5-2023

Problems in Algorithmic Self-assembly and a Genetic Approach to Patterns

Andrew Rodriguez
The University of Texas Rio Grande Valley

PROBLEMS IN ALGORITHMIC SELF-ASSEMBLY AND A GENETIC APPROACH TO PATTERNS

A Thesis

by

ANDREW RODRIGUEZ

Submitted in Partial Fulfillment of the
Requirements for the Degree of MASTER
OF SCIENCE

Major Subject: Computer Science

The University of Texas Rio Grande
Valley May 2023

PROBLEMS IN ALGORITHMIC SELF-ASSEMBLY

AND A GENETIC APPROACH TO PATTERNS

A Thesis
by
ANDREW RODRIGUEZ

COMMITTEE MEMBERS

Dr. Timothy Wylie
Chair of Committee

Dr. Robert Schweller
Committee Member

Dr. Bin Fu
Committee Member

Dr. Yifeng Gao
Committee Member

May 2023

ABSTRACT

Rodriguez, Andrew, Problems in Algorithmic Self-Assembly and a Genetic Approach to Patterns. Master of Science (MS), May, 2023, 88 pp., 34 figures, references, 22 titles.

As it becomes increasingly harder to make transistors smaller, replacements for traditional silicon computers become sought after. To study the computing power of these potential computers, various theoretical models have been proposed, such as the abstract Tile Assembly Model (aTAM) and chemical reaction networks (CRNs). This thesis compiles research in various models such as the aTAM, Tile Automata, and CRNs. This work shows an investigation of covert computation in the aTAM and an evolutionary algorithm to approximate solutions to the pattern self-assembly tile set synthesis (PATS) problem. Next, optimal state complexity for building squares in Tile Automata is shown along with a Tile Automata simulation of the staged assembly model (SAM). Lastly, reachability for restricted general CRNs, reachability for feed-forward CRNs, and reachability for Void and Autogenesis CRNs are shown to be in various complexity classes.

DEDICATION

To Toby the Cat. Always sleeping quietly by my side while I think and read and write.

Here’s to many more naps, Toby.

ACKNOWLEDGMENTS

Thank you to my parents, Raul and Veronica, and to my sister, Audrey, for being my biggest supporters. I would not be where I am today without them.

Thank you to Dr. Robert Schweller and Dr. Tim Wylie. In two short years, they have changed my life forever.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

CHAPTER I

INTRODUCTION

As traditional silicon computers face more obstacles for higher levels of performance, research fields such as molecular programming show promise as aides or replacements. Molecular computing aims to combine computer science, chemistry and molecular biology to build nanoscale computers. DNA computing is by far the most popular research in this field, with the implementation of a DNA tile set that can be reprogrammed for a wide variety of 6-bit algorithms in (Woods et al. 2019). This DNA tile set is based on the abstract Tile Assembly Model (aTAM) the first tile self-assembly model introduced in Erik Winfree's dissertation (Winfree 1998). Furthermore, chemical reaction networks (CRNs) are another popular research area for molecular computing. CRNs are a theoretical model of computing where chemical species react based on the rules of the system.

In this work, I compile my research in a variety of these theoretical models and show the complexity of certain problems within those models. Each chapter focuses on a signal computational model and its various restrictions. In an attempt to be self-contained, I give definitions and examples of each model at the beginning of each chapter. In chapter II, I show our research which explored designing covert tile assembly computers in the aTAM, with a focus on tile assembly computers with a polynomial size description. Additionally, I show my research on using an evolutionary algorithm to approximate solutions to the pattern self-assembly tile set synthesis problem. Chapter III first highlights our research on building squares with optimal state complexity in Tile Automata, then shows how Tile Automata simulates another model, termed the staged assembly model (SAM). Lastly, in chapter IV, I show reachability results for three distinct restricted versions of chemical reaction networks.

Some sections contain research that was conducted with collaborators. The use of 'we'

highlights which sections contain research written with other coauthors. Some of the collaborators include Robert M. Alaniz, Timothy Gomez, Elise Grizzell, Robert Schweller and Tim Wylie.

CHAPTER II

ABSTRACT TILE ASSEMBLY MODEL

2.1 Model Definitions

At a high level, the Abstract Tile-Assembly Model (aTAM) uses a set of *tiles* capable of sticking together to construct shapes. These tiles are typically squares (2D) or cubes (3D) with *glues* on each side where they may attach to one another. A glue is labeled to indicate its type, governing what other tiles it may bond with and the *strength* of the bond. A tile with all of its labels is a *tile type*. A *tile set* contains all the tile types of the system. A single tile may attach at a location if the combined strength of the matching glues is greater than or equal to the *temperature* τ. An *assembly* is a shape made up of one or more combined tiles. Construction is started around a designated *seed* assembly S. Any assembly capable of being made from the seed is called a *producible* assembly. An assembly is *terminal* if no more tiles can attach. A terminal assembly is said to be *uniquely produced* if it is the only terminal assembly that can be made by a tile system. A tile system is formally represented as an ordered triplet $\Gamma = (T, s, \tau)$ of the tile set, seed assembly, and temperature parameter, respectively.

2.1.1 aTAM Formal Definitions

Definition (Tiles). Let Π be an alphabet of symbols called the *glue types*. A tile is a finite edge polygon with some finite subset of border points, each assigned a glue type from Π. Each glue type $g \in \Pi$ also has some integer strength $str(g)$. Here, we consider unit square tiles of the same orientation with at most one glue type per face, and the *location* to be the center of the tile located at integer coordinates.

Definition (Assemblies). An assembly A is a finite set of tiles whose interiors do not overlap.

If each tile in A is a translation of some tile in a set of tiles T, we say that A is an assembly over tile set T. For a given assembly A, define the *bond graph* G_A to be the weighted graph in which each element of A is a vertex, and the weight of an edge between two tiles is the strength of the overlapping matching glue points between the two tiles. Only overlapping glues of the same type contribute a non-zero weight, whereas overlapping, non-equal glues contribute zero weight to the bond graph. The property that only equal glue types interact with each other is referred to as the *diagonal glue function* property, and is perhaps more feasible than more general glue functions for experimental implementation (see (Cheng et al. 2005) for the theoretical impact of relaxing this constraint). An assembly A is said to be τ-*stable* for an integer τ if the min-cut of G_A has weight at least τ.

Definition (Tile Attachment). Given a tile t, an integer τ, and an assembly A, we say that t may attach to A at temperature τ to form A' if there exists a translation t' of t such that $A' = A \cup \{t'\}$, and the sum of newly bonded glues between t' and A meets or exceeds τ. For a tile set T, we use notation $A \rightarrow_{T,\tau} A'$ to denote there exists some $t \in T$ that may attach to A to form A' at temperature τ. When T and τ are implied, we simply say $A \rightarrow A'$. Further, we say that $A \rightarrow^* A'$ if either $A = A'$, or there exists a finite sequence of assemblies $\langle A_1 \ldots A_k \rangle$ such that $A \rightarrow A_1 \rightarrow \ldots \rightarrow A_k \rightarrow A'$.

Definition (Tile Systems). A tile system $\Gamma = (T, S, \tau)$ is an ordered triplet consisting of a set of tiles T called the system's *tile set*, a τ-stable assembly S called the system's *seed* assembly, and a positive integer τ referred to as the system's *temperature*. A tile system $\Gamma = (T, S, \tau)$ has an associated set of *producible* assemblies, $\texttt{PROD}_\Gamma$, which define what assemblies can grow from the initial seed S by any sequence of temperature τ tile attachments from T. Formally, $S \in \texttt{PROD}_\Gamma$ is a base case producible assembly. Further, for every $A \in \texttt{PROD}_\Gamma$, if $A \rightarrow_{T,\tau} A'$, then $A' \in \texttt{PROD}_\Gamma$. That is, assembly S is producible, and for every producible assembly A, if A can grow into A', then A' is also producible.

We further denote a producible assembly A to be *terminal* if A has no attachable tile from T at temperature τ. We say a system $\Gamma = (T, S, \tau)$ *uniquely produces* an assembly A if all producible assemblies can grow into A through some sequence of tile attachments. More formally, Γ *uniquely*

produces an assembly $A \in \mathtt{PROD}_\Gamma$ if for every $A' \in \mathtt{PROD}_\Gamma$ it is the case that $A' \rightarrow^* A$. Systems that uniquely produce one assembly are said to be *deterministic*.

2.1.2 Covert Definitions

Here, we provide formal definitions for computing a function with a tile system and the further requirements for the covert computation of a function. Our formulation of computing functions is that used in (Cantu et al. 2019), which is a modified version of the definition provided in (Keenan et al. 2016) to allow for each bit to be represented by a subassembly potentially larger than a single tile.

Definition (Tile Assembly Computers). Informally, a Tile Assembly Computer (TAC) for a function f consists of a set of tiles, along with a format for both input and output. The input format is a specification for how to build an input seed to the system that encodes the desired input bit-string for function f. We require that each bit of the input be mapped to one of two assemblies for the respective bit position: a sub-assembly representing "0" or a sub-assembly representing "1". The input seed for the entire string is the union of all these sub-assemblies. This seed, along with the tile set of the TAC, forms a tile system. The output of the computation is the final terminal assembly this system builds. To interpret what bit-string is represented by the output, a second *output* format specifies a pair of sub-assemblies for each bit. The bit-string represented by the union of these subassemblies within the constructed assembly is the output of the system.

For a TAC to *covertly* compute f, the TAC must compute f and produce a unique assembly for each possible output of f. We note that our formulation for providing input and interpreting output is quite rigid and may prohibit more exotic forms of computation. Further, we caution that any formulation must take care to prevent "cheating" that could allow the output of a function to be partially or completely encoded within the input. To prevent this, a type of *uniformity* constraint, akin to what is considered in circuit complexity (Vollmer 1999), should be enforced.

Definition (Input/Output Templates). An n-bit input/output template over tile set T is a

sequence of ordered pairs of assemblies over T: $A = (A_{0,0}, A_{0,1}), \ldots, (A_{n-1,0}, A_{n-1,1})$. For a given n-bit string $b = b_0, \ldots, b_{n-1}$ and n-bit input/output template A, the *representation* of b with respect to A is the assembly $A(b) = \bigcup_i A_{i,b_i}$. A template is valid for a temperature τ if this union never contains overlaps for any choice of b and is always τ-stable. An assembly $B \supseteq A(b)$, which contains $A(b)$ as a subassembly, is said to represent b as long as $A(d) \nsubseteq B$ for any $d \neq b$. We refer to the size of a template as the size of the largest assembly defined by the template.

Definition (Function Computing Problem). A *tile assembly computer* (TAC) is an ordered quadruple $\Im = (T, I, O, \tau)$ where T is a tile set, I is an n-bit input template, and O is a k-bit output template. A TAC is said to compute function $f : \mathbb{Z}_2^n \to \mathbb{Z}_2^k$ if for any $b \in \mathbb{Z}_2^n$ and $c \in \mathbb{Z}_2^k$ such that $f(b) = c$, then the tile system $\Gamma_{\Im,b} = (T, I(b), \tau)$ uniquely assembles a set of assemblies which all represent c with respect to template O.

Definition (Covert Computation). A TAC *covertly* computes a function $f(b) = c$ if 1) it computes f, and 2) for each c, there exists a unique assembly A_c such that for all b, where $f(b) = c$, the system $\Gamma_{\Im,b} = (T, I(b), \tau)$ uniquely produces A_c. In other words, A_c is determined by c, and every b where $f(b) = c$ has the exact same final assembly.

Definition (Polynomial-Sized Tile Assembly Computers). We say a TAC is polynomial size if the input template, tile set, and output template are all polynomial in n. However, this requirement still allows the producible assemblies to be exponentially larger. We say a TAC is *strictly* polynomial size if the produced assemblies are also polynomial in size.

2.1.3 Pattern Self-Assembly Tile Set Synthesis Definitions

Here, I give definitions and examples of the pattern self-assembly tile set synthesis (PATS) problem.

Definition (Pattern Self-Assembly Tile Set Synthesis). Given a 2D two-colored pattern P, find a tile set that builds P.

Definition (Minimum Pattern Self-Assembly Tile Set Synthesis). Given a 2D two-colored pattern P, find the *smallest* tile set that builds P. In (Kari et al. 2016), this problem was proven to be NP-hard.

