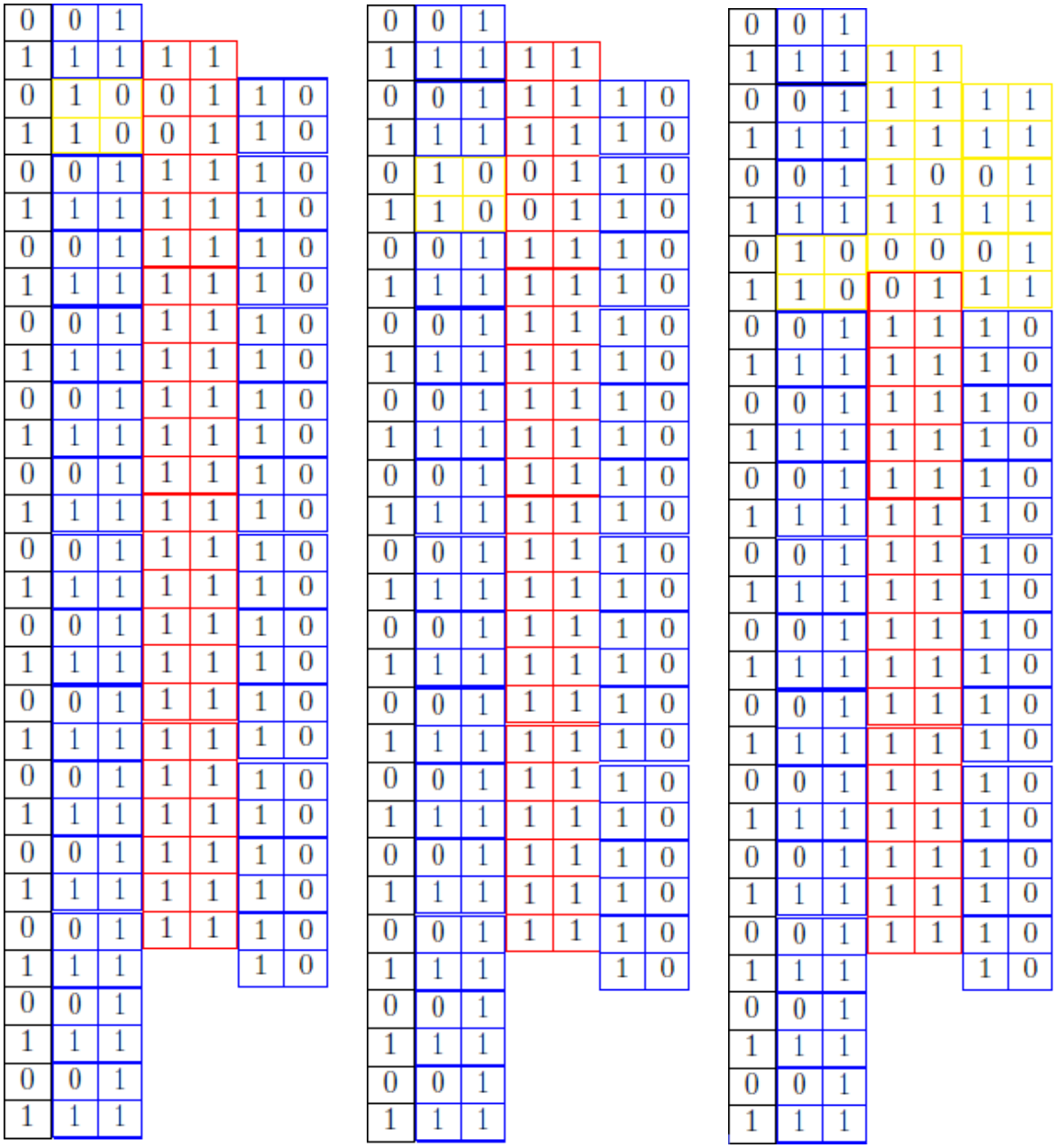


Figure 4.6: New design: error correction in each of the three XOR tiles of a triplet

and middle examples, the single error is corrected, whereas in the right example, where the error occurs in top tile of the triplet, the error propagates to the next row. Therefore, the new design can only partially correct errors in XOR tiles.

This new design so far fully succeeds in information propagation, but not in error

correction. To address both problems successfully we introduce an two levels of correction tiles between consecutive rows of XOR tiles. The first level of correction tiles is similar to the one described above and abide to the following template:

x	c
y	c
y	c
z	w
z	c
w	w

This template is a 6×2 tile with six sticky ends on each side and yields 16 distinct correction tiles (see Section A.1.1.2). Correction is implemented by the same rules as above:

if $(x = y = z)$ then there is no error and $c \leftarrow x$
else if $(x = y \neq z)$ then there is error in z and $c \leftarrow x$
else if $(x = z \neq y)$ then there is error in y and $c \leftarrow x$
else if $(y = z \neq x)$ then there is error in x and $c \leftarrow y$

The second level of correction tiles abide to the following template:

x	x
x	x
y	c
x	x
y	c
z	x

This template is a 6×2 tile with six sticky ends on each side and yields eight distinct tiles (see Section A.1.1.3). The second level aims at correcting left-over, propagated errors from the first level. Correction at this level is implemented by the following rules:

if $(y = z)$ then there is no error and $c \leftarrow y$
else if $(y \neq z)$ then there is error in y and $c \leftarrow z$

As shown in Figure 4.7, if an error occurs at the top XOR tile a triplet (a), the error will be partially corrected by the first level of correction tiles. Specifically, the lower correction tile in the first level (c) will correct the error and yield the correct information

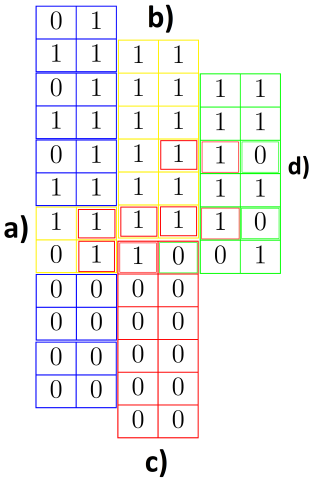


Figure 4.7: First design: example of how the second level of correction works

to the next row, whereas the upper correction tile in the first level (b) is not able to correct the error, because it has limited access to the lower triplet of XOR tiles, so it will propagate incorrect information to the next level. When such errors occur, at this specific point in the first level we always have a tile that has the correct information and one that does not, so by introducing an additional level with a correction tile that attaches to both those tiles (d) we can correct the error.

Figure 4.8 illustrates error correction under the complete first design with two levels of correction tiles. The first example is an error-free assembly. The remaining three examples correspond to errors (marked in yellow) occurring in each of the three XOR tiles in a triplet. In all cases, the single error is corrected and information propagates correctly to the next row of XOR tiles. The only corruption left in the pattern is the erroneous tile itself.

4.2.2 Boundary Tiles

As described in Chapter 3, boundary tiles are necessary for two reasons. Firstly, to maintain a fixed width for the lattice shaping it into a ribbon. Secondly, to prevent merging between different lattices.

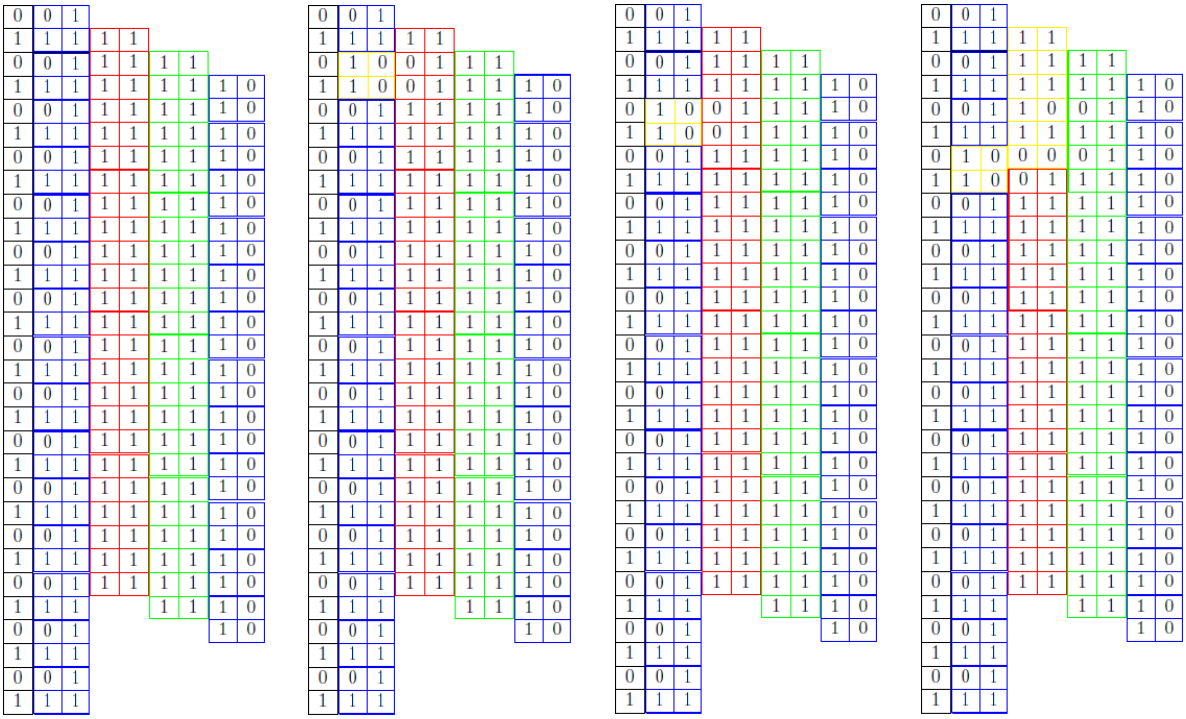


Figure 4.8: First design: error correction in each of the three XOR tiles of a triplet

XOR Boundary Tiles In the first design XOR boundary tiles follow the same principles as the boundary tiles in the original design, meaning that their purpose is to propagate information without changing it. They only difference in our first design is that the tiles have four sticky ends on each side contrary to the two sticky ends on each side of the original XOR boundary tiles. Our XOR boundary tiles abide to the following templates (left: upper boundary, right: lower boundary):

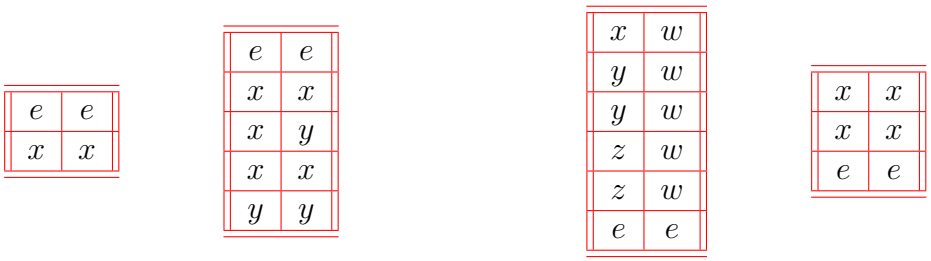
e	e
x	x
x	x
x	x

x	x
x	x
x	x
e	e

These templates are 4×2 tiles with four sticky ends on each side and yield a total of four distinct tiles (see Section A.1.2.1).

First-Level Correction Boundary Tiles These boundary tiles just need to propagate the information without altering it, maintaining at the same time the error detection

and correction properties of the first level correction tiles. Practically, in order to create the general form of these boundary tiles, we just need to "cut" the regular first level tiles to make them smaller. In particular, for the upper boundary tiles we keep the lower part of the regular tiles, whereas for the lower boundary tiles we keep the upper part of the regular tiles. Additionally, we need two types of lower and upper tiles, because we have two different sizes of boundary tiles in each first level of correction tiles. Our first-level correction boundary tiles abide to the following templates (left two: upper boundary, right two: lower boundary):

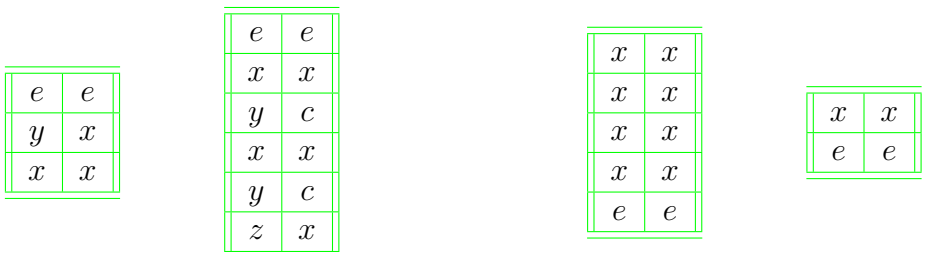


Error correction in the largest of these templates is implemented by the following rules:

if $(x = y = z)$ then no error and $w = x$
else if $(x = y \neq z)$ then there is error in z and $w \leftarrow x$
else if $(x = z \neq y)$ then there is error in y and $w \leftarrow x$
else if $(y = z \neq x)$ then there is error in x and $w \leftarrow y$

These templates yield a total of 16 distinct tiles (see Section A.1.2.2).

Second-Level Correction Boundary Tiles The boundary tiles for the second level of correction tiles are created similarly to the first level. Again, we need two types of lower and upper tiles, because we have two different sizes of boundary tiles in each second level of correction tiles. Our second-level correction boundary tiles abide to the following templates (left two: upper boundary, right two: lower boundary):



Error correction in the largest of these templates is implemented by the following rules:

if $(y = z)$ then there is no error and $c \leftarrow y$
else if $(y \neq z)$ then there is error in y and $c \leftarrow y$

These templates yield a total of 14 distinct tiles (see Section A.1.2.3).

4.2.3 Assembly

Figure 4.9 illustrates the use of boundary tiles through a complete ribbon assembly with our first design using the templates described above. Notice that the DNA origami seed must also be given in triplets of two symbols, following our redundancy scheme, and be bounded withing the boundary symbols e. At positions 1 and 4, one can see the two different types of upper boundary tiles for the first level of correction tiles. At positions 6 and 9, the two different types of lower boundary tiles for first level of correction tiles can be seen. At positions 2 and 5, the two different types of upper boundary tiles for the second level of correction tiles can be seen. Finally, at positions 7 and 10, the two different types of lower boundary tiles for the second level of correction tiles can be seen. Position 3 contains an upper boundary tile for a XOR row, whereas position 8 contains a lower boundary tile for a XOR row. Figure 4.10 shows a specific instance of this (error-free) assembly originating out of a DNA origami seed of the form `e000000101010000000e`.