2.2 Covert Computation

In our manuscript, we investigate the problem of covert computation in the aTAM. We start by showing how to construct a polynomial size tile set that simulates a covert circuit with only positive glues and one step in the third dimension. Next, we characterize the functions computable by our constructions and those of previous work (Cantu et al. 2019) into a complexity class named P/Poly.

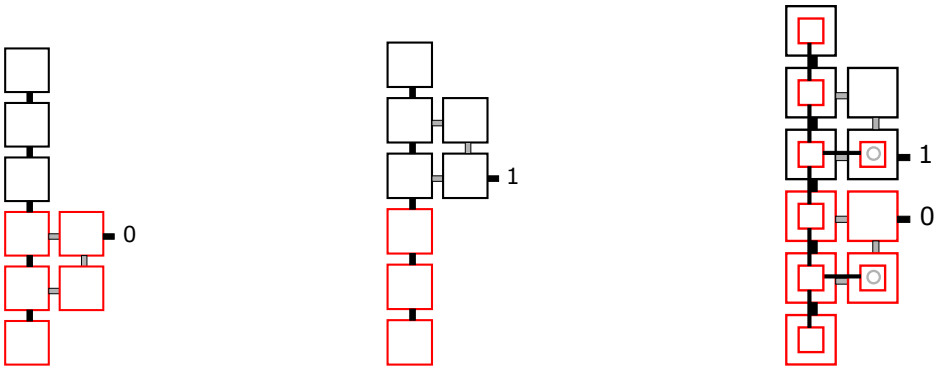
2.2.1 3-Dimensional Covert Circuits

In this section, we show how to perform covert computation in the aTAM using 3 dimensions. The computation behaves similarly to the covert circuit construction in (Cantu et al. 2019) by building NAND gates and FANOUTs using dual rail logic. We start with showing a NOT that switches which wire is "on", then extending to a NAND by utilizing cooperative binding.

The main difference between the two constructions is when *backfilling* occurs, which is the process of filling in the unused dual rail line once that line is no longer needed. Here, we do not backfill as we go, rather, we fill in the assembly once the computation is complete.

2.2.2 Input Assemblies

Our input assembly consists of n 1×6 columns with two of four tiles attached on the right (Figures 2.1a and 2.1b). The top two tiles will be included when the input is 1, and the bottom two tiles if the input is 0. These tiles have enough attachment strength to be stable when both are present, however, since the tiles only have strength 1 bonds, they may not attach alone. This initially prevents the growth of the other bit, which is not placed until the computation is complete, further elaboration of this process is described in section 2.2.6.



(a) Bit Assembly Template 0 (b) Bit Assembly Template 1 (c) Bit Assembly Output Template

Figure 2.1: Input assemblies and their respective input templates. The dark squares represent the bit set to one, and the red squares represent the bit set to zero. Grey glues are strength-1, black and red glues are strength 2. Circles in the center of tiles represent 3D glues.

2.2.3 Wires and NOT Gates

Bit information is represented and transferred using a wire. A wire is constructed using two rows of tiles (Figure 2.2a), each representing a binary value of 0 or 1. This dual rail system initially grows only one of the rows from the input assembly based off the input and then builds into the gates. Before the circuit finishes growing, only one row of each wire will be constructed, and at the end, the other wire row will be built.

Gates such as the NOT grow off the wires. An example of a NOT gate can be seen in Figure 2.2b, notice how we utilize the third dimension to cross the wires over each other. This gate swaps the position of the rows of tiles; a row that represents a 0 will now be in the upper row and represent a 1. At the end of each gate is a diode gadget that was used in previous work (Cantu et al. 2019). The gadget is a 2×2 subassembly that grows only in one direction. If the first tile is placed, the whole thing will be first. If the last tile is placed, nothing else grows since it connects using two strength 1 glues. This prevents errors caused by "backward" growth.

2.2.4 NAND Gates

We construct a NAND gate using the NOT gate and cooperative binding. The full NAND gate can be seen in Figure 2.3. If either input to a NAND gate is 0, the output is always 1. This can be seen in Figures 2.4a, 2.4b, and 2.4c. If any red tile is placed, the 1 output of the gate will be built.

(a) Dual Rail Logic (b) NOT Gate

Figure 2.2: (a) We use dual rail gates. The input glue of 1 grows the black tiles and 0 grows the red. (b) A NOT gate is implemented by crossing the wires over each other.

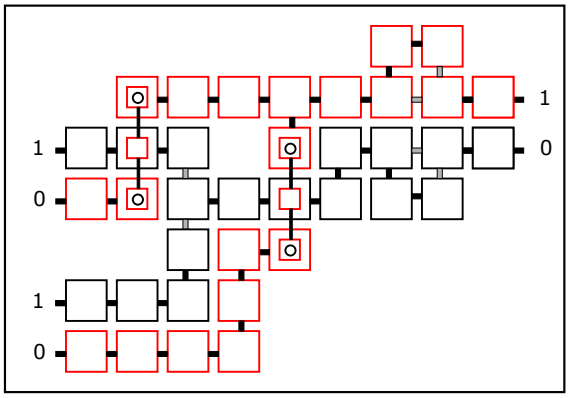


Figure 2.3: Full NAND Gate construction in the full circuit. The tiles in black represent tiles that will be built from an input of 1 input, while the red tiles come from an input of 0.

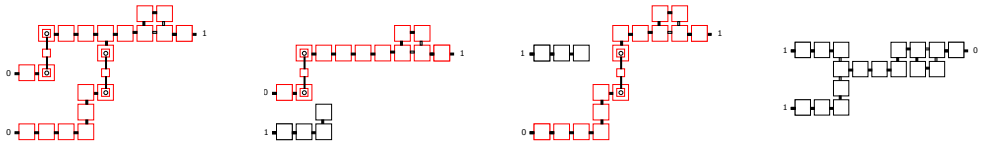
If both inputs are 1, the 0 output can be constructed using cooperative binding.

One thing to note in the case of one output being 0 and the other being 1 is that the red tiles will be placed along the other wire. However, this will not cause any issues since it can only build back up to the output of the previous gate due to the diode gadget.

2.2.5 Fan Out and Crossover

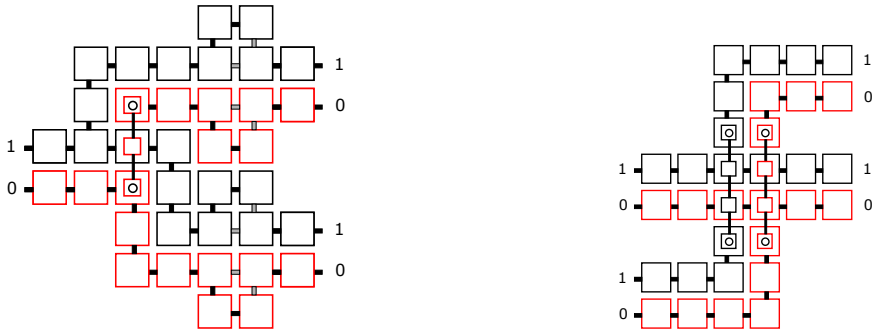
Two other gadgets that assist in creating circuits are the Fan Out and Crossover gadgets. The Fan Out (Figure 2.5a) splits a wire in order to copy the value to two gates. It does this by having each tile path split, and then use the third dimension to swap the positions.

The crossover gadget (Figure 2.5b) allows for the creation of non-planar circuits. Using the third dimension, a wire can go over another wire in order to reach its input. While such 3D crossovers simplify constructions greatly, we note that such crossovers are not necessarily needed, as planar circuits can simulate such crossovers using XOR gates (Cantu et al. 2019).



(a) Input 00 (b) Input 01 (c) Input 10 (d) Input 11

Figure 2.4: Growth of possible inputs to a NAND gate. The gate will stay like this after computing, before the history is hidden.



(a) Fan Out Gadget (b) Crossover gadget

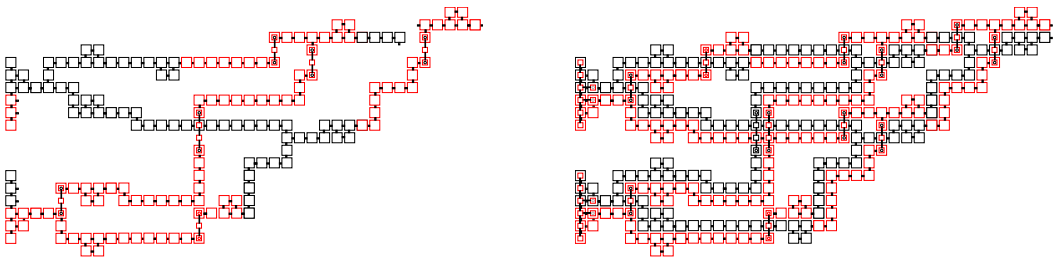
Figure 2.5: (a) A FANOUT gadget (b) While a crossover is not required for universal computation, we can easily implement one by using the 3rd dimension.

2.2.6 Backfilling and Target Assemblies

In order to perform covert computation, there must exist a unique assembly for each output. The gray tile at the end of the circuit in Figure 2.6a is one of two flag tiles that denotes the output of the circuit. Once this tile is placed, a row of tiles is built back towards the input (Figure 2.6b). Once the input assembly is reached, the tiles above the input are placed, thus allowing for the input assemblies to be filled in. This causes the entire circuit to be filled out, which hides the original input and computation history.

Theorem. For any n-bit function f that is computable by a Boolean circuit, there exists a Tile Assembly Computer $\Im$ which covertly computes f in the 3D aTAM with only positive glues. Further, $\Im$ is strictly polynomial in n.

Proof. We can construct the tile set T_c from the circuit c that computes f. Arrange the gates and wires on the square grid using $O(n^2)$ space, and scale up each gate and wire by a constant factor. Wires are scaled up by a factor of 2 to account for the dual rail logic wires. The gates are scaled up



(a) Computed Circuit (b) Output Assembly

Figure 2.6: Example structures of the computation circuit of an XOR using NOTs and NANDs. The circuit before backfilling is on the left, and the final output is shown on the right side. (a) A circuit once the output is computed. (b) Once the output grows backward, the other input bits are placed.

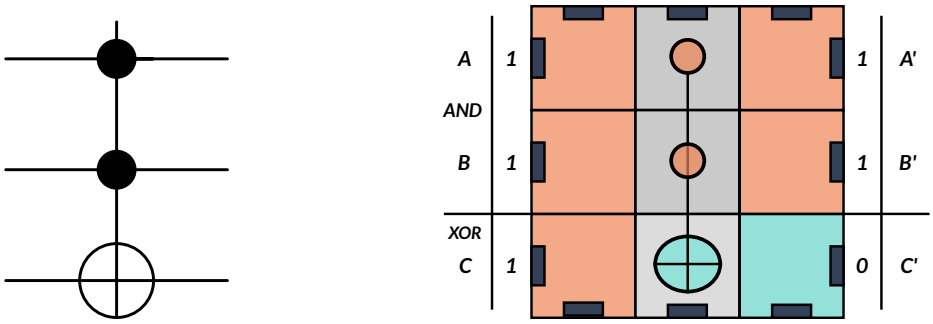
by a factor depending on which gate it is, however, all the gates we present are only a constant size. This creates assembly $A_{c,Full}$.

We now show that $\Im$ computes f. Consider an n-bit input x to f, using the input template create seed assembly A_x. Each gate will grow from A_x, computing the circuit on each input. Since backfilling does not occur until the circuit finishes computing, we guarantee only the correct outputs grow from the final gate. The circuit is computed covertly since the output then grows back to the start of the circuit and places the unused inputs. □

2.2.7 Exponential Assembly Covert Computer in 2D

In this section, we show that covert computation is possible in 2D in the standard aTAM, where the input can be described in polynomial size, yet the final terminal assembly is exponential in size. Thus, while we are able to achieve strictly polynomial-sized covert computation in 3D, we achieve (non-strict) polynomial-sized covert computation in 2D.

This construction is possible by first computing the function using reversible Toffoli gates, and then replicating and computing the circuit for all possible inputs. Once the output of the original input is placed, the Toffoli gate reverses its computation to build a mirror of the circuit with the input replicated on both the right and left. The output builds an assembly arm used to place tiles on either side of the assembly to increment and decrement the mirrored inputs based on the binary value of the original input, thus seeding a new input for exponential growth in each direction. Thus, for a 4-bit input, it builds the circuit for all 2^4 possible inputs after it builds the output template.



(a) (b)

Figure 2.7: (a) Logical representation of Toffoli gate. (b) A Toffoli gate on a grid can be represented by the three vertical 'cells' of elementary logic gates.

2.2.8 Toffoli Gate

The Toffoli gate is a 3-bit reversible universal logic gate (Figure 2.7a), we denote the inputs A, B, C, and the outputs A', B', C'. The first two input and output bits map to each other: $A = A'$ and $B = B'$. The third output flips the C bit if both A and B are 1. Logically expressed, this is $C \otimes (A \wedge B) = C'$.

We can express an n-bit d-depth reversible circuit as a $n \times d$ grid where each row represents a wire, and each column is a layer of gates and wires. Each gate can be represented by tiles computing the elementary 2-bit AND and XOR and implementing a fan out, as shown in Figure 2.7b.

2.2.9 Covert Circuit

The input template is a specific tile for each bit. Given an n-bit string, we create a $n \times 1$ bit assembly with stability-granting left and bottom circuit construction scaffolds, as shown in Figure 2.9c.

The circuit assembly is a $n \times (d+2)$ rectangle. Each Toffoli gate is a 3×1 subassembly. Three possible computations of a single Toffoli gate are shown in Figure 2.8. Typically, these gates must be reversible, meaning the circuit may grow from the east or west but produce the same assembly We note that the gate itself is not covert, and the "covertness" comes from the full construction.

An example Toffoli circuit is shown in Figure 2.9a along with the logical representation in

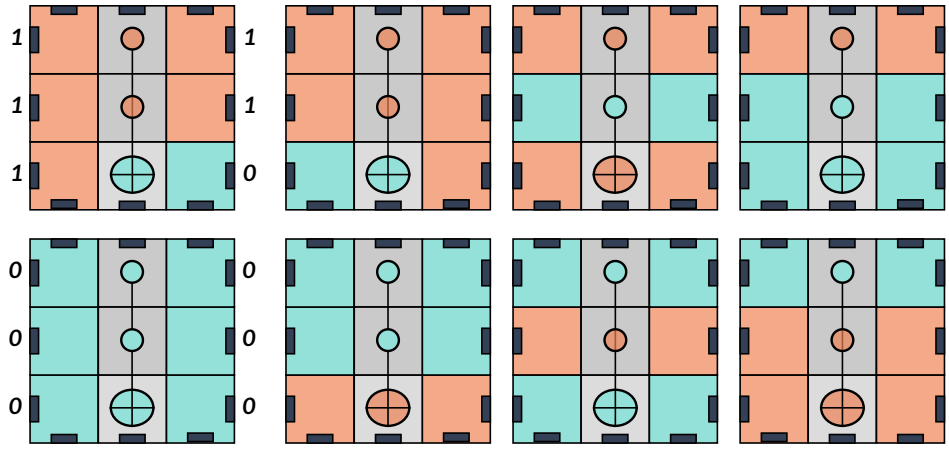


Figure 2.8: All possible computations of a single Toffoli gate. 1 (orange), 0 (blue). $111 \rightarrow 110, 110 \rightarrow 111, 101 \rightarrow 101, 100 \rightarrow 100$, Row 2: $000 \rightarrow 000, 001 \rightarrow 001, 010 \rightarrow 010$, and $011 \rightarrow 011$.

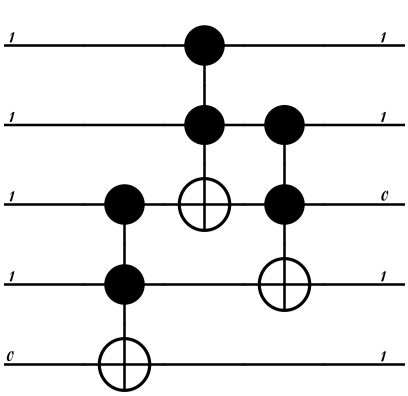
Figure 2.9b. A constructed circuit assembly in one direction can be seen in Figure 2.9d.

2.2.10 Increment/Decrement Input to Next Circuit Logic

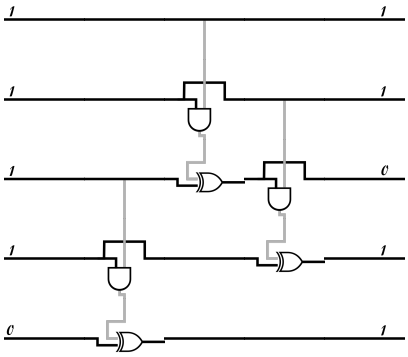
After completion of a circuit, three columns of tiles are built: mark for increment (left), copy or flip (center), and mark for decrement (right). The order of growth of these columns depends on the starting direction. Growing from the left to increment input to the next circuit or from the right to decrement it. Cooperatively with those columns, below the output arm begins its extension to transmit the outcome, accept or reject, of the original circuit. This arm extension continues to the center circuit output outcome tile location. From here, the circuit construction scaffold, previously provided in the input template, may loop back to the edge of the circuit so the new input scaffold and bits may place as illustrated in Figure 2.11. The circuit growth continues normally from that point forward, with the exception of the output tile placement.

2.2.11 Output Assembly

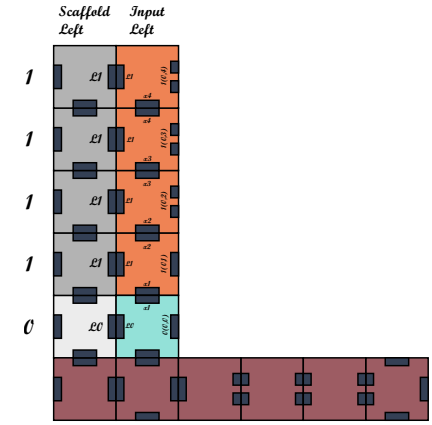
Once the output is built, the rows below have d tiles attached in the east and west directions that encode the output. Through cooperative attachment, tiles are placed to allow the strings to increment/decrement, as described above. The final terminal assembly contains every possible computation.



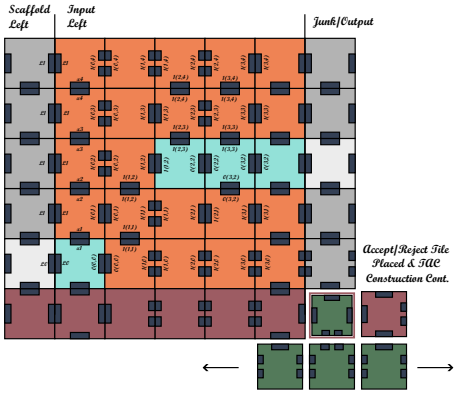
(a) Toffoli Circuit



(b) AND/XOR Diagram



(c) Toffoli Input Assembly



(d) Toffoli Circuit Assembly

Figure 2.9: (a) Example 5-bit Toffoli Circuit. (b) The Toffoli circuit represented with AND and XOR gates. (c) Example Input Assembly. For each bit (1 or 0), we place the scaffold (grey or white) and input bit tile (orange or blue). The bottom is a row of circuit construction scaffold tiles (maroon). (d) The Toffoli Circuit Assembly built in one direction. The (green) tile below the output/junk column represents the (positive) output and will allow the output control row to place.

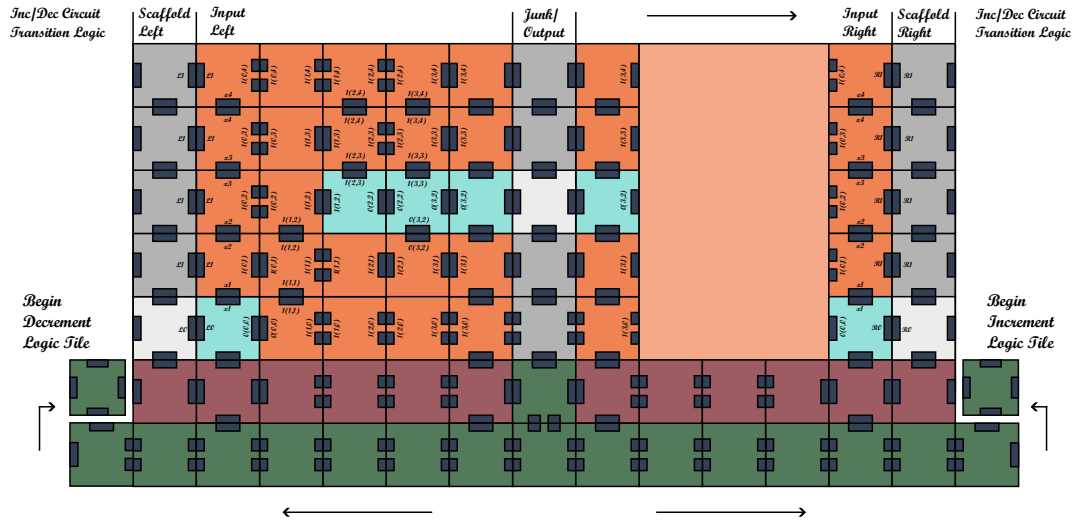


Figure 2.10: An example of a symmetrical circuit that has built both sides and is placing begin decrement and increment logic tiles.

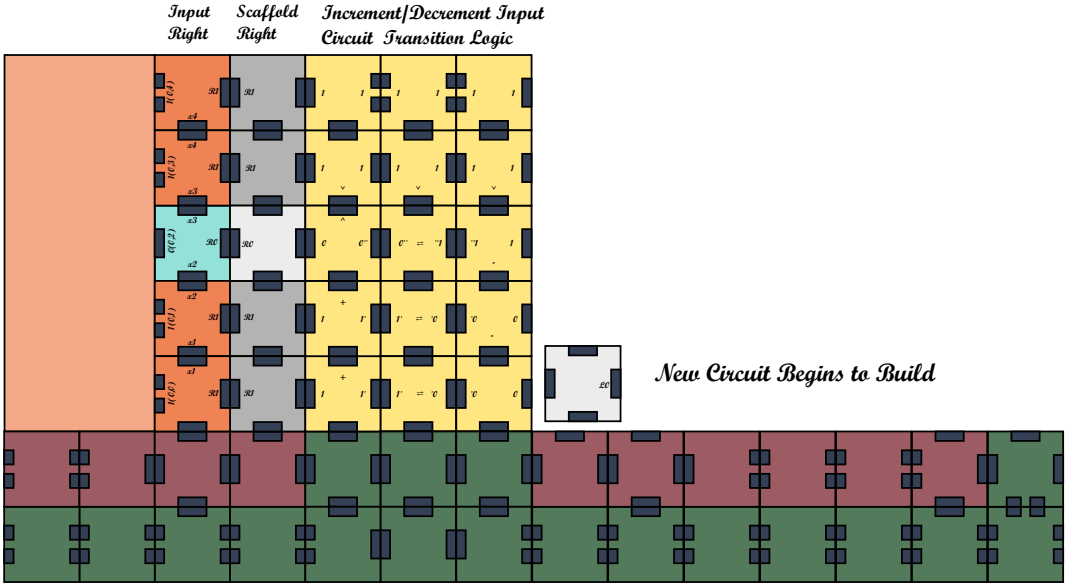


Figure 2.11: An example of a new circuit created by incrementing the output from a previously built assembly.

Theorem. For all functions $f(x)$ that are computable by a n-bit reversible circuit R, there exists a polynomial tile assembly computer $\Im = (T, I, O, 2)$ that covertly computes $f(x)$ and has an output assembly of size $O(2^n)$.

Proof. If there exists a n-bit reversible circuit R that computes $f(x)$, we construct tile assembly computer $\Im = (T, I, O, 2)$ as follows. From the circuit R that computes f, we design a circuit R' to compute f with Toffoli gates as described in section 2.2.9. Using R' and the developed input increment/decrement logic for circuit replication, we construct a tile set T_c.

We create the input assembly I by converting the n-bit input string x to tiles L_i in scaffold left (figure 2.9c) and associated input, and a bottom row of tiles called the left circuit construction scaffold.

From here, the left assembly will grow into figure 2.9d, once the output is determined to be 'accept' or 'reject', the output indicator tile is placed, and the original output indicator arms grow to allow the Right Assembly the ability to grow as well as place begin decrement and increment logic tiles on the bottom left and right sides of the completed assembly respectively, as seen in figure 2.10.