4.2.4 Positioning

For correct lattice growth under our first design we must ensure that tiles attach to specific valid positions. Figure 4.11 shows examples of correct and incorrect positioning of a first-level correction tile. Correct positioning can be achieved through the use of appropriate encoding of the various sticky ends. As mentioned before sticky ends are single DNA strands usually four- to eight-bases long. When describing a tile in the abstract form $\begin{array}{|c|c|}\hline x & y \\ \hline z & w \\ \hline \end{array}$ each one of the x, y, z, w labels represents a sticky end. Two tiles can attach to each other through complementary sticky ends. For example, if sticky end y is AAAGGG, another tile can attach to y through a sticky end complementary to y, that is $\bar{y}$=TTTCCC, as shown in Figure 4.12. Both y and w in the figure may represent the same symbol, e.g. bit value `1`, but in order to enforce correct attachment each one of

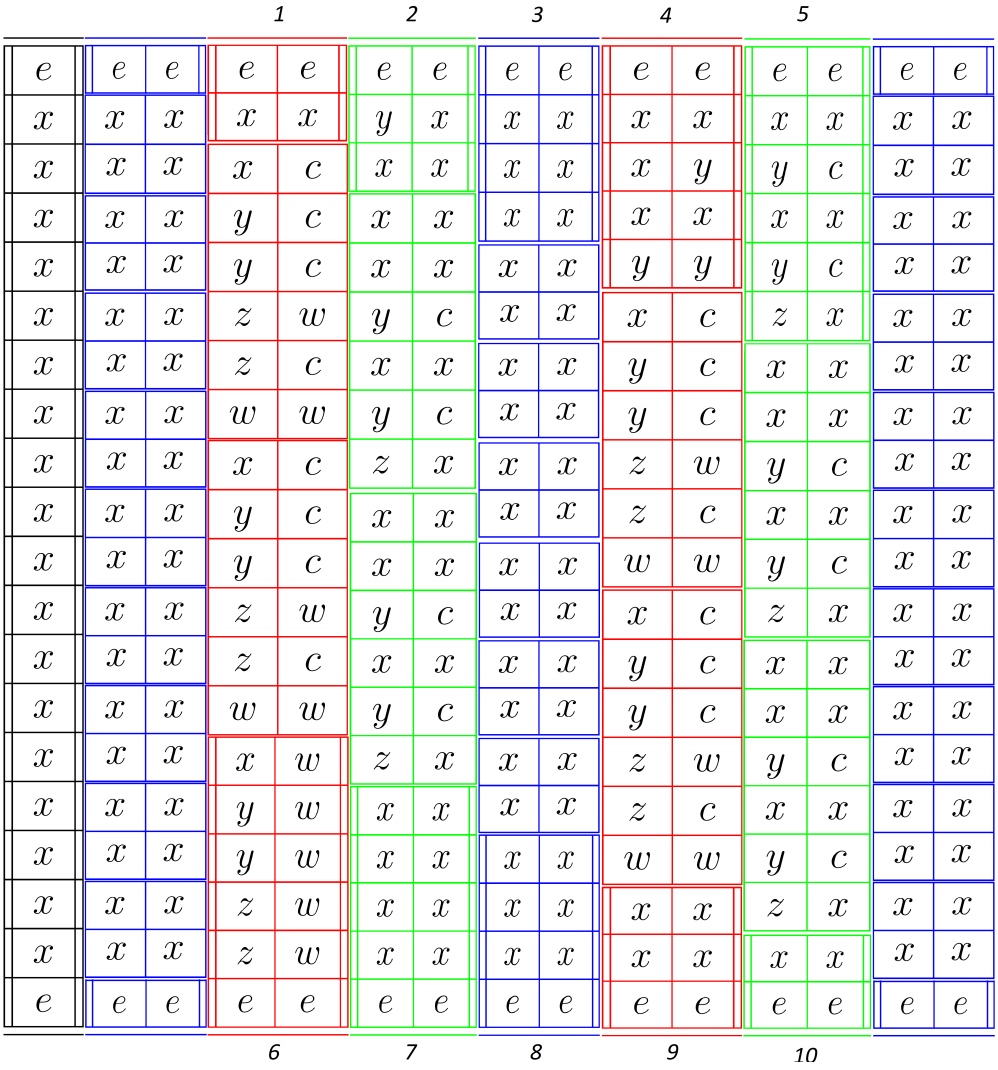


Figure 4.9: First design: the general form of a complete assembly

them may encode that symbol with a different combination of bases at its sticky end. In order to keep our figures simpler, sticky ends that encode the same symbol with different combination of bases are colored differently.

Each tile in the first level of correction must attach to four other tiles, namely to three tiles from a triplet of XOR tiles and one tile from the next XOR triplet. For correct positioning of the first-level correction tiles, the first tile of each XOR triplet must have different sticky ends from the other two XOR tiles and the first-level correction tile must

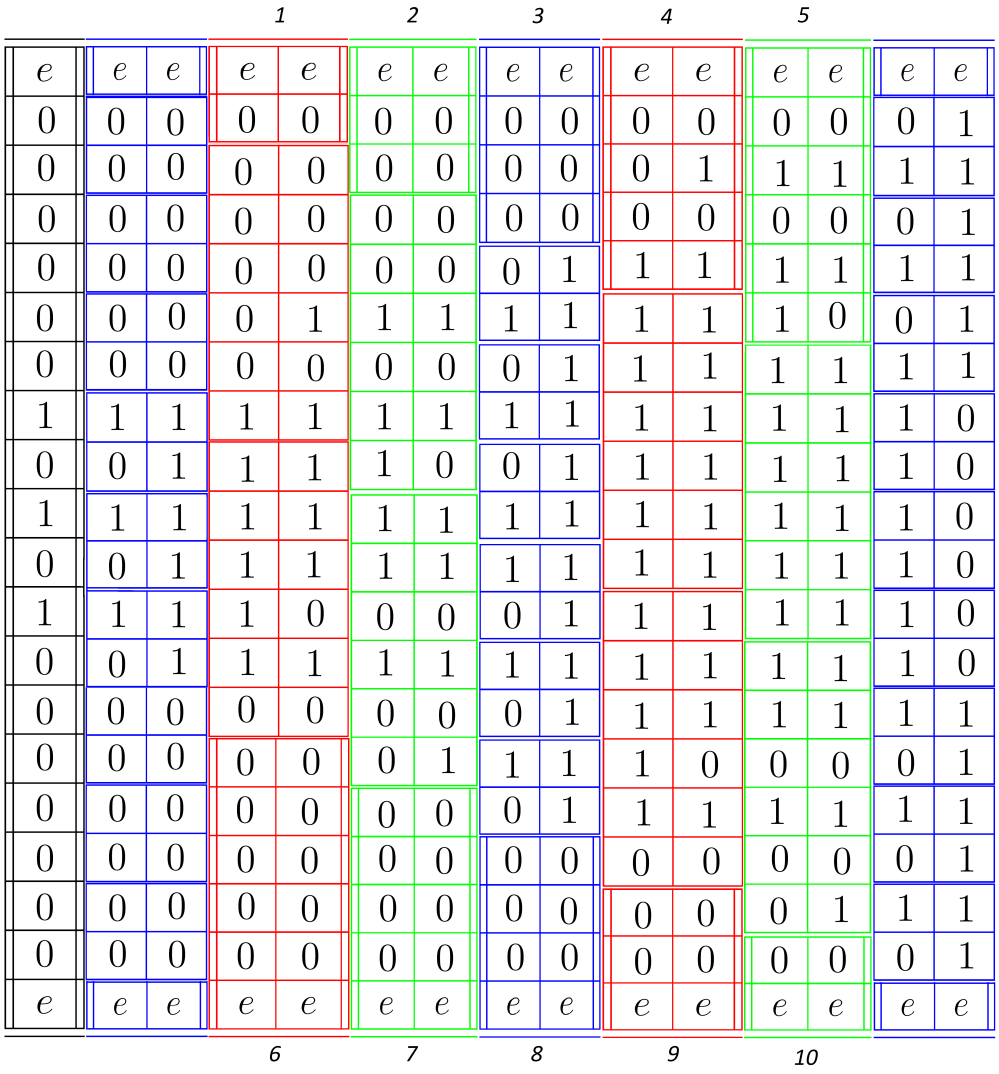


Figure 4.10: First design: an example of a complete assembly

have complementary sticky ends to those of the XOR tiles, as shown in Figure 4.13. Different encoding of the sticky ends (black and red) ensure that XOR and first-level correction tiles attach to each other correctly. Each colored sticky end may represent any symbol; the difference in color simply implies different encoding of the sticky ends, even if the represented symbols are identical. Using the same principles, we must ensure that the second-level correction tiles attach to first level at the precise correct positions. The two top sticky ends of a first-level correction tile must have different sticky ends from

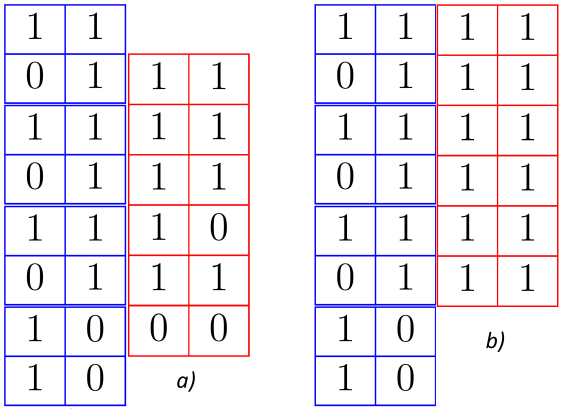


Figure 4.11: First design: example of (a) correct and (b) incorrect positioning

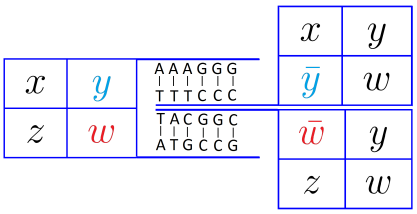


Figure 4.12: Attachment of sticky ends y and w to complementary ends $\bar{y}$ and $\bar{w}$

the remaining sticky ends and the second-level correction tiles must have complementary sticky ends to them, as shown in Figure 4.13. Once again, we arrange the sticky ends of the second-level correction tiles so that the XOR tiles attach at the right position. The first tile of each XOR triplet should have on the left side different sticky ends from the other two tiles in the triplet, in order to maintain the correct positioning of subsequent tiles, as shown in Figure 4.13.

Overall, our first design requires five different kinds of sticky ends, each of them encoding two possible symbols (bits `0` and `1`). In addition, two complementary sticky ends are needed for encoding symbol `e` at the boundaries. So, in total we need 11 different single DNA strands and their complementary 11 DNA strands to encode properly all sticky ends of our tiles.

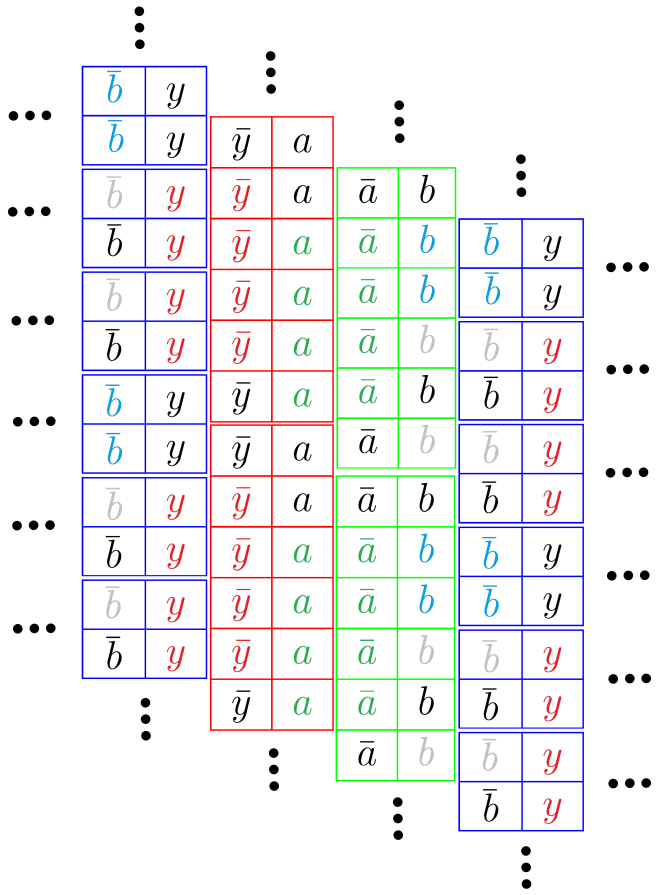


Figure 4.13: Correct positioning of tiles in consecutive XOR rows and correction levels

4.2.5 Theoretical Analysis

To assess the properties of our first design, we are going to perform a theoretical analysis and calculate the probability of an error to propagate given a probability p for a single sticky end to attach incorrectly at some point in the assembly.

Original Design For an error to propagate in the original design all it takes is an incorrect output of a XOR tile. In order to have an incorrect output, one of the two left sticky ends must attach incorrectly (probability p), but not the other (probability $(1-p)$). If both of them attach incorrectly, the output would be the correct one, because of the characteristics of the XOR function! Figure 4.14 shows an example. So, the probability

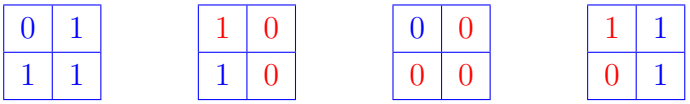


Figure 4.14: No errors (left), one error (middle), and two errors (right) in a XOR tile

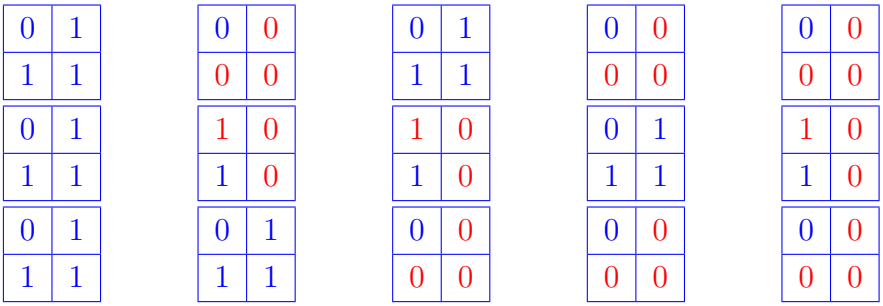


Figure 4.15: No errors (left), two errors (middle), and three errors (right) in a triplet

of a propagated error in the original design that will generate a wrong lattice is:

$$p_{\text{orig}} = 2p(1-p)$$

First Design with Errors Only at XOR Tiles Initially, we assume that errors can occur only at XOR tiles and we calculate the probability per tile for an error to propagate throughout the assembly. Under our first design, an erroneous output from a single XOR tile in a triplet will be locally corrected and will not propagate throughout the assembly. However, an error will propagate, if at least two out of the three XOR tiles in a triplet give simultaneously incorrect outputs. The probability of one XOR tile to give incorrect output was calculated above and is denoted as p_{orig}. There are three different combinations of errors occurring simultaneously in exactly two out of the three XOR tiles in a triplet (probability $p_{\text{orig}}^2(1-p_{\text{orig}})$ for each one) and one case of errors occurring simultaneously at all three of them (probability p_{orig}^3), as shown in Figure 4.15. So, the probability for our first design, assuming errors only at XOR tiles, to generate an a propagated error is:

$$p_{\text{XOR}} = 3p_{\text{orig}}^2(1-p_{\text{orig}}) + p_{\text{orig}}^3 = 3p_{\text{orig}}^2 - 2p_{\text{orig}}^3 = p_{\text{orig}}^2(3-2p_{\text{orig}})$$

First Design with Errors at XOR and Correction Tiles We are going to calculate the probability of propagated errors by focusing on each row/level of tiles separately. In a row of XOR tiles the probability for the error to propagate is the one calculated above:

$$p_{\text{XOR}} = p_{\text{orig}}^2(3 - 2p_{\text{orig}})$$

At the first level of correction tiles, the top five left sticky ends of a correction tile attach to a triplet of XOR tiles in order to implement error detection and correction. The bottom left sticky end attaches to the first tile of the next triplet of XOR tiles to transfer information for the next XOR operation. So, for an error to propagate, it must occur under circumstances either in the top five left sticky ends or in the bottom left one. Specifically, for an error to propagate from the top five sticky ends, the tile must wrongly detect and correct an error that in fact did not occur. Using Figure 4.16 as a guide, it is easy to see that this can happen, if at least two of x, y, z end up carrying wrong information. There are three pairs: Tile 1 and x, Tile 2 and y, Tile 3 and z. End x will receive wrong information from Tile 1, if only one of them is incorrect, that is either the output of Tile 1 is incorrect (probability p_{orig}) and x attaches correctly (probability $1-p$), or the output of Tile 1 is correct (probability $1-p_{\text{orig}}$) and x attaches incorrectly (probability p). Therefore, the probability of such a mismatch between Tile 1 and x is:

$$p_{1,x} = p_{\text{orig}}(1-p) + (1-p_{\text{orig}})p = p + p_{\text{orig}} - 2pp_{\text{orig}}$$

For the remaining two pairs, since y and z appear doubled (two sticky ends each), the probability of y's or z's being incorrect is p^2 and the probability of being correct is $1-p^2$. Therefore, the probability of a mismatch for each of these pairs will be

$$p_{2,y} = p_{3,z} = p_{\text{orig}}(1-p^2) + (1-p_{\text{orig}})p^2 = p^2 + p_{\text{orig}} - 2p^2p_{\text{orig}}$$

Therefore, the probability that an error will propagate through the top five stick ends is:

$$p_{\text{level1}} = p_{1,x}p_{2,y}(1-p_{3,z}) + p_{1,x}(1-p_{2,y})p_{3,z} + (1-p_{1,x})p_{2,y}p_{3,z} + p_{1,x}p_{2,y}p_{3,z}$$

An error cannot propagate through the sixth sticky end, because Tile 7 will correct it, even if wrong information passes through w.

At the second level of correction tiles, errors can propagate only through sticky ends of type a or c, since sticky ends of type b do not transfer information to the output.

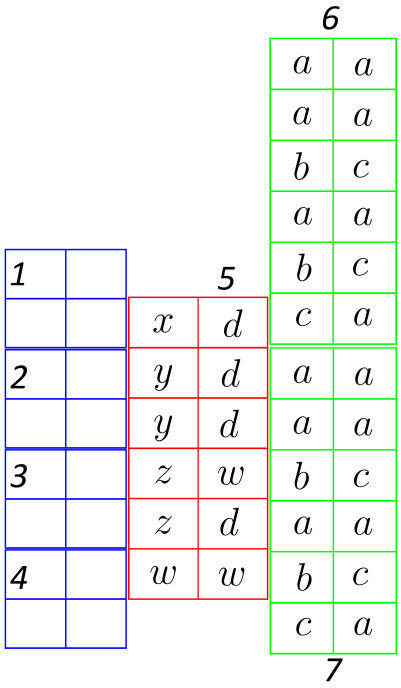


Figure 4.16: First design: explaining error propagation through correction levels

Additionally, errors propagated through a (Tile 7) or c (Tile 6) come only from Tile 5 (outputs d). We have already calculated the probability p_{level1} of an error being propagated to outputs d. There are two cases from this point on. In the first case, either d is correct (probability $1 - p_{\text{level1}}$) and a's are incorrect (probability p^3) or d is incorrect (probability p_{level1}) and a's are correct (probability $1 - p^3$). Therefore, the probability of error propagation in this case will be

$$p_{d,a} = p_{\text{level1}}(1 - p^3) + (1 - p_{\text{level1}})p^3 = p^3 + p_{\text{level1}} - 2p^3 p_{\text{level1}}$$

In the second case, either d is correct (probability $1 - p_{\text{level1}}$) and c is incorrect (probability p) or d is incorrect (probability p_{level1}) and c is correct (probability $1 - p$). Therefore, the probability of error propagation in this case will be

$$p_{d,c} = p_{\text{level1}}(1 - p) + (1 - p_{\text{level1}})p = p + p_{\text{level1}} - 2pp_{\text{level1}}$$

This type of error is in fact the weak spot of the second level of correction tiles and of the whole first design. Other things being equal, errors can propagate much easier through c

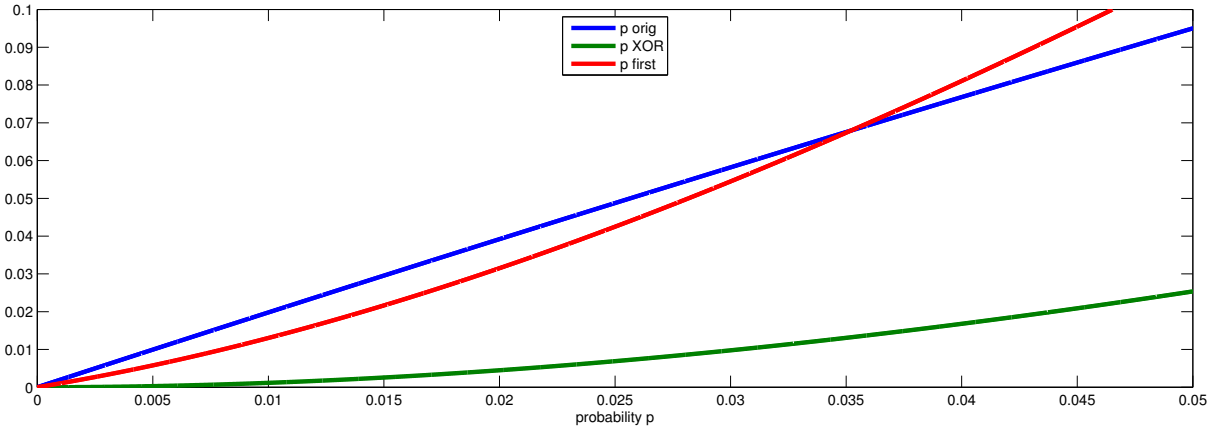


Figure 4.17: Error propagation probability vs. probability of incorrect attachment

than through a; this difference is quantified by the difference in probability values p and p^3. In summary, the probability for an error being propagated beyond the second level of correction tiles under our first design, and therefore to the next row of XOR tiles, is the probability of the intersection of the two cases mentioned above:

$$p_{\text{first}} = p_{d,a} + p_{d,c} - p_{d,a}p_{d,c}$$

It is not clear by comparing the three probability expressions (p_{orig}, p_{XOR}, and p_{first}) which one is better, given that we wish to make this probability as small as possible. As shown in Figure 4.17 for small values of p (below 3.5%), our first design has better chances to build a correct lattice. For larger values of p the original design has better chances than our design to create a correct lattice, but a complete correct lattice is highly unlike in this case. Laboratory experimental results with the original design (described in Chapter 3) showed an average value of error probability per tile of about 1.4% (this is p_{orig} in our notation), which means that the probability p for a sticky end to attach incorrectly is $p = 0.7\%$. For this value of p, we have $p_{\text{orig}} = 1.4\%$, $p_{\text{XOR}} = 0.057\%$, and $p_{\text{first}} = 0.85\%$. At this range, our first design is certainly better than the original one. Moreover, if in our design errors occur only at XOR tiles, the chances for an incorrect lattice are even less.

4.3 Second Design

As seen in Figure 4.17 there is significant difference in error propagation probability between the case where errors occur only at XOR tiles and the case where errors occur everywhere. For that gap responsible is the weak spot at the second level of correction tiles, namely the fact that a single incorrect attachment of sticky end c of a correction tile in the second level (Figure 4.16) directly propagates into the rest of the assembly. Our second design for a self-correcting assembly aims to fix that particular weak spot of our first design. This slightly different and more involved design succeeds in correcting errors in regular XOR tiles, while incurring better control on errors occurring in the additional tiles it introduces.

4.3.1 Correction Tiles

Similarly to the first design, our second design introduces two levels of correction between regular XOR rows. The first level of correction tiles is identical to the one in the first design, described already in Section 4.2.1. Here we focuses only on the second level of correction tiles which differs. By shifting down the positioning of tiles at this level and by a slightly different tile design, we can duplicate the weak sticky end (bottom left end in the first design) to reduce its probability of error and still maintain the desired correction properties. Therefore, the second level of correction tiles in our second design abide to the following template:

x	x
y	c
x	x
y	c
z	x
z	c

This template is a 6×2 tile with six sticky ends on each side and yields eight distinct tiles (see Section A.2.1.3). As before, this second level aims at correcting left-over, propagated errors from the first level. Correction at this level is implemented by the following rules:

if $(y = z)$ then there is no error and $c \leftarrow y$
else if $(y \neq z)$ then there is error in y and $c \leftarrow z$

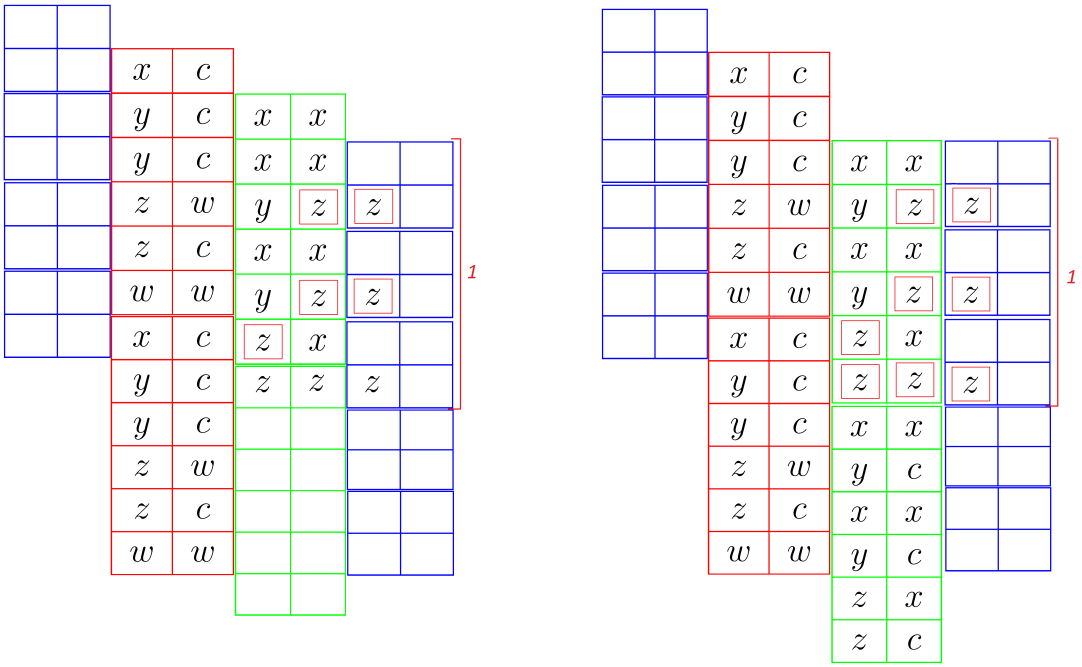


Figure 4.18: Assembly differences between first (left) and second (right) designs

In this template all the different sticky ends are duplicated and for an error to propagate both sticky ends must attach incorrectly. Figure 4.18 will help us understand the way the lattice assembly has altered with the new tiles. In the assembly under the first design a single error at the marked sticky end z suffices to alter the triplet of XOR tiles (marked by 1) in the next row and let the error propagate. By shifting down the green tiles in the second design we can duplicate the marked sticky ends z. This way we drastically reduce the probability of an error to propagate, because both sticky ends must attach incorrectly for the error to propagate.

4.3.2 Boundary Tiles

In our second design the XOR boundary tiles and the first-level correction boundary tiles are identical to those of the first design, described already in Section 4.2.2.

The second-level correction boundary tiles are created by "cutting" the regular correction tiles to the right size. Again, we need two types of lower and upper tiles, because we have two different sizes of boundary tiles in each second level of correction tiles. Our

second-level correction boundary tiles abide to the following templates (left two: upper boundary, right two: lower boundary):

e	e

e	e
y	c
x	c
x	c

x	x
x	x
x	x
e	e

e	e

Error correction in the larger template for the upper boundary is implemented by the following rules:

if $(y = x)$ then there is no error and $c \leftarrow x$
else if $(y \neq x)$ then there is error in y and $c \leftarrow x$

These templates yield a total of eight distinct tiles (see Section A.2.2.3).

4.3.3 Assembly

Figure 4.19 illustrates the use of boundary tiles through a complete ribbon assembly with our second design using the templates described above. Notice that the DNA origami seed must also be given in triplets of two symbols, following our redundancy scheme, and be bounded withing the boundary symbols e. At positions 2 and 5, the two different types of upper boundary tiles for the second level of correction tiles can be seen. Also, at positions 7 and 10, the two different types of lower boundary tiles for the second level of correction tiles can be seen. Figure 4.20 shows a specific instance of this (error-free) assembly originating out of a DNA origami seed of the form `e000000101010000000e`.

4.3.4 Positioning

Once again we must use different encoding of bits at different tiles so we can ensure the correct positioning of tiles in the lattice. The XOR tiles and the first-level correction tiles in our second design are identical to the first design and there is no need for any change in their sticky ends. We just need to change the sticky ends of the second-level correction tiles to ensure that they attach to first level at the precise correct positions. This change is shown in Figure 4.21 in comparison to the first design. The bottom two sticky ends of the second-level correction tiles must be complementary to the two top sticky ends of a

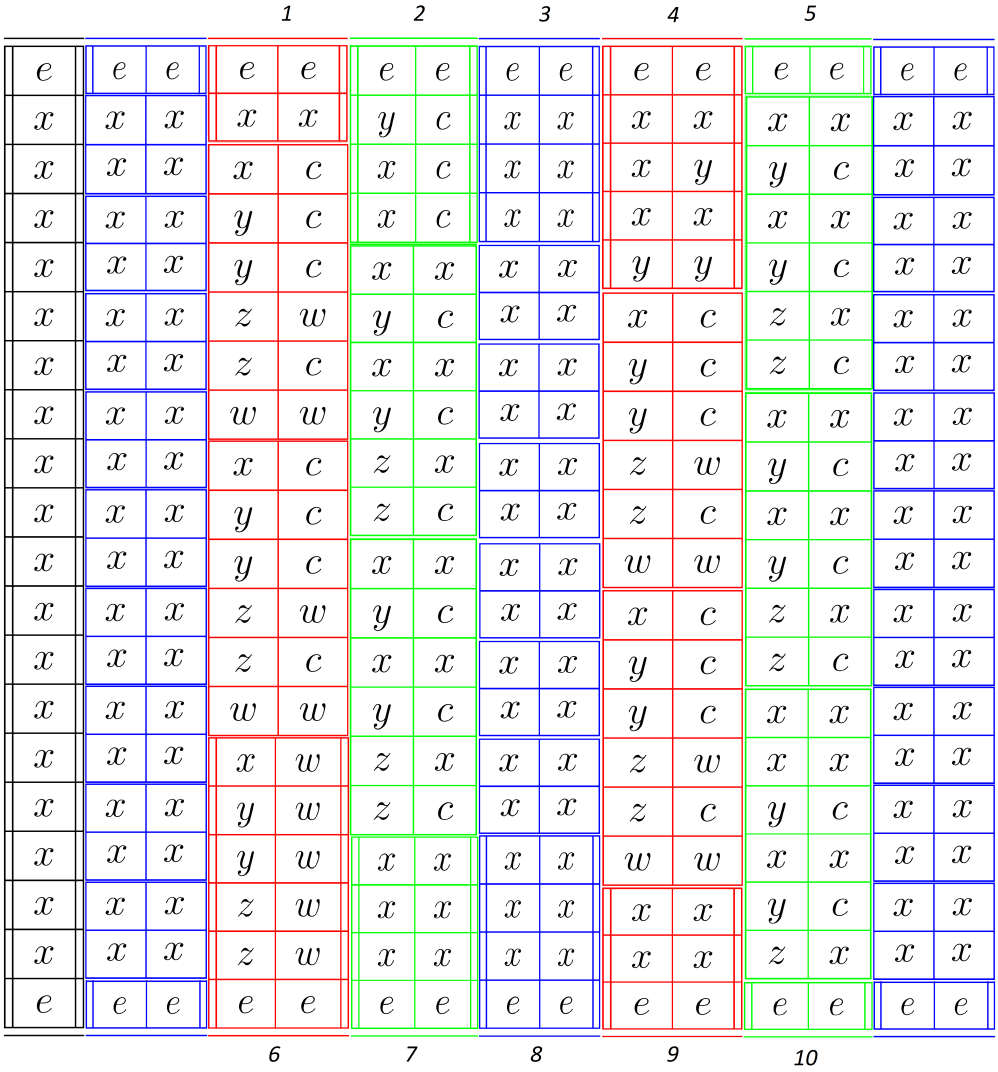


Figure 4.19: Second design: the general form of a complete assembly

first-level correction tile. We also arrange the sticky ends of the second-level correction tiles so that the XOR tiles attach at the right position in the next row.

Overall, our second design requires five different kinds of sticky ends, each of them encoding two possible symbols (bits `0` and `1`). In addition, two complementary sticky ends are needed for encoding symbol `e` at the boundaries. So, in total we need 11 different single DNA strands and their complementary 11 DNA strands to encode properly all sticky ends of our tiles.