All n-bit computations of $f(y)$ for y less than original input x will be computed to the left of the original assembly, and all $x_n > x$ after being decremented and incremented using the reversible and symmetric logic in yellow from figure 2.11. Growth is halted by the INC/DEC logic at overflow in either direction.

The ability to grow further left/right circuit construction scaffolds is dependent on the output arms from the original output indicator arms growing to the center of the circuit about to begin construction where the output accept/reject indicator tile would place, preserving the output status for every circuit built in the TAC.

As there are only two possible assemblies that can be built, accept all or reject all, the Tile Assembly Computer is polynomial size in description and exponential in output size. □

We have shown that if the output assembly is allowed to be exponential in size, that covert computation is possible in the aTAM, even in two dimensions. However, in practice, this is not

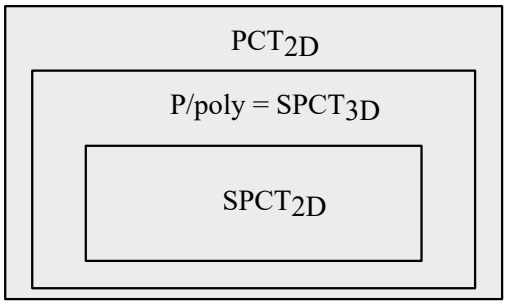


Figure 2.12: Diagram showing important classes defined in this section and their relation to P/poly. Note that none of these containments are known to be proper.

usually a plausible solution. Given that Unique Assembly Verification is in P (L. M. Adleman et al. 2002), it is unlikely that covert computation is possible with a strictly polynomial-size TAC.

Conjecture. There does not exist a strictly polynomial-size Tile Assembly Computer in the 2D Abstract Tile-Assembly Model.

2.2.12 Polynomial-Sized Covert Circuits

In this section, we define and investigate complexity classes based on decision problems computable by polynomial-sized covert computers. We start by introducing the class P/poly and defining three classes of covertly computable problems: the class of problems covertly computable by a strictly polynomial 3D system ($SPCT_{3D}$), the class of problems computable by a strictly polynomial 2D system ($SPCT_{2D}$), and the class of problems computable by a (non-strict) polynomial 2D system (PCT_{2D}). We show how these classes relate to each other, including the result that P/poly is equal to $SPCT_{3D}$. Our results in this section are summarized in Figure 2.12.

2.2.13 Complexity Classes

The class P/poly is a well-studied complexity class defined as the class of problems solvable by a polynomial-sized circuit. One note about this class is it puts no requirement on the circuit other than that it exists. This has an equivalent definition as the problems solvable by a polynomial-time Turing machine with a polynomial advice string. We can think of this as the Turing machine being given a description of the circuit and evaluating it. Here, the advice string or circuit must be identical for all inputs of length n.

Definition (P/poly). The class of problems solvable by a polynomial-sized Boolean circuit. Alternatively, defined as the problems solvable by a polynomial-time Turing machine $M < x, a_{|x|} >$, where x is the input and $a_{|x|}$ is an advice string that is based only on the length of x. That is, if two inputs x, y have the same size $|x| = |y|$, then they must use the same advice string.

We define the following three complexity classes to categorize the functions that are computable by polynomial-size covert TACs.

Definition (SPCT_{3D}). The class of problems solvable by a strict polynomial sized covert tile assembly computer in the 3D Abstract Tile-Assembly Model.

Formally, a language L is in SPCT_{3D} if there exists a sequence of covert TACs $C = \{C_1, C_2, ...\}$ such that the i^{th} TAC, C_i, is strictly polynomial in i and if it correctly computes all $x \in L$ where $|x| = i$.

Definition (SPCT_{2D}). The class of problems solvable by a strict polynomial sized covert tile assembly computer in the 2D Abstract Tile-Assembly Model.

Definition (PCT_{2D}). The class of problems solvable by a polynomial sized covert tile assembly computer in the 2D Abstract Tile-Assembly Model.

2.2.14 Strict Polynomial Size Equivalence

To show equivalence between P/poly and SPCT_{3D}, we first define the 2-Promise Unique Assembly Verification problem, a modified version of Unique Assembly Verification where we are given two assemblies, a and b, rather than a single target. The problem asks to separate two cases: accept if an assembly containing a as a subassembly is produced, and reject if an assembly containing b is produced. We assume it is promised that one of these cases is true. This problem is solvable in polynomial time since you only need to attach tiles until one of the two assemblies is produced (Lemma 2.2.1).

Definition (2-Promise Unique Assembly Verification problem). **Input:** Assemblies a, b and an aTAM system (T, s, τ) which is promised to uniquely produce one of two assemblies, A or B,

such that $a \subseteq A$ and $b \subset B$. **Output:** 'Yes', if Γ uniquely assembles A, and 'No', if Γ uniquely assembles B.

Lemma. The 2-Promise Unique Assembly Verification problem is solvable in polynomial time in the 3D aTAM.

Proof. Call greedy grow (from (L. M. Adleman et al. 2002)) to get maximal producible assembly C. If Γ uniquely assembles C and $a \subseteq C$, return 'yes'. Otherwise, return 'no'. □

Equipped with the algorithm for the 2-promise problem, and taking the description of a covert computer as an advice string, it follows that we can compute the seed assembly from the input template, and the two possible output assemblies from the output template, and then run the algorithm for the 2-Promise UAV problem (Lemma 2.2.2). This puts any problem solvable by a polynomial-sized covert circuit in the class P/poly. The other direction of equivalence is given by the 3D covert computer constructions.

Lemma. If a language L is computable by a strict polynomial-sized covert tile assembly computer in the 3D aTAM, then L is in P/poly.

Proof. Let $\Im_n(T,I,O,\tau)$ be the covert computer for the strings in language L of size n. Since $\Im_n$ is of strict polynomial size, we can encode the tile set, input/output templates, and temperature in $poly(n)$ bits. Thus, $\Im_n$ will be our advice string for membership in P/poly. Further, we are only considering decision problems. Thus, there are only two output templates which we denote as a_a and b_r for accept and reject, respectively.

Consider a Turing machine given the string x and covert circuit $\Im_{|x|} = (T,I,(a_a,b_r),\tau)$ that does the following:

- Convert x to an assembly $I(x)$ using the input template.
- Call the algorithm for 2-Promise UAV on input $((T,I(x),\tau),a_a,b_r)$.
- If the algorithm accepts then $x \in L$, else $x \notin L$

This Turing machine essentially runs the covert computer on x and then checks the output by seeing which template is included in the final assembly. □

Theorem. The classes $\text{SPCT}_{3\text{D}}$ and P/poly are equivalent.

Proof. By Lemma 2.2.2, if a language is in P/poly there is a Boolean circuit of polynomial size which computes it, giving us P/poly $\subseteq \text{PCT}_{3D}$. In Theorem 2.2.1 we show that if there exists a Boolean circuit, there exists a strictly polynomial sized covert computer that computes the circuit. □

2.2.15 Polynomial Sized 2D Covert Circuits

Here, we use previous constructions to show that the class of polynomial sized 2D covert circuits is at least as strong as strict polynomial covert circuits. That is every language in $SPCT_{3D}$ is in PCT_{2D}.

Theorem. If a language L is in P/poly then L is in PCT_{2D}

Proof. In Lemma 2.2.2 we show that if a language is in P/poly there is a Boolean circuit of polynomial size which computes it. Any Boolean circuit can be turned into a reversible circuit, thus by Theorem 2.2.2, if there exists a reversible circuit, there exists a polynomial tile assembly computer that computes it in 2D. □

2.3 Pattern Self-Assembly Tile Set Synthesis

Here, I show my findings concerning the PATS problem. I use an evolutionary algorithm to try and approximate minimal PATS solutions. I give an overview of evolutionary algorithms in general, along with a detailed description of the algorithm I implemented with various fitness functions.

2.3.1 Evolutionary Algorithms

Evolutionary algorithms are inspired by nature and solve problems using strategies which emulate behaviors of living organisms. Genetic algorithms (GAs) are a class of evolutionary

algorithms, and is the basis of my algorithm. GAs begin with an initial population of organisms. The first population is called generation 0. Each organism represents a potential solution to the problem that is being solved. In each iteration, or generation, of the algorithm, the population is evaluated using some fitness function. A fitness function will take the organism and assign some score based on how well the solution gathered from the organism solves the problem. Each fitness function is defined by some criteria that categorizes a good and a bad solution. After assigning a score to every organism in the population, the organisms with the highest scores are used to build the next generation's population. The last phase consists of mutating the population to promote diversity in the solutions, before repeating the process for the next generation.

Since the minimum pattern self-assembly tile set synthesis (PATS) problem is hard, even for 2 colors, no polynomial time algorithm exists for solving this problem. I was curious if a genetic algorithm could *approximate* solutions to this problem by finding tile sets with a 'good enough' number of tiles. I discuss the preliminary results at the end of this section, which includes the performance of different fitness functions and the size of the tile set over a number of generations. First, I'll talk about the algorithm in more detail.

2.3.2 Encoding PATS Solutions

An 'organism' of the algorithm is represented as a tuple containing a seed assembly and a glue table. The seed assembly will be an L-shaped assembly which tiles will build on to assembly the goal pattern. The glue table GT has entries $GT(s,w)$ where s is a south glue and w is a west glue. Each entry in the table is a tuple $GT(s,w) = (n,e)$ such that a tile t can be created using the input and output of the table entry $t = (n,e,s,w)$. Using a glue table to generate the tiles of the tile set guarantee that we can always completely build the assembly given that the table contains all the glues of the system.

Mutating these organisms is simple. Based on some mutation rate μ, the glues of the seed assembly and the entries in the glue table can change randomly to any other glue.

2.3.3 Algorithm Specifics

The algorithm is implemented in Python. For each generation, the entire population is evaluated using some fitness function. Before evaluation, an aTAM simulator is run to generate the final assembly built by each organism of the population. The top 10% of the population are used to create the next generation. The mutation rate for mutating the seed assembly and glue table specifies the rate at which each glue in the seed assembly or each entry of the table is mutated. When a glue or glue pair is mutates, a new glue or glue pair is randomly generated from the set of glues in the system.

The mutation rate of the seed assembly is $\mu_S = 0.30$ and the mutation rate of the glue table is $\mu_{GT} = 0.25$. The population size is 200 and the number of generations is $2,000$. Six different fitness functions were created, which will be explained in detail in the next section. For testing, 4 patterns were used: a 4×4 checkerboard pattern, a 4×4 lined pattern, a randomly generated 4×4 pattern, and a randomly generated 5×5 pattern.

2.3.4 Fitness Functions

Definition (Pattern Match First). The fitness function named pattern match first (PMF) looks at the entire pattern. Points are deducted for every tile with the incorrect color. Points are deducted for every unique tile used to build the pattern. This creates an unattainable goal as you *have* to use some number of tiles, but this is intended. Initially, each tile in the assembly does not have a color assigned. When a tile type is first encountered without a color, one is assigned based on the color it should be. Once a color is assigned, points are deducted if the color is ever incorrect for a given 'pixel' of the pattern.

Definition (Pattern Match Best). The fitness function named pattern match best (PMB) is similar to PMF. The difference is with the color assignment. Each tile type is assigned the color that gives the most correctly colored tiles.

Definition (Pattern Match First Tile Limit). The fitness function named pattern match first tile limit (PMFTL) is PMF with an added constraint. In each generation, the smallest, fully correct,

tile set is used to update the tile limit. For example, if in the current generation, there is a tile set that correctly builds the goal pattern using 8 tiles, then 8 becomes the tile limit. The tile limit is only updated if the value would decrease. Extra points are deducted for every unique tile type used that is over the current tile limit, with the hope of guiding the population to find smaller and smaller tile sets.

Definition (Pattern Match Best Tile Limit). The fitness function named pattern match best tile limit (PMBTL) is PMB, but with the tile limit constraint.

Definition (Line Match First). The fitness function named line match first (LMF) uses very similar techniques to PMF and PMB. Instead of deducting points per tile, LMF deducts points per *line* of tiles, including horizontal and vertical. If a single tile in the current line is wrong, points are deducted. Color is assigned in the same way as PMF.

Definition (Line Match Best). The fitness function named line match best (LMB) is LMF with the color assignment procedure as PMB.