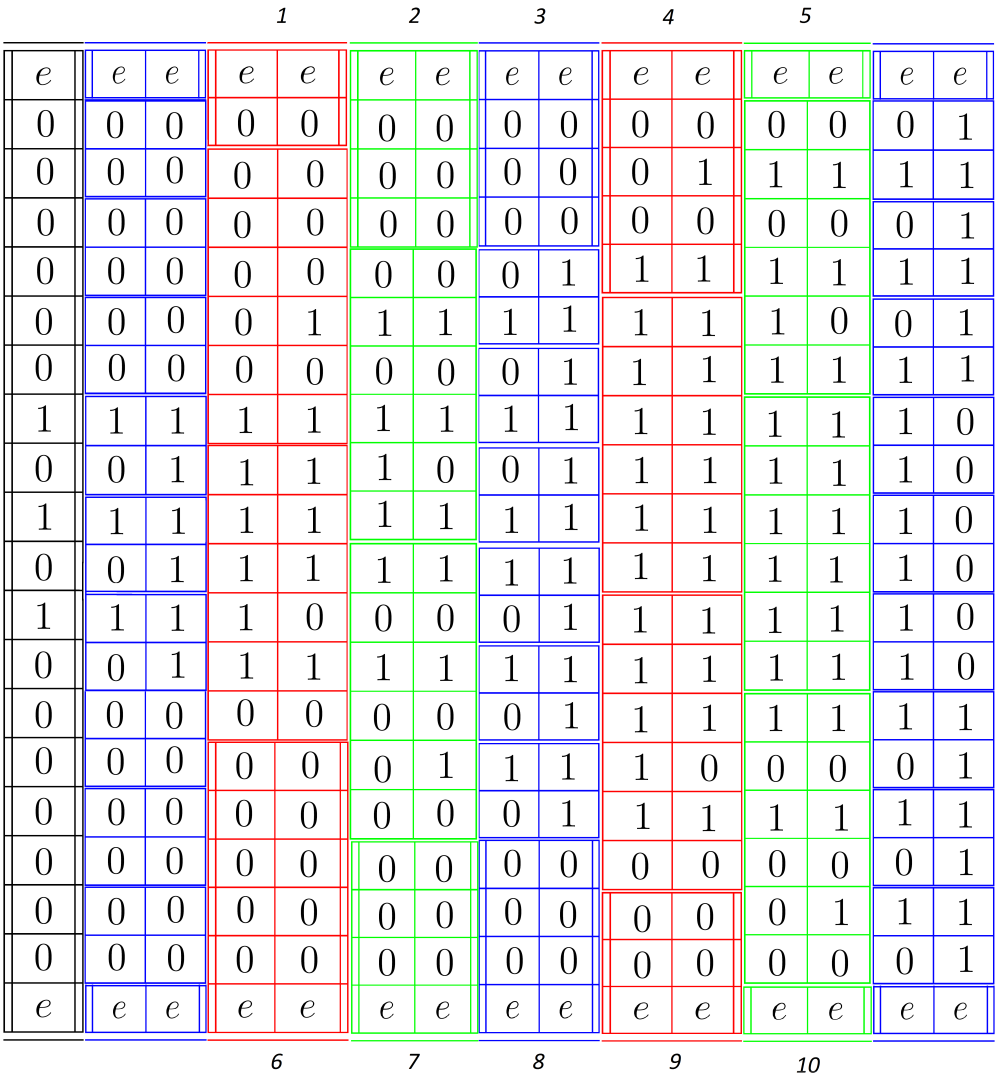


Figure 4.20: Second design: an example of a complete assembly

4.3.5 Theoretical Analysis

The analysis for the error propagation probability under our second design is identical to that of the first design up to the first level of correction. Here, we focus only on the second level of correction.

Using Figure 4.22 as a guide, at the second level of correction tiles, errors can propagate only through sticky ends of type a or c, since sticky ends of type b do not transfer information to the output. Additionally, errors propagated through a (Tile 7) or c (Tile

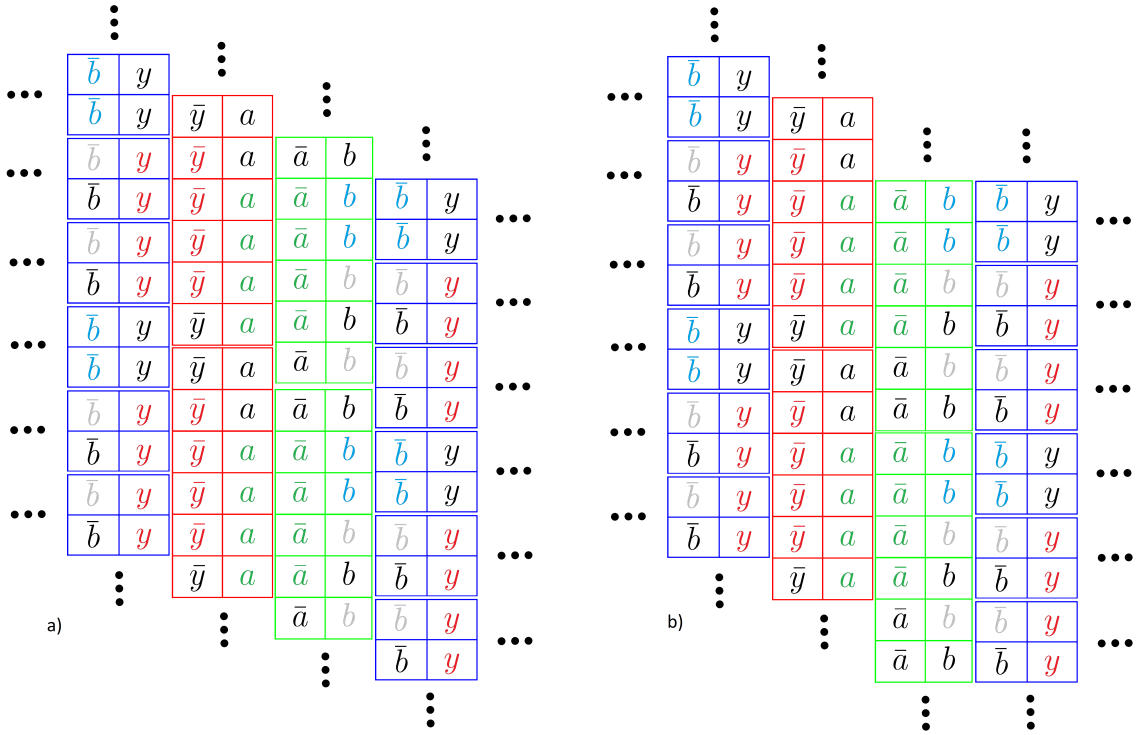


Figure 4.21: Correct positioning of tiles in first (left) and second (right) designs

6) come only from Tile 5 (outputs d). We have already calculated the probability p_{level1} of an error being propagated to outputs d (see Section 4.2.5). There are two cases from this point on. In the first case, either d is correct (probability $1-p_{\text{level1}}$) and a's are incorrect (probability p^2) or d is incorrect (probability p_{level1}) and a's are correct (probability $1-p^2$). In the second case, either d is correct (probability $1-p_{\text{level1}}$) and c's are incorrect (probability p) or d is incorrect (probability p_{level1}) and c's are correct (probability $1-p^2$). Therefore, the probability of error propagation is the same in each of theses cases:

$$p_{d,ac} = p_{\text{level1}}(1-p^2) + (1-p_{\text{level1}})p^2 = p^2 + p_{\text{level1}} - 2p^2 p_{\text{level1}}$$

Finally, the probability for an error being propagated beyond the second level of correction tiles under our second design, and therefore to the next row of XOR tiles, is the probability of the intersection of the two cases mentioned above:

$$p_{\text{second}} = 2p_{d,ac} - p_{d,ac}^2$$

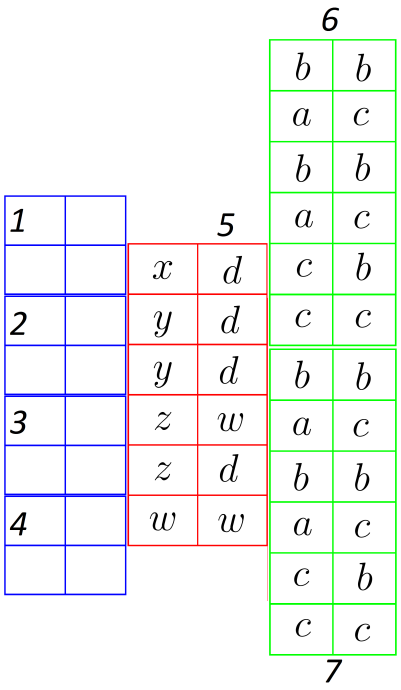


Figure 4.22: Second design: explaining error propagation through correction levels

As shown in Figure 4.23 our second design significantly improves over the first design. For values of p below 7.5%, our second design has better chances to build a correct lattice than both the original and our first design. More specifically, for the experimentally verified value of $p = 0.7\%$ we have $p_{\text{orig}} = 1.4\%$, $p_{\text{XOR}} = 0.057\%$, $p_{\text{first}} = 0.85\%$, and $p_{\text{second}} = 0.16\%$. At this range, our second design is clearly a winner over the first design and the original one.

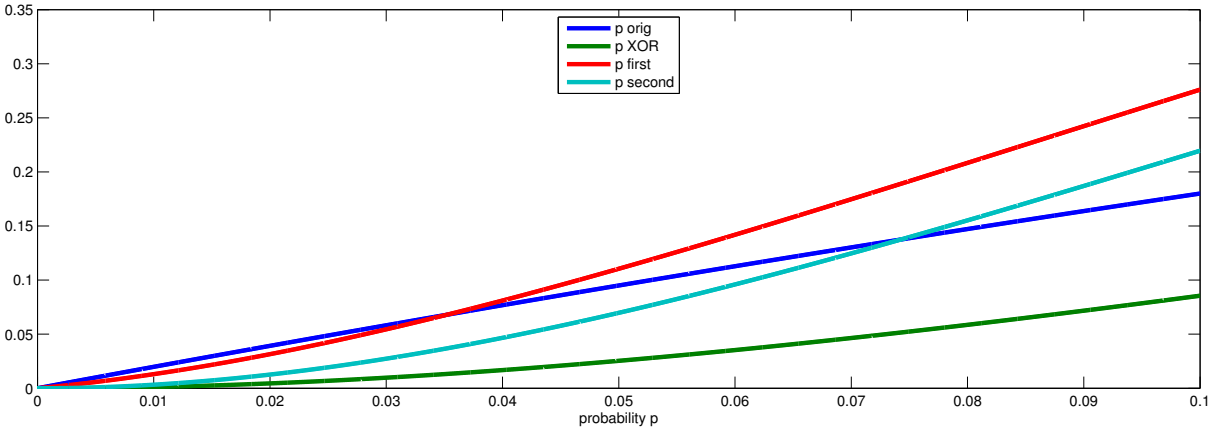


Figure 4.23: Error propagation probability vs. probability of incorrect attachment

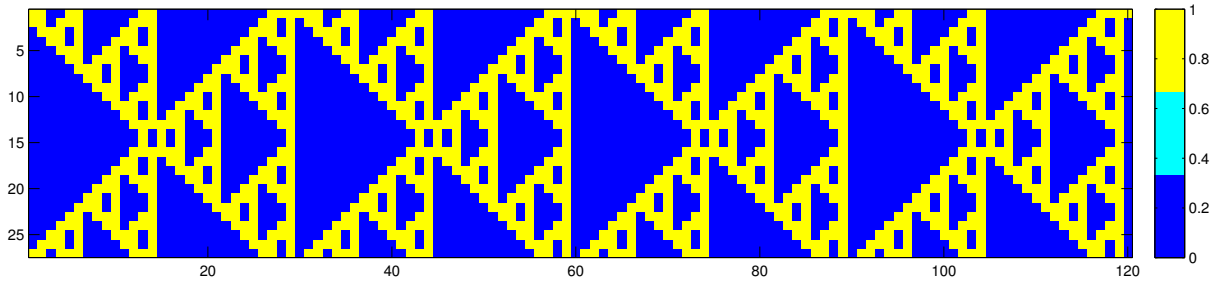
Chapter 5

Implementation

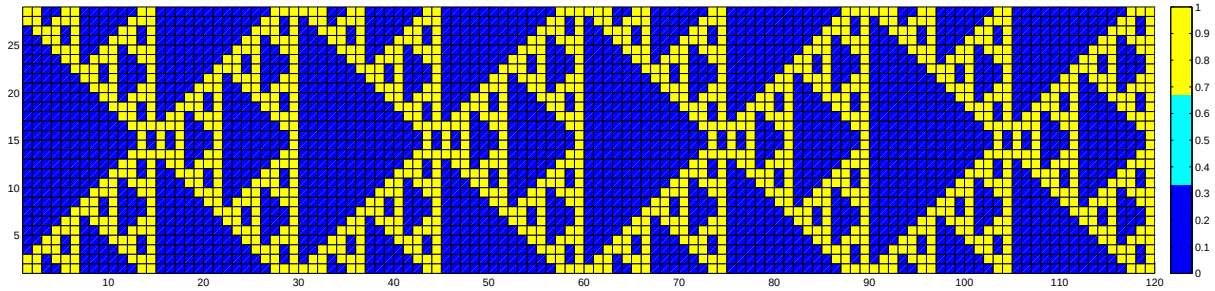
Unfortunately, real experimental evaluation of our designs is beyond our capabilities due to lack of expertise and facilities. In order to test our designs in some informative way, we implemented a simulation of self-assembly in MATLAB. The idea for the simulation is simple. We represent the lattice with a two dimensional array. Every cell in the array represents a sticky end, e.g. a XOR tile with two sticky ends on each side is represented by a 2×2 sub-array in the lattice array. Similarly, we define the tile library, so that each tile is given as a two-dimensional array. We allow for an infinite number of each kind of tile and we let them shuffle and attach to each other starting at the first row of the lattice representing the seed. In order to see the results of the simulation we print the two-dimensional lattice with either `imagesc` (each sticky end is a pixel in the image) or `surf` (each sticky end is a cell in the grid), as shown in Figure 5.1.

The function that simulates the self-assembly accepts the following parameters:

- Size `X` - The width of the lattice in tiles.
- Size `Y` - The length of the lattice in periods.
- Seed `row` - An array of size 1×2`X` representing the seed row of the lattice.
- Error `prob` - Error probability of a sticky end to attach incorrectly.

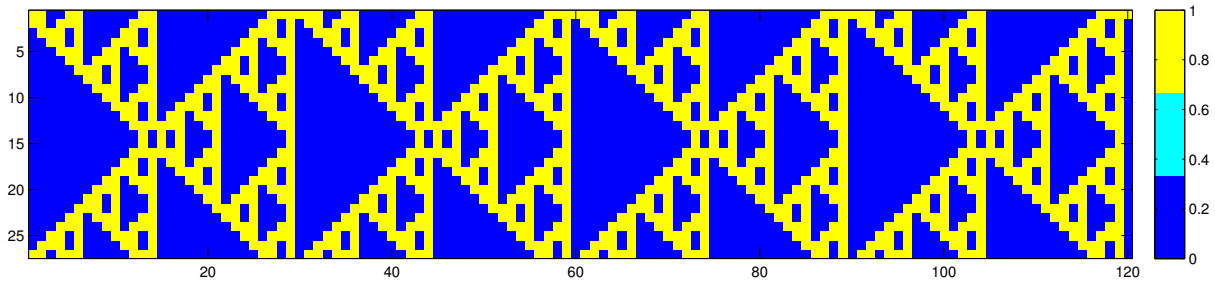


(a) Self-assembly lattice printed with imagesc

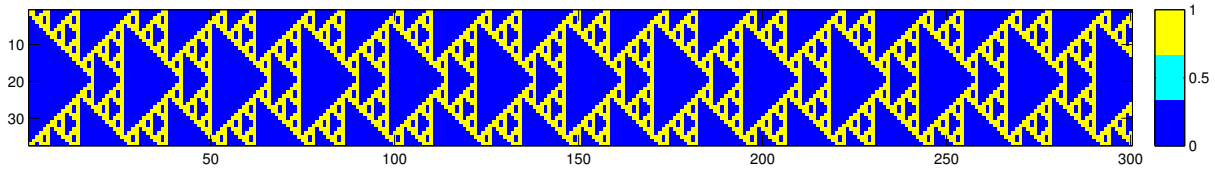


(b) Self-assembly lattice printed with surf

Figure 5.1: Visualization of self-assembly lattices in MATLAB.