2.3.5 Preliminary Results

Towards the end of this project, I realized a lot could be improved on and reworked within the design and implementation of the algorithm. Because of this, the results of this project are in the form of comments and observations. Here, I will discuss my observations of the algorithm design and how well it performed in an informal sense.

Each of the fitness functions performed about the same when comparing how quickly the initial population improved as the generations went on. For certain patterns, such as the checkerboard and the lined pattern, the line match fitness functions performed better than the pattern match fitness functions. For the random patterns, the line match functions performed worse than the others.

In terms of the overall performance of the algorithm, things could be improved. Although genetic algorithms have the potential to be very quick in finding solutions, the implementation is everything. Because the PATS problem deals with tile sets and aTAM assemblies, running a simulator to build the assemblies for every call of a fitness function is slow, especially when larger

patterns are used. Another potential slowdown is the fitness functions themselves. If the fitness function is implemented or designed poorly, the solutions generated by the algorithm will be poor as well.

There are various improvements possible to the algorithm. One being adding a dynamic mutation rate, which starts high in earlier generations to promote diversity, then decreases to refine solutions. Hardware acceleration is another potential improvement. The biggest improvement would be to explore other fitness functions and see what works better for certain patterns. These improvements and more are potential areas for future work.

CHAPTER III

TILE AUTOMATA

3.1 Model Definitions

The Tile Automata model differs quite a bit from typical self-assembly models since a *tile* may change *state*, which draws inspiration from Cellular Automata. Thus, there are two aspects of a TA system: the self-assembling that may occur with tiles in a state and the changes to the states once they have attached to each other. To address these aspects, we define the building blocks and interactions, and then the definitions around the model and what it may assemble or output. Finally, since we are looking at a limited TA system, we also define specific limitations and variations of the model.

3.1.1 Building Blocks

The basic definitions of all self-assembly models include the concepts of tiles, some method of attachment, and the concept of aggregation into larger assemblies. The Cellular Automata aspect also brings in the concept of transitions.

Definition (Tiles). Let Σ be a set of *states* or symbols. A tile $t = (\sigma, p)$ is a non-rotatable unit square placed at point $p \in \mathbb{Z}^2$ and has a state of $\sigma \in \Sigma$.

Definition (Affinity Function). An *affinity function* Π over a set of states Σ takes an ordered pair of states $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ and an orientation $d \in D$, where $D = \{\perp, \vdash\}$, and outputs an element of $\mathbb{Z}^{0+}$. The orientation d is the relative position to each other with $\vdash$ meaning horizontal and $\perp$ meaning vertical, with the σ_1 being the west or north state respectively. We refer to the output as the *Affinity Strength* between these two states.

Definition (Transition Rules). A *Transition Rule* consists of two ordered pairs of states $(\sigma_1, \sigma_2), (\sigma_3, \sigma_4)$ and an orientation $d \in D$, where $D = \{\perp, \vdash\}$. This denotes that if the states (σ_1, σ_2) are next to each other in orientation d (σ_1 as the west/north state) they may be replaced by the states (σ_3, σ_4).

Definition (Assembly). An assembly A is a set of tiles with states in Σ such that for every pair of tiles $t_1 = (\sigma_1, p_1), t_2 = (\sigma_2, p_2)$, $p_1 \neq p_2$. Informally, each position contains at most one tile. Further, we say assemblies are equal in regard to translation. Two assemblies A_1 and A_2 are equal if there exist a vector $\vec{v}$ such that $A_1 = A_2 + \vec{v}$.

Let $B_G(A)$ be the bond graph formed by taking a node for each tile in A and adding an edge between neighboring tiles $t_1 = (\sigma_1, p_1)$ and $t_2 = (\sigma_2, p_2)$ with a weight equal to $\Pi(\sigma_1, \sigma_2)$. We say an assembly A is τ-stable for some $\tau \in \mathbb{Z}^0$ if the minimum cut through $B_G(A)$ is greater than or equal to τ.

3.1.2 The Tile Automata Model

Here, we define and investigate the *Seeded Tile Automata* model, which differs by only allowing single tile attachments to a growing seed, similar to the aTAM.

Definition (Seeded Tile Automata). A Seeded Tile Automata system is a 6-tuple $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, s, \tau)$ where Σ is a set of states, $\Lambda \subseteq \Sigma$ a set of *initial states*, Π is an *affinity function*, Δ is a set of *transition rules*, s is a stable assembly called the *seed* assembly, and τ is the *temperature* (or threshold). Our results use the most restrictive version of this model where s is a single tile.

Definition (Attachment Step). A tile $t = (\sigma, p)$ may attach to an assembly A at temperature τ to build an assembly $A' = A \bigcup t$ if A' is τ-stable and $\sigma \in \Lambda$. We denote this as $A \rightarrow_{\Lambda,\tau} A'$.

Definition (Transition Step). An assembly A is transitionable to an assembly A' if there exist two neighboring tiles $t_1 = (\sigma_1, p_1), t_2 = (\sigma_2, p_2) \in A$ (where t_1 is the west or north tile) such that there exist a transition rule in Δ with the first pair being (σ_1, σ_2), the second pair being (σ_3, σ_4), and $A' = (A \setminus \{t_1, t_2\}) \bigcup \{t_3 = (\sigma_3, p_1), t_4 = (\sigma_4, p_2)\}$. We denote this as $A \rightarrow_{\Delta} A'$.

Definition (Producibles). We refer to both attachment steps and transition steps as production steps, we define $A \to_* A'$ as the transitive closure of $A \to_{\Lambda,\tau} A'$ and $A \to_\Delta A'$. The set of *producible assemblies* for a Tile Automata system $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, s, \tau)$ is written as $PROD(\Gamma)$. We define $PROD(\Gamma)$ recursively as follows:

- $s \in PROD(\Gamma)$
- $A' \in PROD(\Gamma)$ if $\exists A \in PROD(\Gamma)$ such that $A \to_{\Lambda,\tau} A'$.
- $A' \in PROD(\Gamma)$ if $\exists A \in PROD(\Gamma)$ such that $A \to_\Delta A'$.

Definition (Terminal Assemblies). The set of terminal assemblies for a Tile Automata system $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, \tau)$ is written as $TERM(\Gamma)$. This is the set of assemblies that cannot grow or transition any further. Formally, an assembly $A \in TERM(\Gamma)$ if $A \in PROD(\Gamma)$ and there does not exist any assembly $A' \in PROD(\Gamma)$ such that $A \to_{\Lambda,\tau} A'$ or $A \to_\Delta A'$.

A Tile Automata system $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, s, \tau)$ *uniquely* assembles an assembly A if $A \in TERM(\Gamma)$, and for all $A' \in PROD(\Gamma), A' \to_* A$.

3.1.3 Restrictions

Definition (Affinity Strengthening). We only consider transitions rules that are affinity strengthening, meaning for each transition rule $((\sigma_1, \sigma_2), (\sigma_3, \sigma_4), d)$, the bond between (σ_3, σ_4) must be at least the affinity strength of (σ_1, σ_2). Formally, $\Pi(\sigma_3, \sigma_4, d) \geq \Pi(\sigma_1, \sigma_2, d)$. This ensures that transitions may not induce cuts in the bond graph.

In the case of non-cooperative systems ($\tau = 1$), the affinity strength between states is always 1 so we may refer to the affinity function as an affinity set Λ_s, where each affinity is a 3-pule (σ_1, σ_2, d).

Definition (Freezing). Freezing systems were introduced with Tile Automata. A freezing system simply means that a tile may transition to any state only once. Thus, if a tile is in state A and transitions to another state, it is not allowed to ever transition back to A.

Definition (Deterministic vs. Nondeterministic). For clarification, a deterministic system in TA has only one possible production step at a time, whether that be an attachment or a state transition. A nondeterministic system may have many possible production steps and any choice may be taken.

Definition (Single-Transition System). We restrict our TA system to only use single-transition rules. This means that for each transition rule, one of the states may change, but not both. Note that nondeterminism is still allowed in this system.

3.2 Building Squares with Optimal State Complexity in Restricted Active Self-Assembly

Here, we study the state complexity of assembling $n \times n$ squares in seeded Tile Automata systems where growth starts from a seed and tiles may attach one at a time, similar to the abstract Tile Assembly Model. We provide optimal bounds for three classes of seeded Tile Automata systems (all without detachment), which vary in the amount of complexity allowed in the transition rules. We show that, in general, seeded Tile Automata systems require $\Theta(\log^{\frac{1}{4}} n)$ states. For single-transition systems, where only one state may change in a transition rule, we show a bound of $\Theta(\log^{\frac{1}{3}} n)$, and for deterministic systems, where each pair of states may only have one associated transition rule, a bound of $\Theta((\frac{\log n}{\log\log n})^{\frac{1}{2}})$. Along the way, we provide optimal bounds for the subroutines of building binary strings and building $O(\log n) \times n$ rectangles.

3.2.1 State Space Lower Bounds

In this section we derive information theoretic lower bounds on the minimum number of states needed to build length-n rectangles with a TA system. Of note is that the lower bounds achieved in this section match (for almost all n) the upper bounds for the respective model restrictions provided in Section 3.2.29. The key idea for each result is a counting argument that upper bounds the maximum number of distinct TA systems that exist for a given number of states $|\Sigma|$. For a sufficiently small number of states $|\Sigma|$, and a given integer n, we argue that the number of distinct systems with at most $|\Sigma|$ states is so much smaller than n that the pigeon-hole principle implies that most of the n distinct squares of length at most n cannot be self-assembled by such a system.

3.2.2 Encoding Lemma

To establish our bounds, we first formalize the concept of a proposition holding for *almost all n*. Let $p(n)$ be a function from the positive integers to the set $\{0,1\}$, informally termed a *proposition*, where 0 denotes the proposition being false and 1 denotes the proposition being true. We say a proposition $p(n)$ holds for *almost all n* if $\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} p(i) = 1$.

In the subsequent theorems for each model variant, we will upper bound the number of TA systems that exist for a given number of states indirectly by deriving a binary encoding for any such system. The length of the achieved binary encoding will imply an upper bound on the number of such systems (by raising the number of bits in the encoding to the power of 2). To this end, we first establish a Lemma showing that such an encoding scheme for some universe of TA systems must assign large strings (at least roughly size $\log n$ bits) to any TA system that self-assembles a length-n rectangle, for almost all n. We note that in the following Lemma, the value m will not appear as our proof is only based on the system uniquely building *something* of exactly length-n. We include the m parameter as the self-assembly of rectangles is an important benchmark class of shapes, and we want to be clear that our bounds apply to any such rectangle.

Lemma. Let U be a set of TA systems, $b(\cdot)$ be an injective function mapping each element of U to a string of bits, and ε a real number from $0 < \varepsilon < 1$. Then for almost all integers n, any TA system $\Gamma \in U$ that uniquely assembles an $n \times m$ rectangle for $n \geq m$ has a bit-string of length $|b(\Gamma)| \geq (1-\varepsilon)\log n$.

Proof. For a given $i \geq 1$, let $M_i \in U$ denote the TA system in U with the minimum value $|b(M_i)|$ over all systems in U that uniquely assemble an $i \times j$ rectangle for any j, $i \geq j$, and let M_i be undefined if no such system in U builds such a shape. Let $p(i)$ be the proposition that $|b(M_i)| \geq (1-\varepsilon)\log i$. We show that $\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} p(i) = 1$, which yields the Lemma. To show this we define the following. Let $R_n = \{M_i | 1 \leq i \leq n, |b(M_i)| < (1-\varepsilon)\log n\}$. Thus R_n denotes the set of all M_i that drop below the $(1-\varepsilon)\log n$ bound. Intuitively, we thus want to argue that R_n contains an arbitrarily small number of items as a fraction of n, i.e, that $\frac{1}{n}(n - |R_n|)$ has a limit of 1. Formally, this would show

our result since $n - |R_n| \leq \sum_{i=1}^{n} p(i)$ (due to the fact that membership in R_n only requires being less than $(1-\varepsilon)\log n$, as opposed to $(1-\varepsilon)\log i$ for each M_i). To show this, we apply the pigeon-hole principle to get that $|R_n| \leq 2^{(1-\varepsilon)\log n} = n^{(1-\varepsilon)}$. Therefore,

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} p(i) \geq \lim_{n\to\infty} \frac{1}{n}(n - |R_n|) \geq \lim_{n\to\infty} \frac{1}{n}(n - n^{1-\varepsilon}) = 1.$$

And since this limit is lower bounded by a sequence that converges to 1, and itself cannot exceed 1, we get that this limit converges to 1, concluding the proof. □

3.2.3 Rectangle Lower Bounds

We now move on to our lower bounds which each follow a similar format. For each model variant, we provide a way to explicitly encode any given TA system from the provided model into a binary string of some length dictated by the number of states in the system. We then apply Lemma 3.2.1 to this length to get a corresponding lower bound (for almost all n) for the self-assembly of length-n rectangles.

Theorem (Deterministic TA). For almost all n, any deterministic Tile Automata system that uniquely assembles an $n \times m$ rectangle with $n \geq m$ contains $\Omega(\frac{\log n}{\log\log n})^{\frac{1}{2}}$ states.

Proof. We create an injective mapping $b(\Gamma)$ from any deterministic TA system to bit-strings in the following manner. Let Σ denote the set of states in a given system. We encode the state set in $O(\log|\Sigma|)$ bits, we encode the affinity function in a $|\Sigma| \times |\Sigma|$ table of strengths in $O(|\Sigma|^2)$ bits (assuming a constant bound on bonding thresholds), and we encode the rules of the system in a $|\Sigma| \times |\Sigma|$ table mapping pairs of rules to their unique new pair of rules using $O(|\Sigma|^2 \log|\Sigma|)$ bits, for a total of $O(|\Sigma|^2 \log|\Sigma|)$ bits to encode any $|\Sigma|$ state system.

Let Γ_n denote the smallest state system that uniquely assembles an $n \times m$ rectangle, and let Σ_n denote the state set. By Lemma 3.2.1, $|b(\Gamma_n)| \geq (1-\varepsilon)\log n$ for almost all n, and so $|\Sigma_n|^2 \log|\Sigma_n| = \Omega(\log n)$ for almost all n. We know that $|\Sigma_n| = O(\log n)$ (Theorem 3.2.17 implies this upper bound), so for some constant c, $|\Sigma_n| \geq c(\frac{\log n}{\log\log n})^{\frac{1}{2}}$ for almost all n. □

Theorem 3.2.1 takes advantage of the deterministic rules to encode all rules as a table of labelled pairs of states, and the unique pairs of states they transition to, in size $O(|\Sigma|^2 \log |\Sigma|)$ bits. But such a compact encoding cannot handle non-deterministic rules, and so for Theorem 3.2.2 we instead resort to a brute force $O(|\Sigma|^4)$-bit encoding for system rules, which results in a *lower* lower bound.

Theorem (Nondeterministic TA). For almost all n, any Tile Automata system (nondeterministic) that uniquely assembles an $n \times m$ rectangle with $n \geq m$ contains $\Omega(\log^{\frac{1}{4}} n)$ states.

Proof. We create an injective mapping $b(\Gamma)$ from nondeterministic TA systems to bit-strings. We use the same mapping as in Theorem 3.2.1 except for the rule encoding, and now use a $|\Sigma|^4$ binary table to specify which rules are, or are not, present in the system. From Lemma 3.2.1, the minimum state system to build an $n \times m$ rectangle has $|\Sigma_n|^4 = \Omega(\log n)$ states for almost all n, which implies that $|\Sigma_n| = \Omega(\log^{\frac{1}{4}} n)$ for almost all n. □

Single-Transition TA systems offer a middle-ground between general TA and deterministic TA. They allow non-deterministic rules, preventing the compact encoding of Theorem 3.2.1, but can only change a single state per rule, reducing the dimension needed for a brute-force rule encoding and yielding $O(|\Sigma|^3)$-bit encodings, which in turn yields a middle-ground lower bound.

Theorem (Single-Transition TA). For almost all n, any single-transition Tile Automata system (nondeterministic) that uniquely assembles an $n \times m$ rectangle with $n \geq m$ contains $\Omega(\log^{\frac{1}{3}} n)$ states.

Proof. We use the same bit-string encoding $b(\Gamma)$ as in Theorem 3.2.1, except for the encoding of the ruleset, and we use a $|\Sigma|^2 \times |\Sigma|$ matrix of constants. The encoding works by encoding the pair of states for a transition in the first column, paired with the second column dictating the state that will change, and the entry in the table denoting which of the two possible states changed. We thus encode a $|\Sigma|$-state system in $O(|\Sigma|^3)$ bits, which yields $|\Sigma| = \Omega(\log^{\frac{1}{3}} n)$ for almost all n from Lemma 3.2.1. □

3.2.4 Strings (Patterns)

Here, we provide lower bounds on building binary strings. In order to avoid confusion, for the remainder of this section we will refer to the strings we are trying to build as *patterns*. Meaning an assembly represents a binary pattern, and a system is encoded as a binary string. We start by defining what it means for an assembly to represent a pattern. We then follow a similar structure to the rest of this section by proving an encoding lemma, and then apply this lemma to achieve bounds for each version of the model.

In order to keep our representation method general, we allow for arbitrary scaling and prove our lower bounds hold at any scale. We define scaled patterns using macroblocks. A macroblock is an $a \times b$ subassembly that maps to a specific bit of the pattern. Let $M^{a,b}(\Sigma)$ be the set of all size $a \times b$ macroblocks over states Σ. Let the i^{th} macroblock of an assembly be the size $a \times b$ subassembly whose lower left tile is at location $(a \cdot i, 0)$ in the plane.

Definition (Pattern Representation). An assembly A, over states Σ, represents a pattern P over a set of symbols L at scale $a \times b$, if there exists a mapping from the elements of $M^{a,b}(\Sigma)$ to the elements of L, and the i^{th} macroblock of A maps to the i^{th} symbol of S.

We prove the encoding lemma using the same technique as Lemma 3.2.1.

Lemma. Let U be a set of TA systems and b be an injective function mapping each element of U to a pattern of bits. Then for all $n \geq 0$, there exists a pattern P of length n such that any TA system $\Gamma \in U$ that uniquely assembles an assembly A that represents P at any scale has $|b(\Gamma)| \geq n$.

Proof. Given the string $b(\Gamma)$, the assembly A can be computed and the pattern P output after reading from each macroblock. There are 2^n length-n strings, but only $2^n - 1$ bit-strings with size less than n, so by the pigeonhole principle, at least one of the systems must map to a string of length n. □

Theorem (Deterministic TA). For all $n > 0$, there exists a binary pattern P of length n, such that any deterministic Tile Automata that uniquely assembles an assembly that represents P at any scale contains $\Omega((\frac{n}{\log|n|})^{\frac{1}{2}})$ states.

Proof. Using the encoding method described in Theorem 3.2.1, we represent a TA system with $|\Sigma|$ states in $O(|\Sigma|^2 \log |\Sigma|)$ bits. By Lemma 3.2.2, we know there exists a length-n binary pattern P such that any system Γ that uniquely assembles P requires n bits to describe.

$$O(|\Sigma|^2 \log |\Sigma|) \geq n \tag{3.1}$$

$$O(|\Sigma|^2) \geq \frac{n}{\log |\Sigma|} \tag{3.2}$$

$$|\Sigma| \geq \Omega(\frac{n}{\log |\Sigma|}^{\frac{1}{2}}) \tag{3.3}$$

A trivial construction of assigning each bit a unique state, gives $|\Sigma| \leq n$.

$$|\Sigma| \geq \Omega(\frac{n}{\log n}^{\frac{1}{2}})$$

□

Theorem (Single-Transition TA). For all $n > 0$, there exists a binary pattern P of length n, such that any single-transition Tile Automata that uniquely assembles an assembly that represents P at any scale contains $\Omega(n^{\frac{1}{3}})$ states.

Proof. With Theorem 3.2.3, we may encode the system with $O(|\Sigma|^3)$ bits and Lemma 3.2.2 implies there exists a pattern where $O(|\Sigma|^3) \geq n$, and thus $|\Sigma| \geq \Omega(n^{\frac{1}{3}})$. □

Theorem (Nondeterministic TA). For all $n > 0$, there exists a binary pattern P of length n, such that any Tile Automata (in particular any nondeterministic system) that uniquely assembles an assembly that represents P at any scale contains $\Omega(n^{\frac{1}{4}})$ states.

Proof. The encoding method from Theorem 3.2.2 and bound from Lemma 3.2.2 give us a lower bound of $\Omega(n^{\frac{1}{4}})$ states. □

A	B	S
2	2	0
2	1	1
2	0	1
1	2	1
1	1	0
1	0	1
0	2	1
0	1	0
0	0	0

(a) Encoding S

S_1S_2	$S_{1F}S_{2F}$
$2'_A$ $2''_B$	$2'_A$ 0_S
2_A 1_B	2_A 1_S
2_A 0_B	2_A 1_S
$1'_A$ $2''_B$	$1'_A$ 1_S
1_A 1_B	1_A 0_S
1_A 0_B	1_A 1_S
$0'_A$ $2''_B$	$0'_A$ 1_S
0_A 1_B	0_A 0_S
0_A 0_B	0_A 0_S

(b) Transition Rules

Figure 3.1: (a) A table indexing the string $S = 011101100$ using two columns with base $\lceil |S|^{\frac{1}{2}} \rceil$. (b) Transition rules to print S. We build an assembly where each row has a unique pair of index states in ascending order.

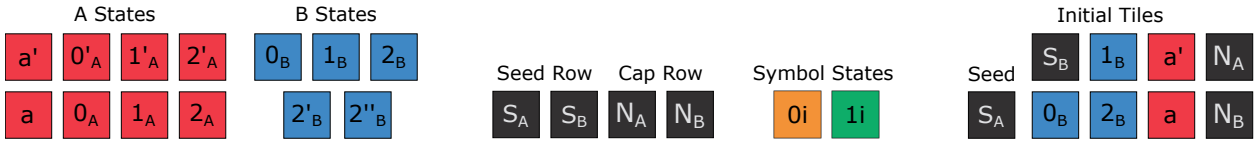


Figure 3.2: States to build a length-9 string in deterministic Tile Automata.

3.2.5 String Unpacking

A key tool in our constructions is the ability to build strings efficiently. We do so by encoding the string in the transition rules. We begin by describing the simplest version of this construction that uses freezing deterministic single-transition rules while achieving $\mathcal{O}(n^{\frac{1}{2}})$ states. In the case of 1-dimensional Tile Automata, we achieve the same bound with freezing, deterministic rules.

3.2.6 Freezing Deterministic Single-Transition

We start by showing how to encode a binary string of length n in a set of (freezing) transition rules that take place on a $2 \times (n+2)$ rectangle, and that will print the string on its right side. We extend this construction to work for an arbitrary-base string.

3.2.7 Overview

Consider a system that builds a length-n string. First, we create a rectangle of index states that is two wide, as seen on the left side of Figure 3.1b. Each row has a unique pair of index states, so each bit of the string is uniquely indexed. We divide the index states into two groups based on which column they are in, and which "digit" they represent. Let $r = \lceil n^{\frac{1}{2}} \rceil$. Starting with index states A_0 and B_0, we build a counter pattern with base r. We use $\mathcal{O}(n^{\frac{1}{2}})$ states, shown in Figure 3.2, to build this pattern. We encode each bit of the string in a transition rule between the two states that index that bit. A table with these transition rules can be seen in Figure 3.1a.

The pattern is built in r sections of size $2 \times r$ with the first section growing off of the seed. The tile in state S_A is the seed. There is also a state S_B that has affinity for the right side of S_A. The building process is defined in the following steps for each section.

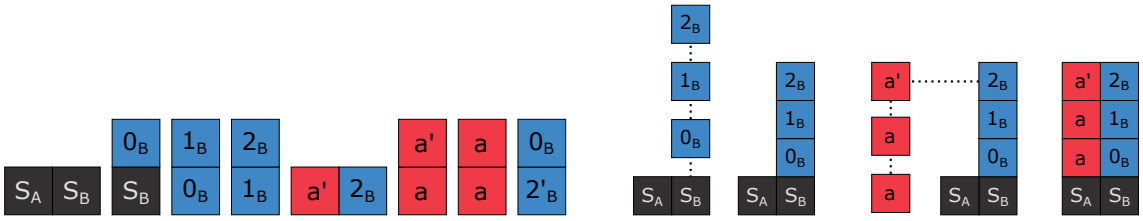
1. The states $S_B, 0_B, 1_B, \ldots, (r-1)_B$ grow off of S_B, forming the right column of the section. The last B state allows for a' to attach on its west side. State a tiles attach below a' and below other a tiles. This places a states in a row south toward the state S_A, depicted in Figure 3.3b.