(a) Lattice 14-tiles wide and 4-periods long



(b) Lattice 19-tiles wide and 7-periods long

Figure 5.2: Examples of lattices with different width

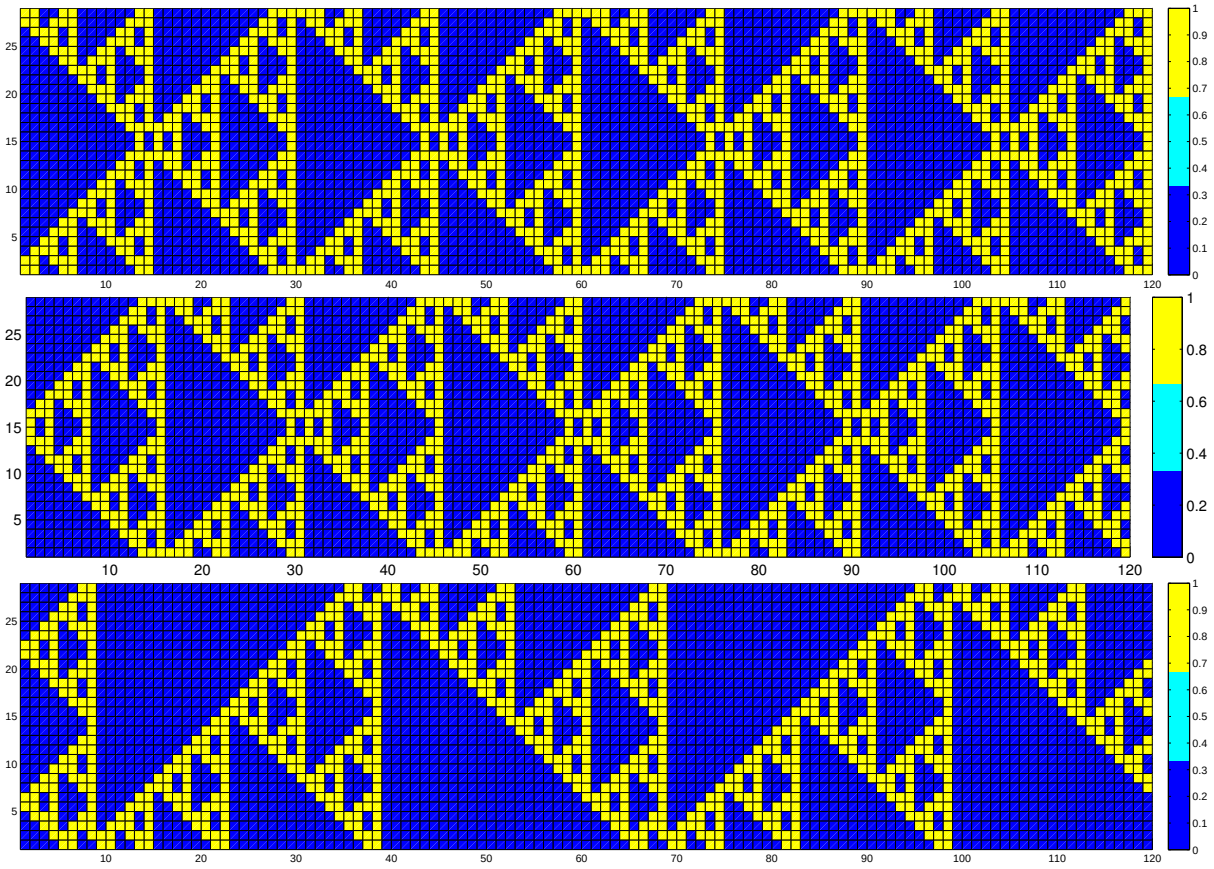


Figure 5.3: Examples of lattices with different seed rows

The simulation function returns the following two-dimensional arrays:

- `correct` - Lattice of the original design without errors
- `incorrect` - Lattice with random errors occurring consistently with `prob`
- `first` - Lattice implemented with the first design with the same errors as the incorrect lattice at the XOR tiles and random errors at the rest of the tiles.
- `second` - Lattice implemented with the second design similarly to the `first` lattice.

A key parameter in the simulation is the value of the error probability. Figure 5.4 illustrates the effect of `prob` on the resulting lattices.

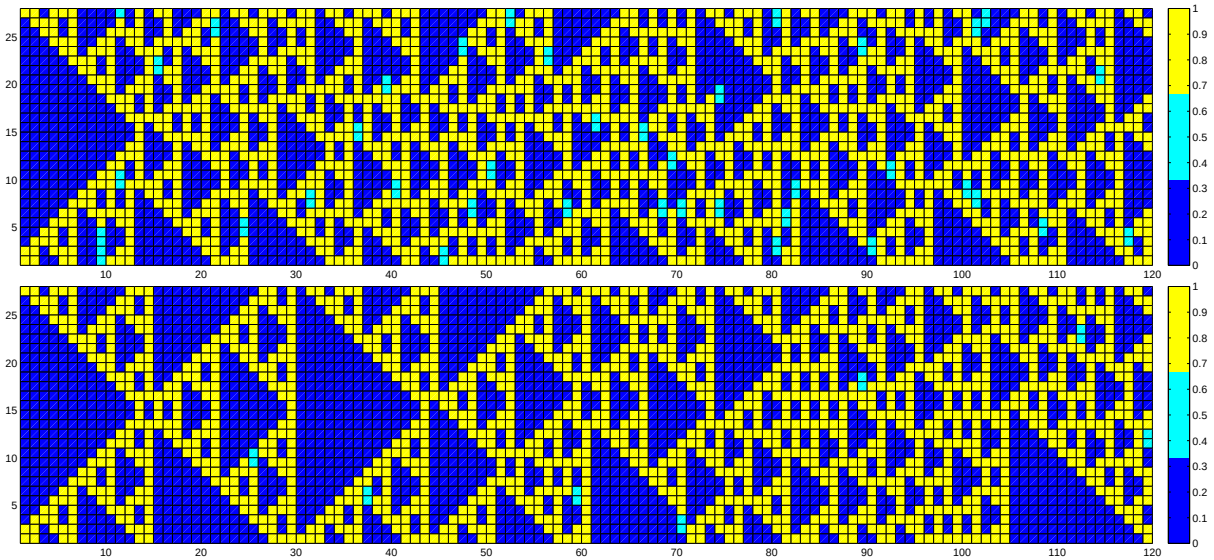


Figure 5.4: Examples of how the error probability parameter affects lattices

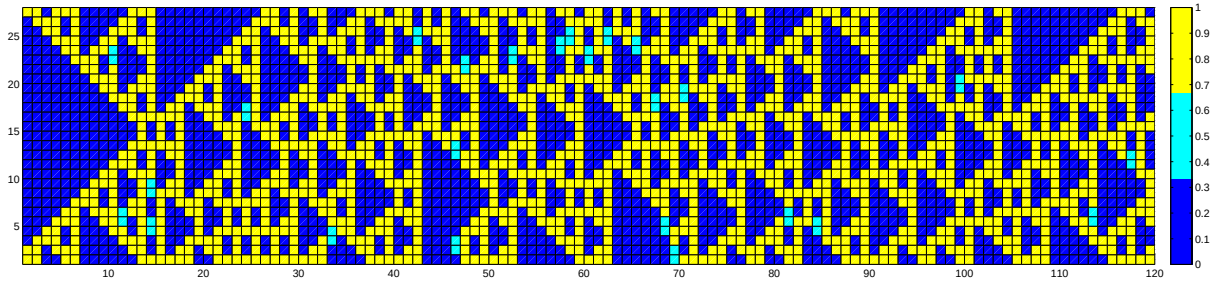
Chapter 6

Simulation Results

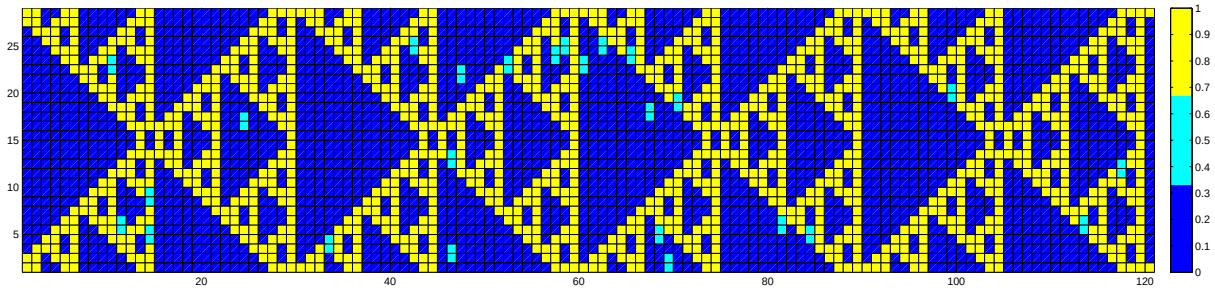
Using our assembly simulation, we tested our designs on numerous examples. Some representative examples are presented in this chapter. All simulations have been conducted with error probability $p = 0.7\%$ for a sticky end to attach incorrectly. This value has been chosen, because it has been experimentally determined through laboratory experiments, as described in Chapter 3. All simulation lattices are 14 tiles wide for the original design and 42 tiles wide for our design (three times wider because of the redundancy) and 4 periods long. Therefore, each assembly with the original design contains about 1600 tiles, whereas each assembly with our designs contains about 8000 tiles. In each example we present performance of (a) the original design, (b) our designs with errors only at XOR tiles, and (c) our first and second designs with errors at all tiles. Also, in our designs we show two different views. The first (XOR view) shows the full lattice really is, whereas the second (full view) shows only the nominal XOR tiles (without the redundant tiles and the correction levels), so that a comparison to the original design can be made.

6.1 Assembly Scars

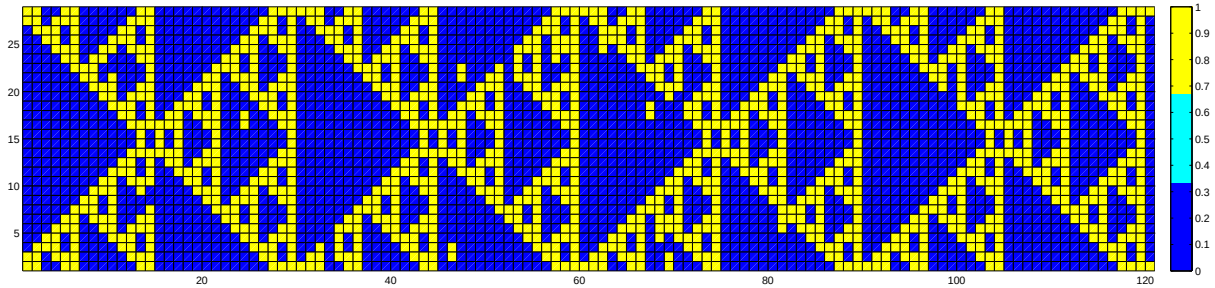
Our designs can detect an error and prevent it from propagating, however it is impossible to detach an incorrectly attached tile. As a result such incorrect tiles remain in the lattice and leave some kind of "scars" in the lattice pattern, as shown in Figure 6.1. These scars will be visible in the examples that follow, even if all errors are detected and corrected.



(a) Original design with random errors marked in cyan



(b) New design with random (corrected) errors marked in cyan



(c) New design with random (corrected) errors unmarked (scars are visible)

Figure 6.1: Example of scars left behind after correction of incorrect tiles

6.2 Random Errors

In this set of experiments, each design faces (different) random errors, which appear with fixed probability p at each sticky end. Figure 6.2 shows the performance of the original design. Errors propagate and the lattice pattern is completely gone. Figure 6.3 shows the performance of our designs (in XOR view for comparison). In the top example, several errors have occurred, but all of them have been detected and corrected. In the contrary, in the bottom example, near the end of the first period, errors occurred at two out of

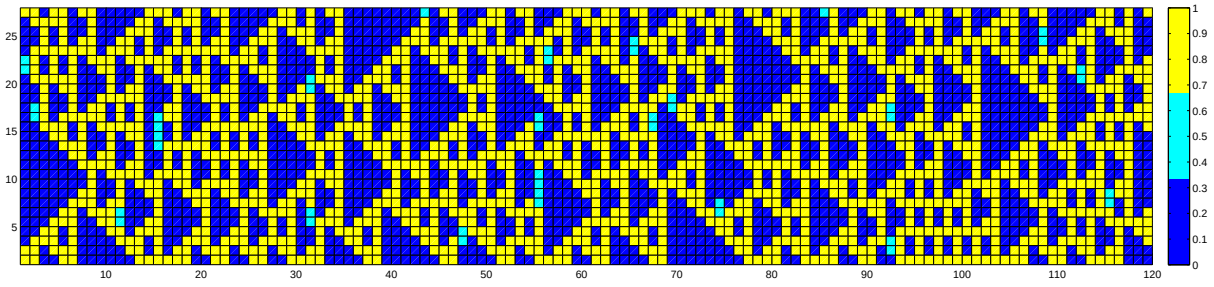


Figure 6.2: Original design with random errors.

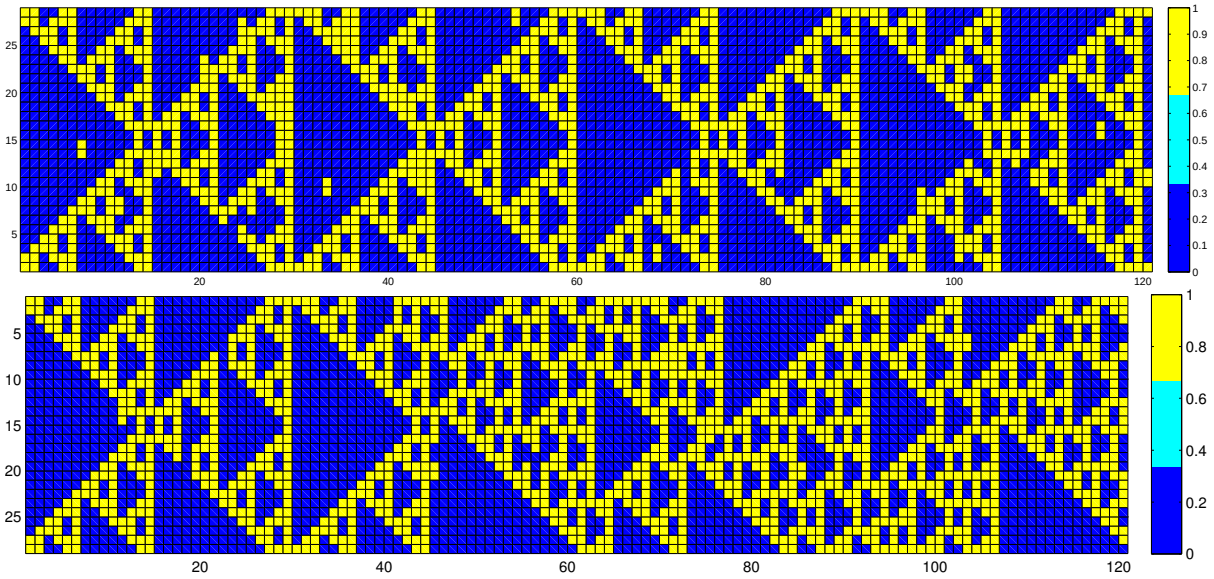


Figure 6.3: Our designs with random errors at XOR tiles only (XOR view)

the three XOR tiles in a triplet and the error propagated ruining the correct pattern. Figure 6.4 shows the same examples in full view. It is worth noting the presence of scars in both views. The full view contains more scars because it shows errors at all redundant XOR tiles, whereas the XOR views shows only errors occurring at the nominal XOR tiles.