2. Once a section is built, the states begin to follow their transition rules shown in Figure 3.4a. The a state transitions with seed state S_A to begin indexing the A column by changing state a to state 0_A. For $1 \leq y \leq n-2$, state a vertically transitions with the other y'_A states, incrementing the index by changing from state a to state $(y+1)_A$.

3. This new index state z_A, propagates up by transitioning the a tiles to the state z_A as well. Once the z_A state reaches a' at the top of the column, it transitions a' to the state z'_A. Figure 3.4b presents this process of indexing the A column.

4. If $z < n-1$, there is a horizontal transition rule from states $(z'_A, (n-1)_B)$ to states $(z'_A, (n-1)'_B)$. The state 0_B attaches to the north of $(n-1)_B$ and starts the next section. If $z = n$, there does not exist a transition.

5. This creates an assembly with a unique state pair in each row as seen in the first column of



(a) Affinity Rules for Initial Tiles (b) Process of Building a section

Figure 3.3: (a) Affinity rules to build each section. We only show affinity rules that are actually used in the system for initial tiles to attach, but the system has more rules in order to meet the affinity-strengthening restriction. (b) The B column attaches above the state S_B as shown by the dotted lines. The a' attaches to the left of 2_B and the other a states may attach below it until they reach S_A.

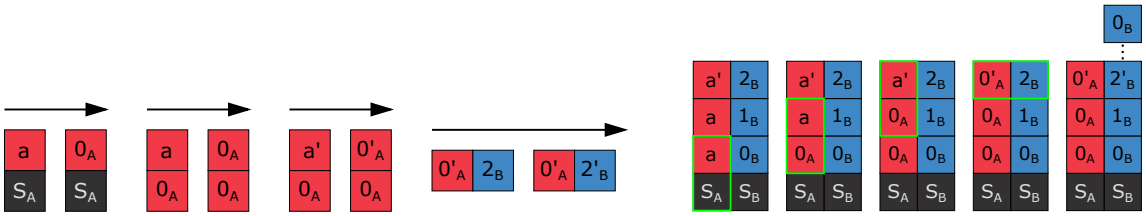
Figure 3.1b.

3.2.8 States

An example system with the states required to print a length-9 string are shown in Figure 3.2. The seed tile has state S_A with initial tiles in state S_B. The index states are divided into two groups. The first set of index states, which we call the A index states, are used to build the left column. For each i, $0 \leq i < r$, we have the states i_A and i'_A. There are two states a and a', that exist as initial tiles and act as "blank" states that can transition to the other A states. The second set of index states are the B states. Again, we have r B states numbered from 0 to $r-1$, however, there is not a prime for each state. Instead, there are two states $r-1'_B$ and $r-1''_B$, that are used to control the growth of the next column and the printing of the strings. The last states are the symbol states 0_S and 1_S, which are the states that represent the string.

3.2.9 Affinity Rules / Placing Section

Here, we describe the affinity rules for building the first section. We later describe how this is generalized to the other $r-1$ sections. We walk through this process in Figure 3.3b. To begin, the B states attach in sequence above the tile S_B in the seed row. Assuming $r^2 = n$, n is a perfect square, the first state to attach is 0_B. 1_B attaches above this tile and so on. The last B state, $(r-1)_B$, does not have affinity with 0_B, so the column stops growing. However, the state a' has affinity on the left of $(r-1)_B$ and can attach. a has affinity for the south side of a', so it attaches below. The a state



(a) Transition Rules to Index the first section (b) Process of Indexing A column

Figure 3.4: (a) The first transition rule used takes place between the seed S_A and the a state changing to 0_A. The state 0_A changes the states north of it to 0_A or $0'_A$. Finally, the state $0'_A$ transitions with 2_B (b) Once the a states reach the seed row, they transition with the state S_A to go to 0_A. This state propagates upward to the top of the section.

also has a vertical affinity with itself. This grows the A column southward toward the seed row.

If n is not a perfect square, we start the index state pattern at a different value. We do so by finding the value $q = r^2 - n$. In general, the state i_B attaches above S_B for $i = q \mod r$.

3.2.10 Transition Rules / Indexing A column

Once the A column is complete and the last A state is placed above the seed, it transitions with S_A to 0_A (assuming $r^2 = n$). A has a vertical transition rule with i_A ($0 \leq i < r$) changing the state A to state i_A. This can be seen in Figure 3.4a, where the 0_A state is propagated upward to the A' state. The A' state also transitions when 0_A is below it, going from state A' to state $0'_A$. If n is not a perfect square, then A transitions to i_A for $i = \lfloor q/r \rfloor$.

Once the transition rules have finished indexing the A column if $i < r-1$, the last state, i'_A, transitions with $(r-1)_B$, changing the state $(r-1)_B$ to $(r-1)'_B$. This transition can be seen in Figure 3.4b. The new state $(r-1)'_B$ has an affinity rule allowing 0_B to attach above it, and allowing the next section to be built. When the state A is above a state j'_A, $0 \leq j < r-1$, it transitions with that state, thus changing from state A to $(j+1)_A$, which increments the A index.

3.2.11 Look up

After creating a $2 \times (n+2)$ rectangle, we encode a length-n string S into the transitions rules. Note that each row of our assembly consists of a unique pair of index states, which we call a *bit gadget*. Each bit gadget will *look up* a specific bit of our string and transition the B tile to a state representing the value of that bit.

Figure 3.1a shows how to encode a string S in a table with two columns using r digits to index each bit. From this encoding, we create our transition rules. Consider the k^{th} bit of S (where the 0^{th} bit is the least significant bit) for $k = ir + j$. Add transition rules between the states i_A and j_B, changing the state j_B to either 0_S or 1_S based on the k^{th} bit of S. This transition rule is slightly different for the northmost row of each section as the state in the A column is i'_A. Also, we do not want the state in the B column, $(r-1)_B$, to prematurely transition to a symbol state. Thus, we have the two states $(r-1)'_B$ and $(r-1)''_B$. As mentioned, once the A column finishes indexing, it changes the state $(r-1)_B$ to state $(r-1)'_B$, thus allowing for 0_B to attach above it, which starts the next column. Once the state 0_B (or a symbol state) is above $(r-1)'_B$, there are no longer any possible undesired attachments, so the state transitions to $(r-1)''_B$, which has the transition to the symbol state.

The last section has a slightly different process as the $(r-1)_B$ state will never have a 0_B attach above it, so we have a different transition rule. This alternate process is shown in Figure 3.5. The state $(r-1)'_A$ has a vertical affinity with the cap state N_A. This state allows N_B to attach on its right side. This state transitions with $(r-1)_B$ below it, changing it directly to $(r-1)''_B$, thus allowing the symbol state to print.

Theorem. For any binary string S with length $n > 0$, there exists a freezing Tile Automata system Γ_s with deterministic single-transition rules, that uniquely assembles a $2 \times (n+2)$ assembly A_S that represents S with $\mathcal{O}(n^{\frac{1}{2}})$ states.

Proof. We discuss the system in parts.

States and Affinity. Let $r = n^{\frac{1}{2}}$. We construct the system Γ_s as follows. We use $O(r)$ index states and a constant number of seed row and symbol states as described above. The seed tile is the state S_A, and our initial tiles are the B index states, and the two states a and a'. The affinity rules are also described above. Due to affinity strengthening, if one state σ_1 may transition to another state σ_2, then σ_1 must have at least the same affinity rules as σ_2. This does not cause an issue for the A column as the a state already has affinity with itself, so all the A states (besides the states at the top of the column with subscript A') will have vertical affinity with each other. The top tile of each A

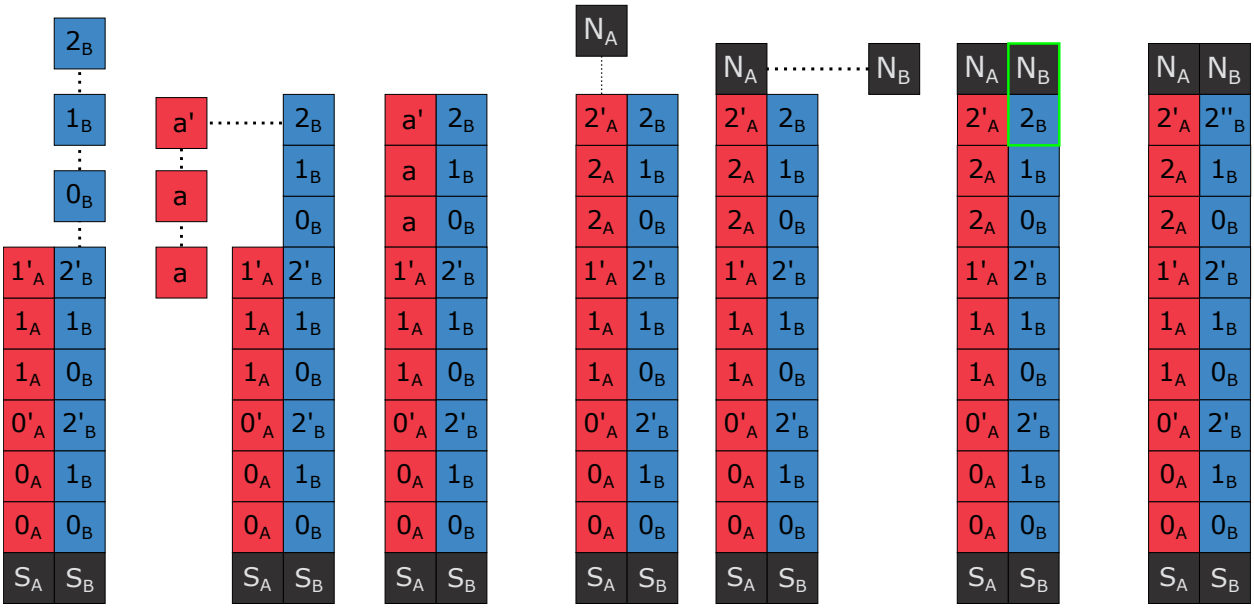


Figure 3.5: (a) Once the last section finishes building, the state N_A attaches above $2'_A$. N_B then attaches to the assembly and transitions with 2_B, and changes it directly to $2''_B$, so the string may begin printing.

column does not have affinity with any states on its north, and it never gains one by transitioning. For $0 \leq i < r-1$, the state i_B has a vertical affinity with $(i+1)_B$. These B index states transition to symbol states, so the symbol states must have vertical affinities with all the B states.

We encode the starting values of the indices in the affinity rule of S_B for the B index, and the transition rule between A and S_A for the A column. In the case that the first B state is $(r-1)_B$, the state that is above S_A will be a'. In this case, we transition a' with S_A, thus changing it directly to $0'_A$.

Transition Rules. Our transition rules to build the sections are described above. This system has deterministic rules since each pair of states has up to one transition rule between them. Now, we formally describe the method to encode string S.

In Figure 3.1a, S is drawn as a table with indices of up to r. Let $S(\alpha,\beta)$ be the k^{th} bit of S where $k = \alpha r + \beta - O$. For $\beta < r-1$, we have a transition rules between states (α_A, β_B) transitioning to $(\alpha_A, 0i)$ if $S(\alpha,\beta) = 0$, or to $(\alpha_A, 1i)$ if $S(\alpha,\beta) = 1$. When $\beta = r-1$, we have the same rule but between (α_A, β''_b). An overview of these rules can be seen in Figure 3.1b.

Assembly. The assembly A_S has the seed row states as its bottom row, followed by rows with an A index state on the left and a symbol state on the right. The top row has the two cap

states N_A and N_B. This assembly is terminal since none of the initial tiles may attach to the north or south row, the A index states do not have any left affinities, and the symbol states do not have any right affinities. This assembly is uniquely produced given how each section is built with only one available move at each step when building the first section until the A column begins indexing. When the A column starts indexing, it is able to change the states of the tiles in the B column. The affinity strengthening requirement forces the symbol states to have affinity with all the B states. This does not cause an issue for most of the B states as the surrounding tiles have already attached, but this is why the state $(r-1)_B$ does not transition to a symbol state, yet instead waits until the next section has started (or the cap row is present) to transition to $(r-1)''_B$. □

3.2.12 Arbitrary Base

In order to optimally build rectangles, we first print arbitrary base strings. Here, we show how to generalize Theorem 3.2.7 to print base-b strings.