The next set of experiments involves random errors allowed to occur at any tile. Figure 6.5 shows an example with the first design presented in both views. Even though the generation of a completely correct lattice is almost impossible, the results are still better than the original design. It can be clearly seen that the pattern maintain its period for a long time, before it is finally interrupted by propagated errors. Figure 6.6 shows

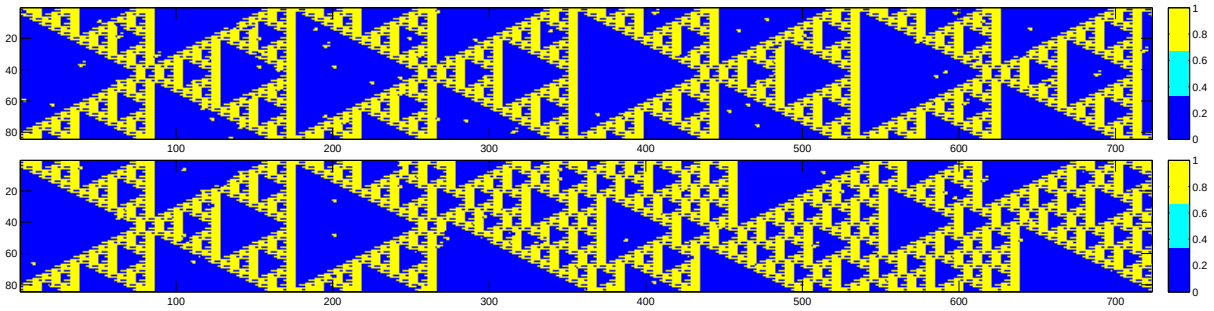


Figure 6.4: Our designs with random errors at XOR tiles only (full view)

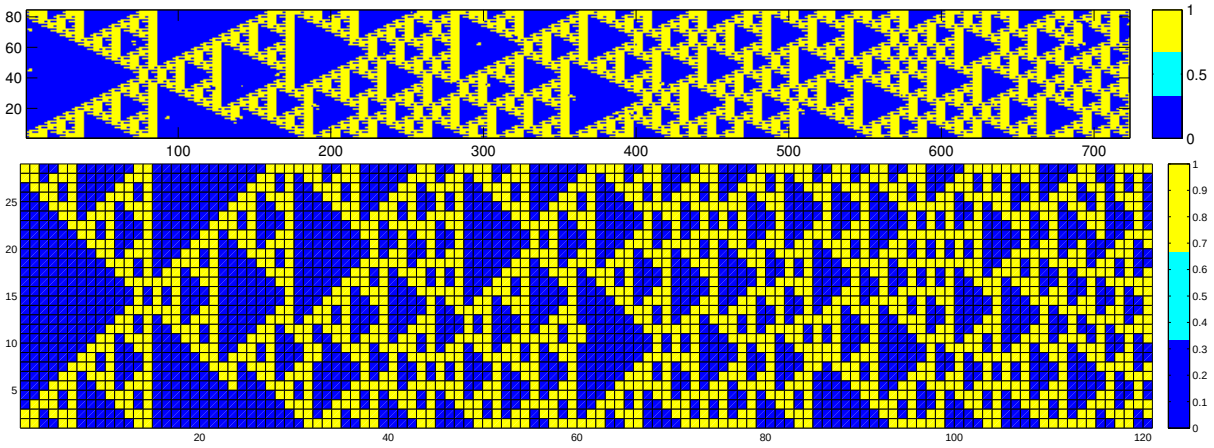


Figure 6.5: First design with random errors allowed at any tile

two examples with the second design in XOR view. In the top example, several errors occurred and were detected and corrected, but at same point before second period an error propagated and interrupted the correct pattern. In contrast, in the bottom example all errors were corrected and the correct lattice was generated. Figure 6.6 shows the same examples in full view.

6.3 Identical Errors

In this section, we present in detail two examples where errors occur at exactly the same positions in the lattices for all designs. The aim of these experiments is to compare the performance of the different designs on equal (identical) terms. The procedure we adopted

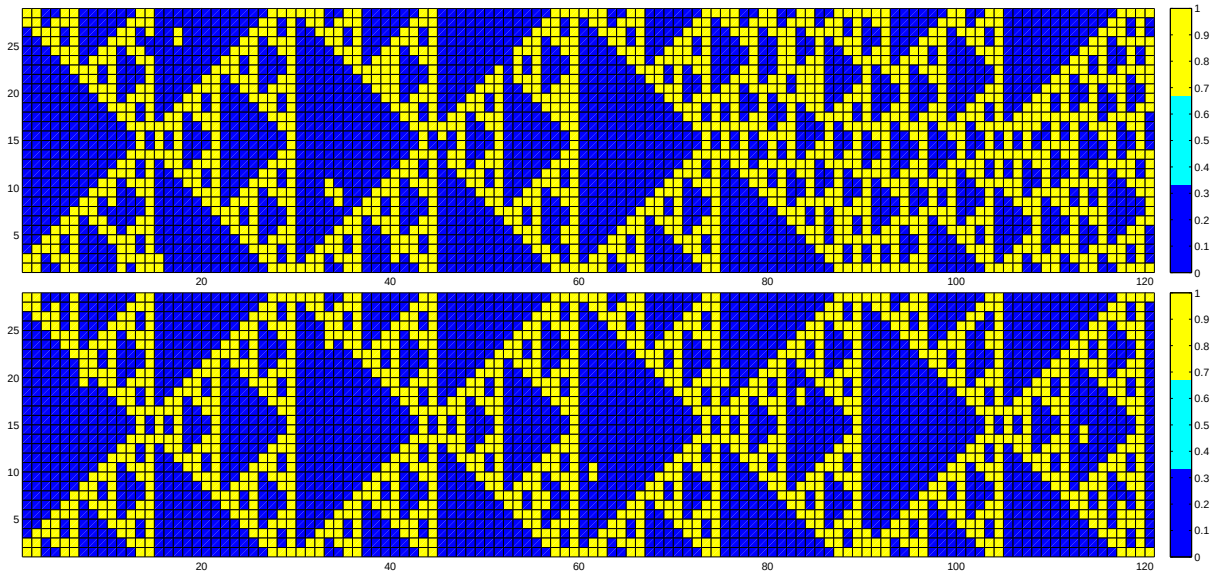


Figure 6.6: Second design with random errors allowed at any tile (XOR view)

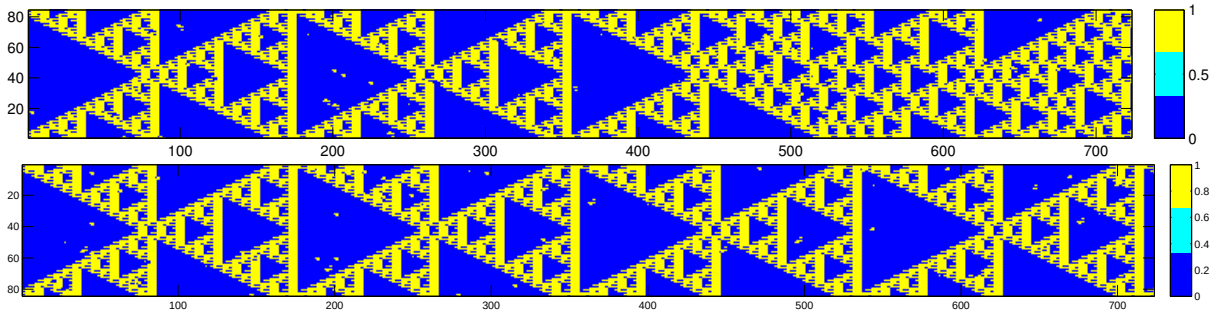


Figure 6.7: Second design with random errors allowed at any tile (full view)

is as follows. First, we run an assembly with the original design with the errors occurring in random as before, but this time we store the exact positions where errors occurred. Then, we run an assembly with our proposed designs with errors only at XOR tiles, but we intentionally insert the recorded errors to the first tile of each triplet triplet, while the other two tiles in the triplet suffer randoms errors independently. Again, we store all error positions. Next, we run an assembly with our first design with errors at any tile. This time, the errors in XOR triplets are the recorded ones, while correction tiles suffer randoms errors independently. Once again, we store all positions where errors occurred

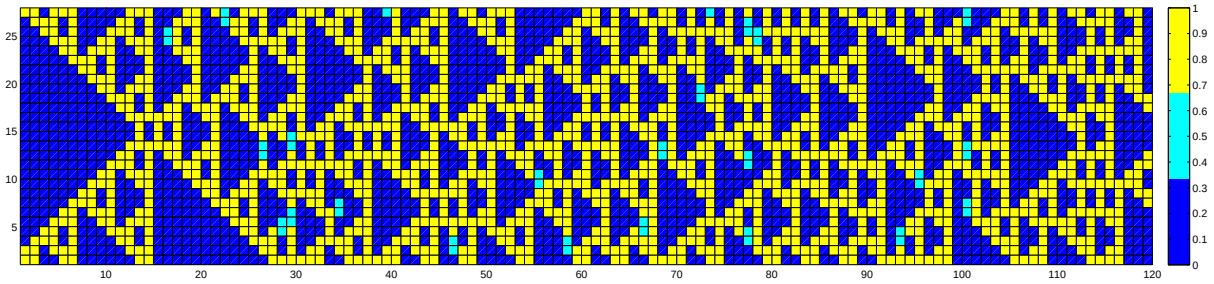


Figure 6.8: Ex. 1: Original design with identical random errors (marked in cyan)

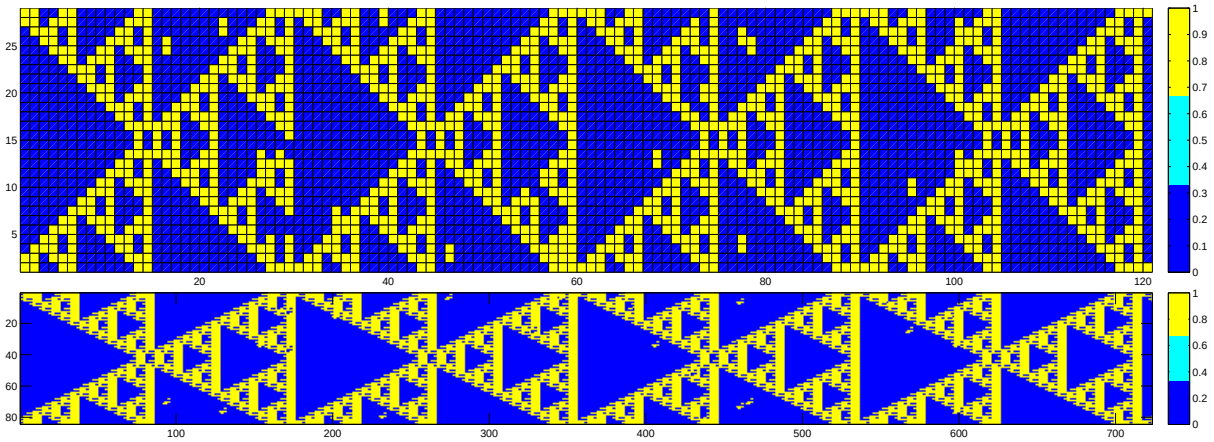


Figure 6.9: Ex. 1: Our designs with identical random errors at XOR tiles (both views)

and finally we run an assembly using the recorded errors in XOR rows and first levels of correction, while the second-level correction tiles suffer randoms errors independently.

Regarding the first example, Figure 6.8 shows the performance of the original design, whereas Figure 6.9 shows the performance of our designs, when errors occur only at XOR tiles. It can be clearly seen that all the errors were corrected and the pattern of the lattice was maintained. The scars of the corrected errors are visible. Figure 6.10 shows the performance of our first design, when errors occur at any tile. Although the overall behavior of the first design is better than the original one, in this example there is no real improvement from the original design. This is because of the weak spot at second level of correction tiles discussed in previous chapters. Figure 6.11 shows the performance of our second design, when errors occur at any tile. As we can see, all the errors were corrected and the pattern of the lattice was maintained. Once again, the scars of the corrected

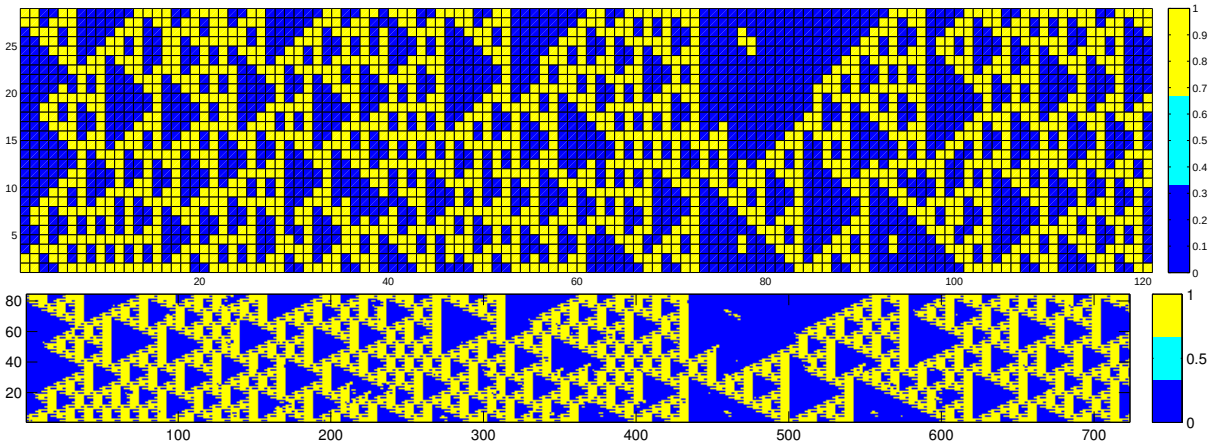


Figure 6.10: Ex. 1: First design with identical random errors at any tile (both views)

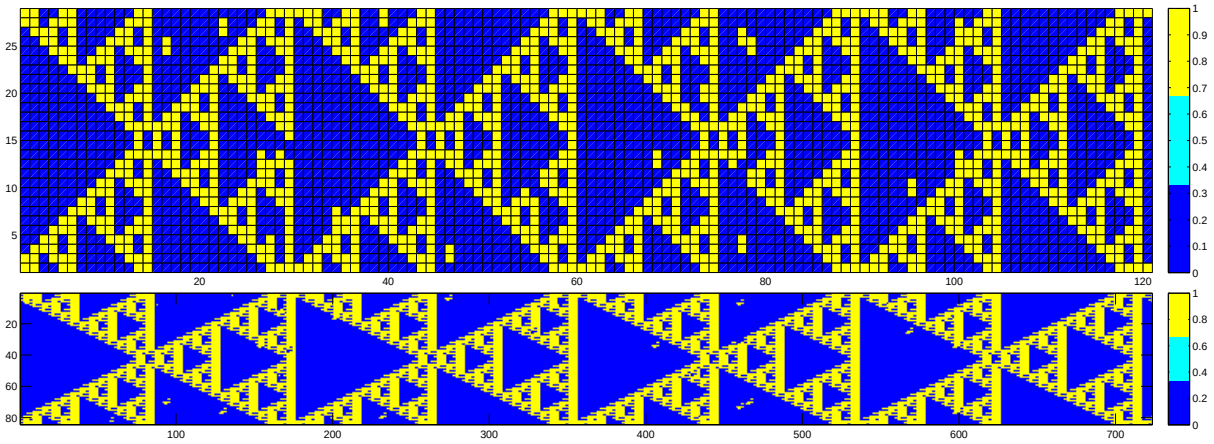


Figure 6.11: Ex. 1: Second design with identical random errors at any tile (both views)

tiles are visible.

Regarding the second example, Figure 6.12 shows the performance of the original design, whereas Figure 6.13 shows the performance of our designs, when errors occur only at XOR tiles. Errors were corrected and the pattern of the lattice was maintained. Figure 6.14 shows the performance of our first design, when errors occur at any tile, whereas Figure 6.15 shows the performance of our second design, when errors occur at any tile. In this example, in contrast to the previous one, the lattice generated by the second design is still incorrect, but still better than the original. Specifically, the design

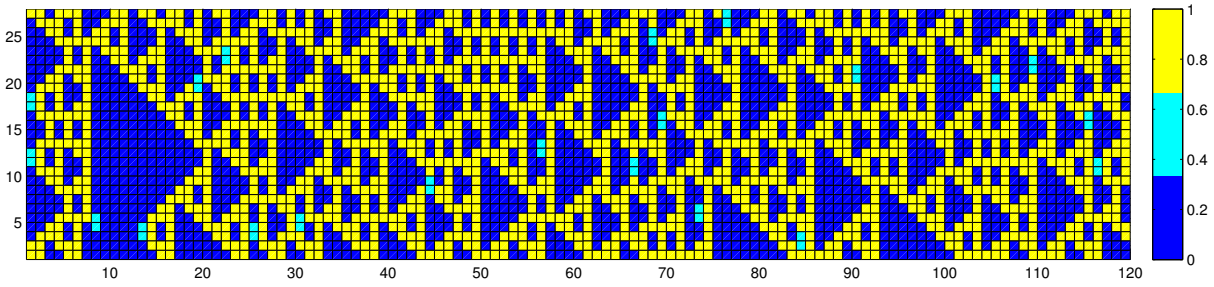


Figure 6.12: Ex. 2: Original design with identical random errors (marked in cyan)

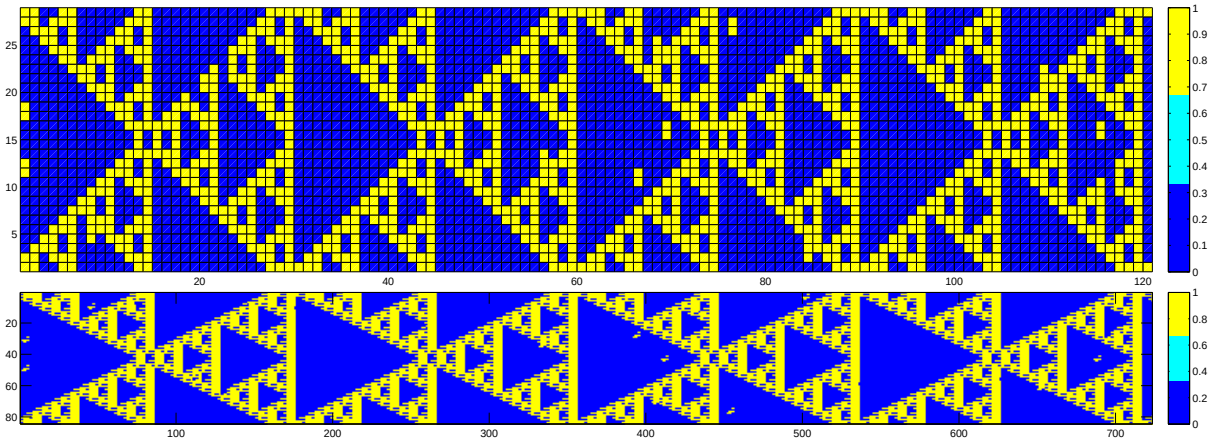


Figure 6.13: Ex. 2: Our designs with identical random errors at XOR tiles (both views)

corrected several errors and almost three period of the pattern is correct. This proves the overall better behavior of our second design.

6.4 Comparison

In order to have a comparative view of the simulation results we ran a stress test for all designs. Specifically, we ran 1000 simulated assemblies for each design for all error probability values ranging from 0% to some large value with step 0.1%. In each case, we counted how many tiles are different compared to the ones in the correct lattice and we calculated the percentage difference between the resulting lattice under each design and the correct lattice.

Figure 6.16 shows the comparison results for values of p ranging from 0% to 50%.

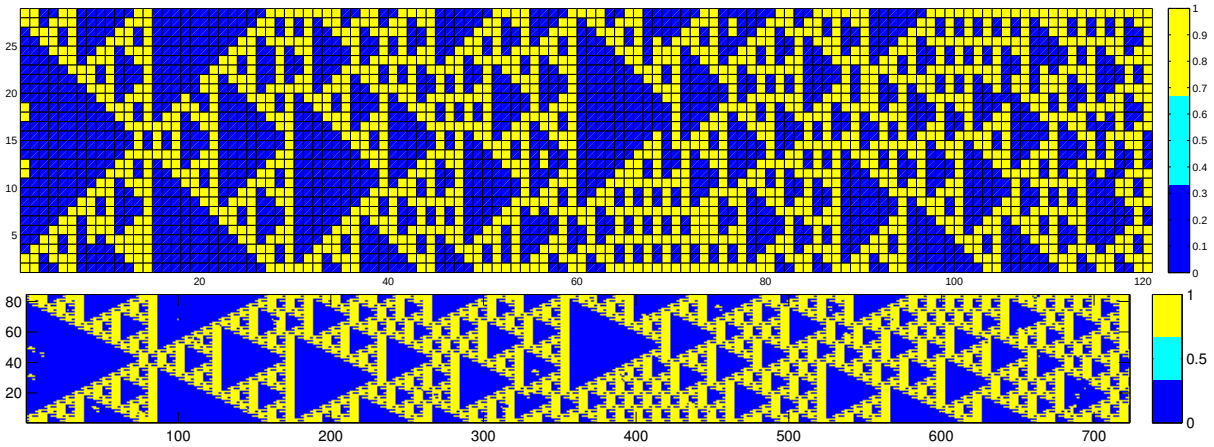


Figure 6.14: Ex. 2: First design with identical random errors at any tile (both views)

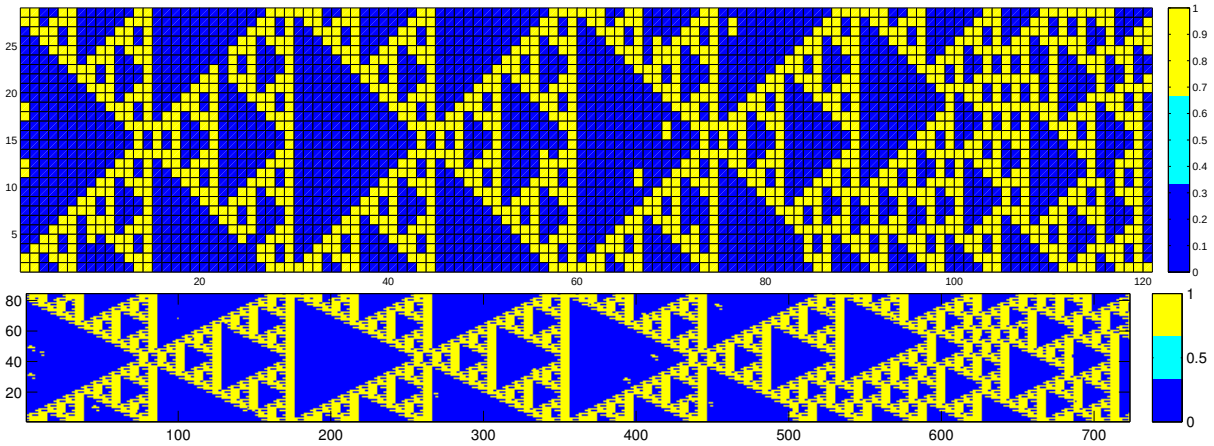


Figure 6.15: Ex. 2: Second design with identical random errors at any tile (both views)

The first design and the original one have the same behavior for values of p less than 2.5%, after which point our design has slightly better behavior. The second design has significantly better results than the first and the original ones and it is close to the ideal, but unrealistic, scenario, where errors occur only at XOR tiles. Also our designs have as upper bound the 25% difference from the correct lattice in contrast to the original design that exhibits a linear growth.

Figure 6.17 shows the comparison results for values of p ranging from 0% to 15%. It also includes the 95% confidence interval bars. Again we see that the first design and

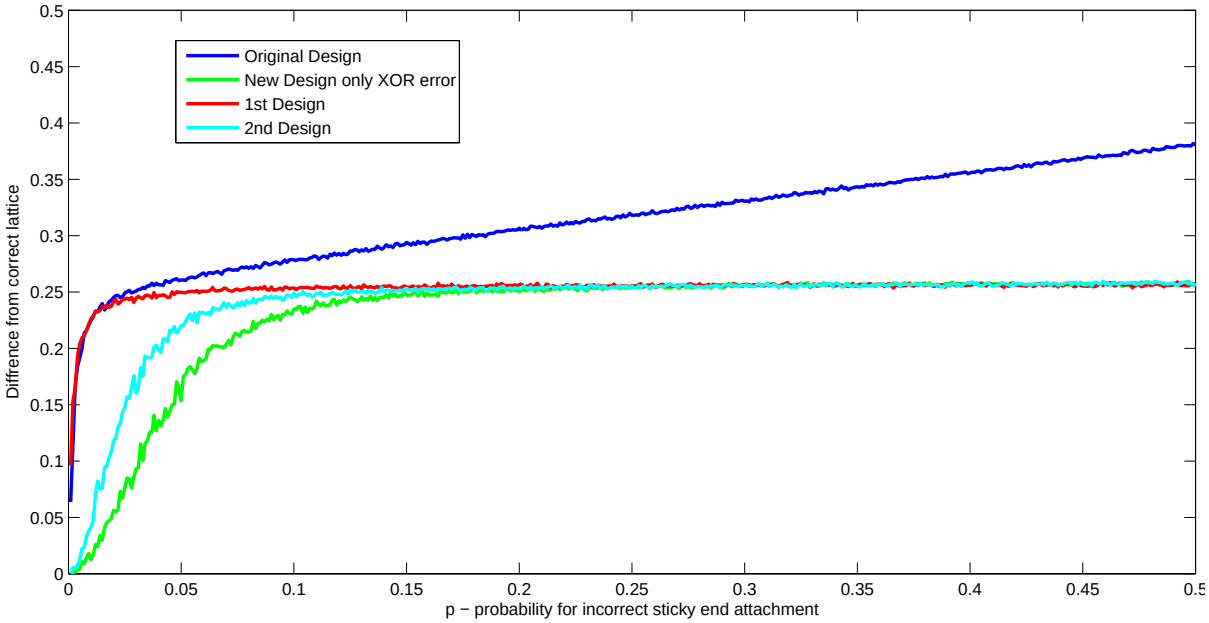


Figure 6.16: Experimental comparison of lattice distortion for $p \in [0.0, 0.5]$

the original one have the same behavior for low values of p, whereas the second design exhibits significantly better results than both of them.

Figure 6.18 shows the comparison results for values of p ranging from 0% to 5%. It also includes the 95% confidence interval bars. Even in this tiny range, the performances of the different designs follow the same pattern, however differences are amplified.

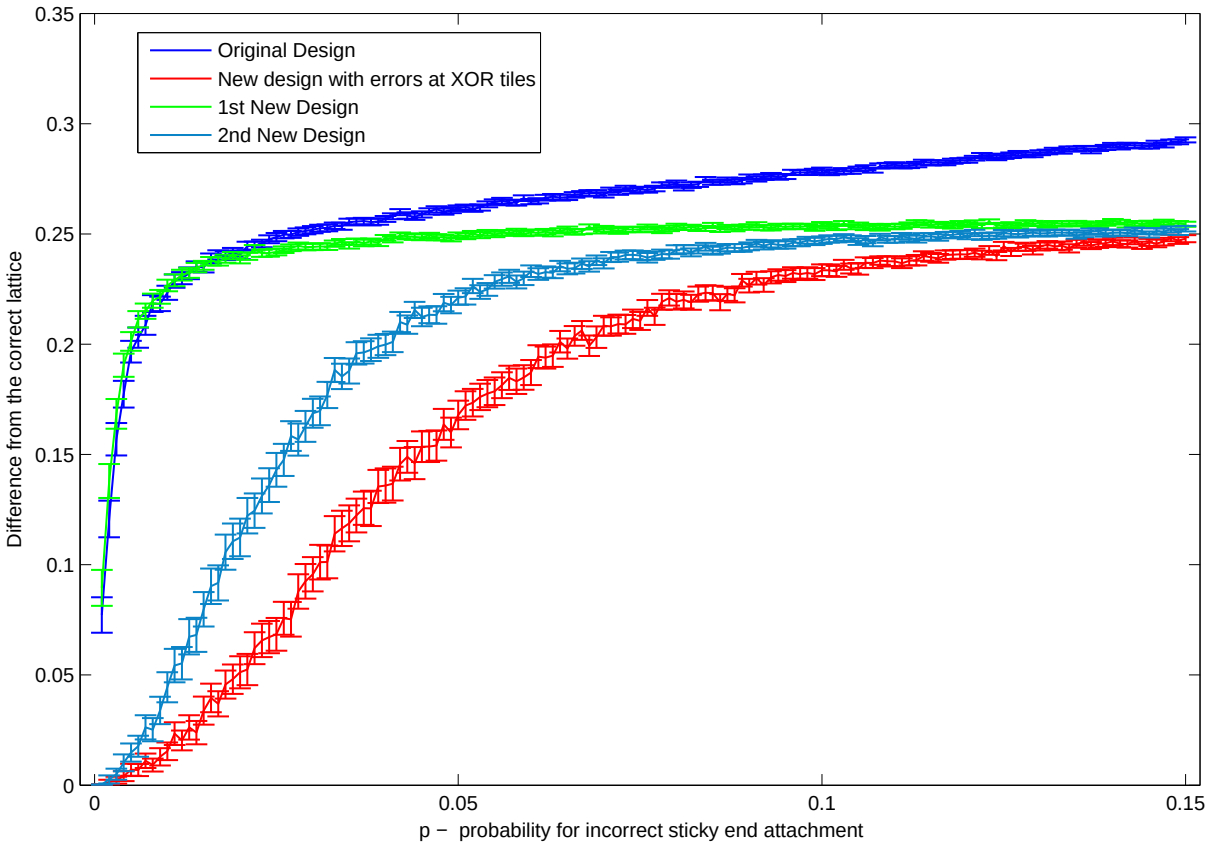


Figure 6.17: Experimental comparison of lattice distortion for $p \in [0.00, 0.15]$

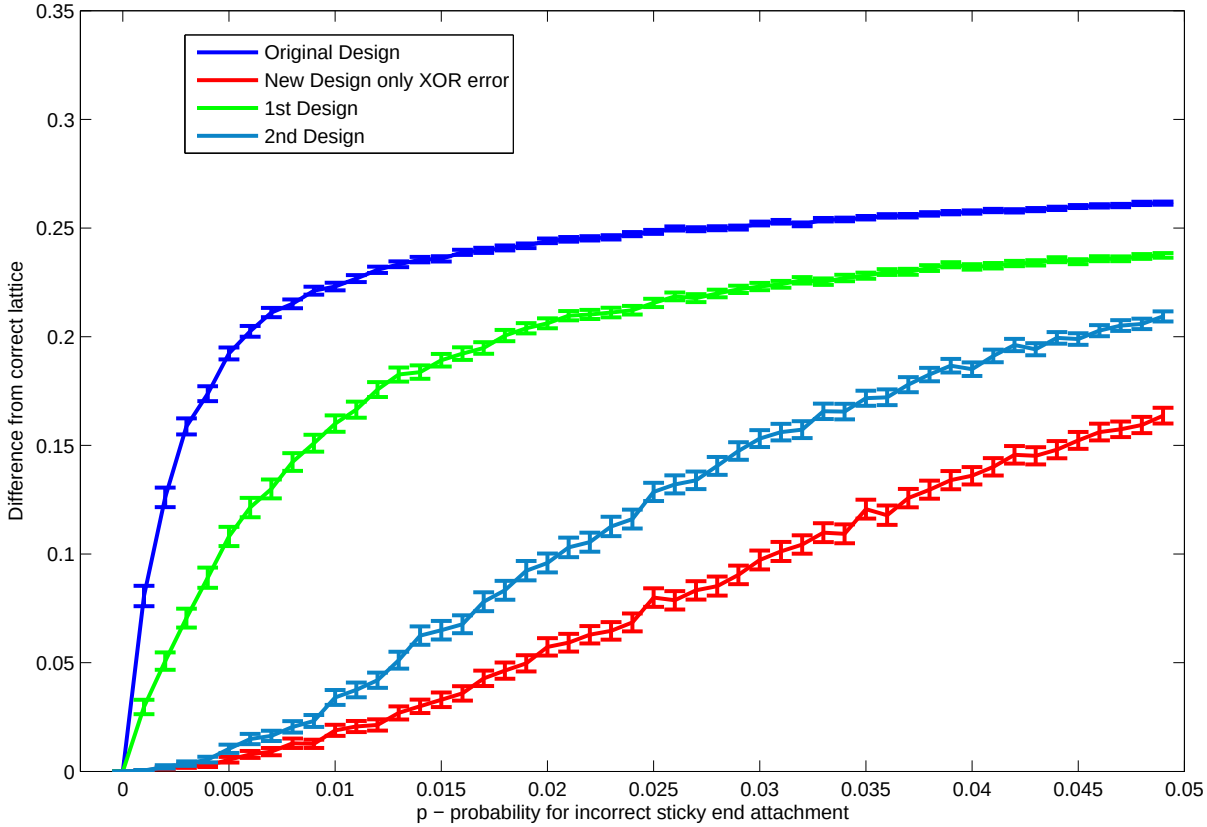


Figure 6.18: Experimental comparison of lattice distortion for $p \in [0.00, 0.05]$

Chapter 7

Conclusion

Error-correction in DNA self-assembly can be challenging, because to achieve it requires some kind of trade-off. In our case, the trade-off was between reliability and complexity. We achieved better reliability by reducing the probability of error propagation, but we sacrificed the simplicity of the original design. However, it seems that this error correction cannot be achieved without increased complexity.

7.1 Advantages and Disadvantages

Our proposed designs offer several advantages:

Reliability The incorporation of error correction reduces the rate of error propagation (slightly for the first design, significantly for the second design). With our designs a single error cannot corrupt the entire pattern, so they have better chances to generate a correct lattice.

Reusability Our proposal for error correction tiles fits other designs, not just XOR operations on which we focused on. Specifically, if both output sticky ends of the non-error-correcting tiles are the same, then we can use the exact same error correction tiles that we used for the XOR function. Even if the output sticky ends are different the same design can be used to implement error correction by minor changes only in the tile library.

On the other hand, our proposed designs come with some disadvantages:

Assembly Size Our designs increase the area of the lattice by a factor of three in each dimension. We increased the width of the ribbon by tripling the number of XOR tiles to achieve redundancy and we tripled the length of the ribbon by introducing two levels of error correction tiles.

Number of Tiles Our designs increase of the number of tiles in the lattice. For the same computation we need five times as many tiles compared to the original design.

Tile Library Our designs increase the size of the tile library. For the original design we needed 10 different tiles, whereas for our first design we need a total of 66 tiles and for our second design a total of 58 different tiles.

7.2 Future Work

Our work can be extended in multiple ways. First, it may be tested on other self-assembly schemes, including binary operations and/or computations beyond XOR, such as the binary counter and the RAM demultiplexer assemblies described in Chapter 2. Second, it could be tried on non-ribbon assemblies; indeed, much of the added complexity in our proposed designs stems from the variety and the multitude of the required boundary tiles. Third, it could serve as a motivating factor for investigating experimentally the construction of DNA tiles with the properties adopted here (large size, multiple sticky ends, multiple inputs and outputs). Are they feasible? Are they reliable? Are assemblies less/more susceptible to errors? Finally, another direction would be to investigate the possibility of pushing redundancy to a third dimension, e.g. parallel lattices, instead of pushing it into the main assembly.

Appendix A

Tile Library

A.1 First Design

A.1.1 Regular Tiles

A.1.1.1 XOR Tiles

Original tiles

Template:

x	$x \oplus y$
y	$x \oplus y$

Number: 4

Tiles:

0	0
0	0

0	1
1	1

1	1
0	1

1	0
1	0

Redundancy tiles

Template:

x	$x \oplus y$
y	$x \oplus y$

Number: 4

Tiles:

0	0
0	0

0	1
1	1

1	1
0	1

1	0
1	0

A.1.1.2 1st-level correction tiles

Template:

x	c
y	c
y	c
z	w
z	c
w	w

if $(x = y = z)$ then no error and $c = x$
else if $(x = y \neq z)$ then there is error in z and $c \leftarrow x$
else if $(x = z \neq y)$ then there is error in y and $c \leftarrow x$
else if $(y = z \neq x)$ then there is error in x and $c \leftarrow y$

Number: 16

Tiles:

No error detection and correction tiles (4 tiles)

0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	1	1	1	1	0
0	0	0	0	1	1	1	1
0	0	1	1	1	1	0	0

Error detection and correction tiles (12 tiles)

1	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	1
0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	1

0	0	0	0	0	1	1	1
1	0	0	0	1	1	0	1
1	0	0	0	1	1	0	1
0	1	1	1	1	1	1	1
0	0	1	0	1	1	1	1
1	1	1	1	1	1	1	1

1	1	0	1	1	1	1	1
1	1	1	1	0	1	1	1
1	1	1	1	0	1	1	1
0	1	1	0	1	0	0	0
0	1	1	1	1	1	0	1
1	1	0	0	0	0	0	0

A.1.1.3 2nd-level correction tiles

Template:

x	x
x	x
y	c
x	x
y	c
z	x

if $(y = z)$ then no error and $c = y$
else if $(y \neq z)$ then there is error in y and $c \leftarrow y$

Number : 8

Tiles:

No error detection and correction tiles (4 tiles)

0	0
0	0
0	0
0	0
0	0
0	0

0	0
0	0
1	1
0	0
1	1
1	0

1	1
1	1
0	0
1	1
0	0
0	1

1	1
1	1
1	1
1	1
1	1
1	1

Error detection and correction tiles (4 tiles)

0	0
0	0
0	1
0	0
0	1
1	0

1	1
1	1
1	0
1	1
1	0
0	1

1	1
1	1
0	1
1	1
0	1
1	1

0	0
0	0
1	0
0	0
1	0
0	0

A.1.2 Boundary Tiles

A.1.2.1 XOR boundary Tiles

Upper Boundary

Template:

e	e
x	x
x	x
x	x

Number: 2

Tiles:

e	e
1	1
1	1
1	1

e	e
0	0
0	0
0	0

Lower Boundary

Template:

x	x
x	x
x	x
e	e

Number: 2

Tiles:

1	1
1	1
1	1
e	e

0	0
0	0
0	0
e	e

A.1.2.2 1st-level correction boundary tiles

First Type Upper Boundary

Template:

e	e
x	x

Number: 2

Tiles:

e	e
0	0

e	e
1	1

Second Type Upper Boundary

Template:

e	e
x	x
x	y
x	x
y	y

Number: 4

Tiles:

e	e
0	0
0	0
0	0
0	0

e	e
1	1
1	1
1	1
1	1

e	e
0	0
0	1
0	0
1	1

e	e
1	1
1	0
1	1
0	0

First Type Upper Boundary

Template:

x	w
y	w
y	w
z	w
z	w
e	e

if $(x = y = z)$ then no error and $w = x$
else if $(x = y \neq z)$ then there is error in z and $w \leftarrow x$
else if $(x = z \neq y)$ then there is error in y and $w \leftarrow x$
else if $(y = z \neq x)$ then there is error in x and $w \leftarrow y$

Number: 8

Tiles:

0	0
0	0
0	0
0	0
0	0
e	e

1	1
1	1
1	1
1	1
1	1
e	e

1	0
0	0
0	0
0	0
0	0
e	e

0	0
1	0
1	0
0	0
0	0
e	e

0	0
0	0
0	0
1	0
1	0
e	e

0	1
1	1
1	1
1	1
1	1
e	e

1	1
0	1
0	1
1	1
1	1
e	e

1	1
1	1
1	1
0	1
0	1
e	e

Second Type Lower Boundary

Template:

x	x
x	x
e	e

Number: 2

Tiles:

0	0
0	0
e	e

1	1
1	1
e	e

A.1.2.3 2nd-level correction boundary tiles

First Type Upper Boundary

Template:

e	e
x	x
x	x

Number: 2

Tiles:

e	e
0	0
0	0

e	e
1	1
1	1

Second Type Upper Boundary

Template:

e	e
x	x
y	c
x	x
y	c
z	x

if $(y = z)$ then no error and $c = y$
else if $(y \neq z)$ then there is error in y and $c \leftarrow y$

Number: 8

Tiles:

e	e
0	0
0	0
0	0
0	0
0	0

e	e
1	1
1	1
1	1
1	1
1	1

e	e
0	0
1	1
0	0
1	1
1	0

e	e
1	1
0	0
1	1
0	0
0	1

e	e
0	0
1	0
0	0
1	0
0	0

e	e
1	1
0	1
1	1
0	1
1	1

e	e
0	0
0	1
0	0
0	1
1	0

e	e
1	1
1	0
1	1
1	0
0	1

First Type Lower Boundary

Template:

x	x
x	x
x	x
x	x
e	e

Number: 2

Tiles:

0	0
0	0
0	0
0	0
e	e

1	1
1	1
1	1
1	1
e	e

Second Type Lower Boundary

Template:

x	x
e	e

Number: 2

Tiles:

0	0
e	e

1	1
e	e

A.2 Second Design

A.2.1 Regular Tiles

A.2.1.1 XOR Tiles

Original tiles

Template:

x	$x \oplus y$
y	$x \oplus y$

Number: 4

Tiles:

0	0
0	0

0	1
1	1

1	1
0	1

1	0
1	0

Redundancy tiles

Template:

x	$x \oplus y$
y	$x \oplus y$

Number: 4

Tiles:

0	0
0	0

0	1
1	1

1	1
0	1

1	0
1	0

A.2.1.2 1st-level correction tiles

Template:

x	c
y	c
y	c
z	w
z	c
w	w

if $(x = y = z)$ then no error and $c = x$
else if $(x = y \neq z)$ then there is error in z and $c \leftarrow x$
else if $(x = z \neq y)$ then there is error in y and $c \leftarrow x$
else if $(y = z \neq x)$ then there is error in x and $c \leftarrow y$

Number: 16

Tiles:

No error detection and correction tiles (4 tiles)

0	0
0	0
0	0
0	0
0	0
0	0

0	0
0	0
0	0
0	1
0	0
1	1

1	1
1	1
1	1
1	1
1	1
1	1

1	1
1	1
1	1
1	0
1	1
0	0

Error detection and correction tiles (12 tiles)

1	0
0	0
0	0
0	0
0	0
0	0

0	0
1	0
1	0
0	0
0	0
0	0

0	0
0	0
0	0
1	0
1	0
0	0

1	0
0	0
0	0
0	1
0	0
1	1

0	0
1	0
1	0
0	1
0	0
1	1

0	0
0	0
0	0
1	1
1	0
1	1

0	1
1	1
1	1
1	1
1	1
1	1

1	1
0	1
0	1
1	0
1	1
0	0

1	1
1	1
1	1
1	1
1	1
1	1

1	1
1	1
1	1
0	0
0	1
0	0

A.2.1.3 2nd-level correction tiles

Template:

x	x
x	x
y	c
x	x
y	c
z	x

if $(y = z)$ then no error and $c = y$
else if $(y \neq z)$ then there is error in y and $c \leftarrow y$

Number : 8

Tiles:

No error detection and correction tiles (4 tiles)

0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	1	1	0	0	1	1
0	0	0	0	1	1	1	1
0	0	1	1	0	0	1	1
0	0	1	0	0	1	1	1

Error detection and correction tiles (4 tiles)

0	0	1	1	1	1	0	0
0	0	1	1	1	1	0	0
0	1	1	0	0	1	1	0
0	0	1	1	1	1	0	0
0	1	1	0	0	1	1	0
1	0	0	1	1	1	0	0

A.2.2 Boundary Tiles

A.2.2.1 XOR boundary Tiles

Upper Boundary

Template:

e	e
x	x
x	x
x	x

Number: 2

Tiles:

e	e
1	1
1	1
1	1

e	e
0	0
0	0
0	0

Lower Boundary

Template:

x	x
x	x
x	x
e	e

Number: 2

Tiles:

1	1
1	1
1	1
e	e

0	0
0	0
0	0
e	e

A.2.2.2 1st-level correction boundary tiles

First Type Upper Boundary

Template:

e	e
x	x

Number: 2

Tiles:

e	e
0	0

e	e
1	1

Second Type Upper Boundary

Template:

e	e
x	x
x	y
x	x
y	y

Number: 4

Tiles:

e	e
0	0
0	0
0	0
0	0

e	e
1	1
1	1
1	1
1	1

e	e
0	0
0	1
0	0
1	1

e	e
1	1
1	0
1	1
0	0

First Type Upper Boundary

Template:

x	w
y	w
y	w
z	w
z	w
e	e

if $(x = y = z)$ then no error and $w = x$
else if $(x = y \neq z)$ then there is error in z and $w \leftarrow x$
else if $(x = z \neq y)$ then there is error in y and $w \leftarrow x$
else if $(y = z \neq x)$ then there is error in x and $w \leftarrow y$

Number: 8

Tiles:

0	0
0	0
0	0
0	0
0	0
e	e

1	1
1	1
1	1
1	1
1	1
e	e

1	0
0	0
0	0
0	0
0	0
e	e

0	0
1	0
1	0
0	0
0	0
e	e

0	0
0	0
0	0
1	0
1	0
e	e

0	1
1	1
1	1
1	1
1	1
e	e

1	1
0	1
0	1
1	1
1	1
e	e

1	1
1	1
1	1
0	1
0	1
e	e

Second Type Lower Boundary

Template:

x	x
x	x
e	e

Number: 2

Tiles:

0	0
0	0
e	e

1	1
1	1
e	e

A.2.2.3 2nd-level correction boundary tiles

First Type Upper Boundary Tiles

Template:

e	e
y	c
x	c
x	c

if $(y = x)$ then there is no error and $c \leftarrow x$
else if $(x \neq y)$ then there is error in y and $c \leftarrow x$

Number: 4

Tiles:

e	e
0	0
0	0
0	0

e	e
1	1
1	1
1	1

e	e
0	1
1	1
1	1

e	e
1	0
0	0
0	0

Second Type Upper Boundary Tiles

Template:

e	e

Number: 1

Tiles:

e	e

First Type Lower Boundary Tiles

Template:

x	x
x	x
x	x
e	e

Number: 2

Tiles:

0	0
0	0
0	0
e	e

1	1
1	1
1	1
e	e

Second Type Lower Boundary Tiles

Template:

e	e

Number: 1

Tiles:

e	e

References

[1] Egli, M., Saenger, W.: Principles of Nucleic Acid Structure. Springer-Verlag (1984) xiii, 5, 6

[2] Rothemudn, P.W.K.: Folding DNA to create nanoscale shapes and patterns. Nature **440** (2006) 297–302 xiii, 9

[3] Fu, T.J., Seeman, N.C.: DNA double-crossover molecules. Biochemistry **32**(13) (1993) 3211–3220 xiii, 1, 2, 9, 10, 20

[4] Winfree, E.: Algorithmic self-assembly of DNA: Theoretical motivations and 2d assembly experiments. Journal of Biomolecular Structure and Dynamics **11**(2) (2000) 263–270 xiii, 10, 11, 12, 14

[5] Barish, R.D., Rothemund, P.W.K., Winfree, E.: Two computational primitives for algorithmic self-assembly: Copying and counting. Nano Letters **5**(12) (2005) 2586–2592 xiii, 11, 13

[6] Lagoudakis, M.G., LaBean, T.H.: 2D DNA self-assembly for satisfiability. In: Proceedings of the 5th DIMACS Workshop on DNA Based Computers. (1999) 141–154 xiii, 13, 14

[7] Winfree, E.: DNA computing by self-assembly. In: Frontiers of Engineering: Reports on Leading-Edge Engineering from the 2003 NAE Symposium on Frontiers of Engineering. (2003) 105–118 xiii, 15, 16

[8] Fujibayashi, K., Hariadi, R., Park, S.H., Winfree, E., Murata, S.: Toward reliable algorithmic self-assembly of DNA tiles: A fixed-width cellular automaton pattern. Nano Letters **8**(7) (2007) 1791–1797 xiii, 2, 19, 20, 21, 22, 23, 24

[9] Lloyd, S.: Quantum-mechanical computer. Scientific American **273**(4) (1995) 350–356 1

[10] Adleman, L.M.: Molecular computation of solutions to combinatorial problems. Science **266**(5187) (1994) 1021–1024 1, 6, 8, 16

[11] Watson, J.D., Crick, F.H.C.: A structure for deoxyribose nucleic acid. Nature **171**(4356) (1953) 737–738 5

[12] Alberts, B., Johnson, A., Lewis, J., Raff, M., Roberts, K., Walters, P.: Molecular Biology of the Cell. 4th edn. Garland Science (2002) 6

[13] Kanellos, M.: New life for Moore's law. CNET News.com (April 2005) http://news.cnet.com/New-life-for-Moores-Law/2009-1006_3-5672485.html. 7

[14] Whitesides, G.M., Grzybowski, B.: Self assembly at all-scales. Science **295**(5564) (2002) 2418–2421 8

[15] Grunbaum, B., Shephard, G.C.: Tilings and Patterns. W. H. Freeman, New York (1987) 8

[16] Seeman, N.C.: Nucleic acid junctions and lattices. Journal of Theoretical Biology **2**(99) (1982) 237 8

[17] Seeman, N.C.: Nanotechnology and the double helix. Scientific American **6**(290) (2005) 64–75 9

[18] Kari, L., Gloor, G., Yu, S.: Using DNA to solve the bounded post correspondence problem. Theoretical Computer Science (2000) 16

[19] Chen, H.L., Goel, A.: DNA Computing, Error Free Self-assembly Using Error Prone Tiles. Volume 10 of 3384. Springer Berlin Heidelberg (2005) 23

[20] Chen, H.L., Schulman, R., Goel, A., Winfree, E.: Reducing facet nucleation during algorithmic self-assembly. Nano Letters **7**(9) (2007) 2913–2919 23

[21] Yan, H., Park, S., Finkelstein, G., Reif, J., LaBean, T.: DNA-templated self-assembly of protein arrays and highly conductive nanowires. Science **301**(5641) (2003) 1882–1884 23

[22] He, Y., Chen, Y., Liu, H., Ribbe, A.E., Mao, C.J.: Design and construction of double-decker tile as a route to three-dimensional periodic assembly of DNA. Journal of the American Chemical Society **127**(35) (2005) 12202–12203 23