Corollary. For any base-b string S with length $n > 0$, there exists a freezing Tile Automata system Γ with deterministic single-transition rules, that uniquely assembles an $(n+2) \times 2$ assembly that represents S with $\mathcal{O}(n^{\frac{1}{2}} + b)$ states.

Proof. In order to print base-b strings, we use b index states from 0_S to $(b-1)_S$. We encode the strings in our transitions the same way as the above proof by transitioning the states (α_A, β_B) to (α_A, k_S) for $k = S(\alpha, \beta)$. □

3.2.13 Optimal Bounds

Using Corollary 3.2.1 and base conversion techniques from (L. Adleman et al. 2001), we achieve the optimal bound for binary strings. The techniques from previous work extend the size of the assembly to a non-constant height.

Theorem. For any binary string S with length $n > 0$, there exists a freezing Tile Automata system Γ_s with deterministic single-transition rules, that uniquely assembles an $n \times (n+2)$ assembly A_S that represents S with $\Theta(\frac{n}{\log n}^{\frac{1}{2}})$ states.

Proof. Divide S into $\frac{|S|}{\log|S|}$ segments- each of length $\log|S|$. Build a length $\frac{|S|}{\log|S|}$ base-$\frac{n}{\log n}^{\frac{1}{2}}$ string in $\mathcal{O}(\frac{n}{\log n}^{\frac{1}{2}})$ states using Corollary 3.2.1. Then using the base conversion technique from (L. Adleman et al. 2001) to achieve the final binary string. □

3.2.14 Height-1 Strings

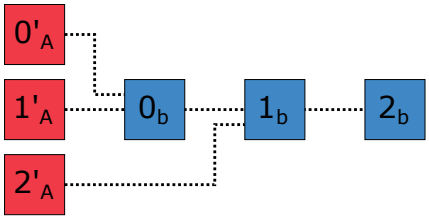
Here, we present an alternate construction to achieve the same bound as above, but at exact scale. This system, however, does not have single-transitions. At a high level, our construction works the same way: by encoding strings in pairs of index states. However, since we are constructing an exact size $1 \times n$ assembly, we must be careful with our arrangement. We do so by building and unpacking one $1 \times r$ section at a time, which is accomplished by having a single A state "walk" across r adjacent B states. Each time A passes over a B state, it leaves a symbol state. The process for the first section can be seen in Figure 3.6a.

Affinity Rules. The seed tile is $0_{A'}$, and it has affinity with 0_b that allows the b tiles to attach. Example affinity rules are shown in Figure 3.6b. We include r of these b states to construct each full section. However, since the last section does not allow the tiles to attach, we have $r_{A'}$ that allows 1_b to attach to keep the correct length.

Transition Rules. The states $r-1_b$ and r_b transition to $r-1_B$ and r_B, respectively. Then each i_b and $i+1_B$ transitions, changing the first state to i_B. These transitions signal that the segment is finished building and the A' state may start unpacking. The first transition that unpacks is a special case between the seed state $0_{A'}$ and 0_B because we need the leftmost tile to have no affinity with any other state, so this enters the 0_S or 1_S symbol state, and the 0_B changes to 0_A. For $0 \leq \beta < r-1$, the states α_A and β_B transition the α_A to the symbol state for $S(\alpha,\beta)$ and β_B to α_A. If $\beta = r-1$, then β_B instead transitions to $\alpha+1_{A'}$. The final transition between the index states unpacks both numbers, and the right state transitions to an end state marked with T, which does not have affinity with anything on its right, thus making the assembly terminal.

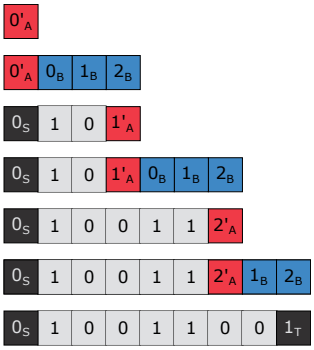
Theorem. For any base-b string S with length $n > 0$, there exists a freezing Tile Automata system Γ with deterministic transition rules, that uniquely assembles a $1 \times n$ assembly that represents

(a) Transition Rules that occur on the first segment of assembly

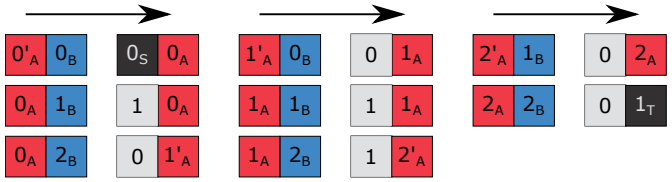


(b) Affinity Rules

Figure 3.6: (a) Once all of the b tiles attach to the seed, they begin to transition to B states. The A' state transitions with these B states to unpack the string. Once it reaches the final B state, it increments to state $1_{A'}$ (b) Affinity rules used to build each section. The 0_b initial tile attach to the left of the A' states, however the state $2_{A'}$ is the final segment, so it has affinity with 1_b to achieve the exact length.



(a) Construction to build string S

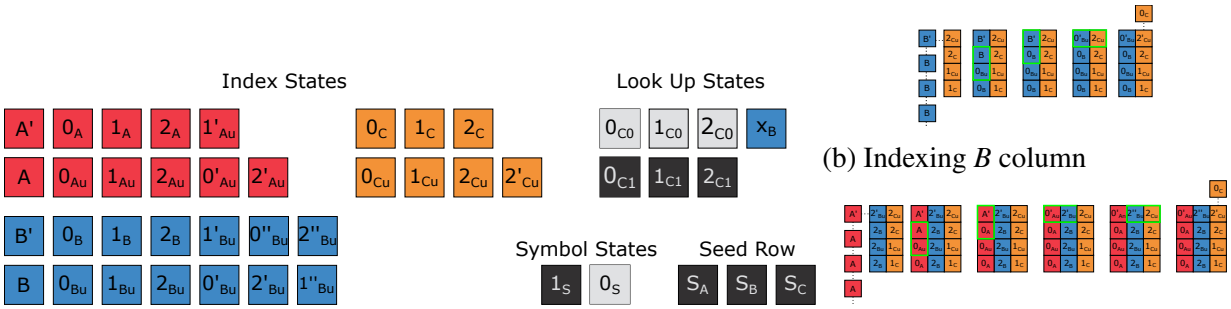


(b) Transition rules to encode S

Figure 3.7: (a) Each segment is built using $n^{\frac{1}{2}}$ tiles. The A state moves along the assembly, leaving a symbol state in the previous location. Each time the A state reaches 2_B it increments to an A' state allowing for another segment to attach. (b) The transition rules which encode each symbol.

S with $\mathcal{O}(n^{\frac{1}{2}}+b)$ states.

Proof. Figure 3.7a outlines the process. Starting from the seed tile $0_{A'}$, each section of B states builds and transitions. The A' state increments after reaching the final B state. This allows another section to build and the process to repeat until the last section, which builds one tile shorter. The system is freezing since each tile goes through the following states b, B, A, then finally a symbol state. The system is deterministic since each pair of states only has a single transition. While the figures depict encoding a binary string, this method can be used to encode a string in any base. □



(a) States space for when $|S| = 27$

(b) Indexing B column

(c) Indexing the A column

Figure 3.8: (a) States needed to construct a length 27 string where $r = 3$. (b) The index 0 propagates upward by transitioning the tiles in the column to 0_B and 0_{Bu}, and transitions a' to $0'_{Bu}$. The state $0'_{Bu}$ transitions with the state 2_{Cu}, and changes the state 2_{Cu} to $2'_{Cu}$, which has affinity with 0_C to build the next section. These rules also exist for the index 1. (c) When the index state 2_B reaches the top of the section, it transitions b' to $2'_{Bu}$. This state does not transition with the C column, but instead has affinity with the state a', which builds the A column downward. The index propagates up the A column in the same way as the B column. When the index state 0_A reaches the top of the section, it transitions the state $2'_B$ to $2''_B$. This state transitions with 2_{Cu}, and changes it to $2'_{Cu}$ allowing the column to grow.

3.2.15 Nondeterministic Single-Transition Systems

For the case of single-transition systems, we use the same method from above, but instead build bit gadgets that are of size 3×2. Expanding to 3 columns allows for a third index digit to be used, and thus giving an upper bound of $O(n^{\frac{1}{3}})$. The second row is used for error checking, which we describe later in the section. This system utilizes nondeterministic transitions (two states may have multiple rules with the same orientation), and is non-freezing (a tile may repeat states). The system also contains cycles in its production graph, which implies the system may run indefinitely. We conjecture this system has a polynomial run-time. Here, let $r = \lceil n^{\frac{1}{3}} \rceil$.

3.2.16 Index States and Look-Up States

We generalize the method from above to start from a C column. The B column now behaves as the second index of the pattern and is built using B' and B as the A column was in the previous system. Once the B reaches the seed row, it is indexed with its starting value. This construction also requires bit gadgets of height 2, so we use index states i_A, i_B, i_C and north index states i_{Au}, i_{Bu}, i_{Cu} for $0 \leq i < r$. This allows us to separate the two functions of the bit gadget into each row. The north

row has transition rules to control the building of each section. The bottom row has transition rules that encode the represented bit.

In addition to the index states, we use $2r$ look-up states, 0_{Ci} and 1_{Ci} for $0 \leq i < r$. These states are used as intermediate states during the look-up. The first number (0 or 1) represents the value of the retrieved bit, while the second number represents the C index of the bit. The A and B indices of the bit will be represented by the other states in the transition rule.

In the same way as the previous construction, we build the rightmost column first. We include the C index states as initial states and allow 0_C to attach above S_C. We include affinity rules to build the column northwards as follows- starting with the southmost state $0_C, 0_{Cu}, 1_C, 1_{Cu}, \ldots, r-2_{Cu}, r-1_C, r-1_{Cu}$.

To build the other columns, the state b' can attach on the left of $r-1_{Cu}$. The state b is an initial state and attaches below b' and itself to grow downward toward the seed row. The state b transitions with the seed row, as in the previous construction, to start the column. However, we alternate between C states and Cu states. The state b above i_C transitions b to i_{Cu}. If b is above i_{Cu}, it transitions to i_C. The state b' above state i_B transitions to i'_{Bu}. If $i < r-1$, the states i'_B and $r-1_{Cu}$ transition horizontally and change $r-1'_{Cu}$, which allows 0_C to attach above it to repeat the process. This is shown in Figure 3.8b.

The state a' attaches on the left of $r-1_{Cu}$. The A column is indexed just like the B column. For $0 \leq i < r-1$, the state i'_{Au} and $r-1'_{Bu}$ change the state $r-1'_{Bu}$ to $r-1''_{Bu}$. This state transitions with $r-1_{Cu}$, changing it to $r-1'_{Cu}$. See Figure 3.8c.

3.2.17 Bit Gadget Look-Up

The bottom row of each bit gadget has a unique sequence of states. We use these index states to represent the bit indexed by the digits of the states. However, since we can only transition between two tiles at a time, we must read all three states in multiple steps. These steps are outlined in Figure 3.9. The first transition takes place between the states i_A and j_B. We refer to these transition rules as look-up rules. We have r look-up rules between these states (for $0 \leq k < r$) that changes the state j_B to that state k_{C0} if the bit indexed by i, j, and k is 0 or the state k_{C1} if the bit is 1.

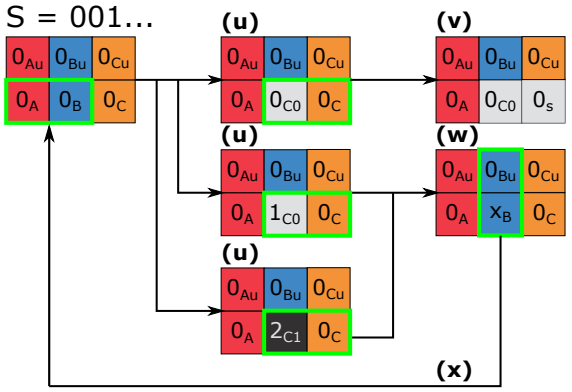


Figure 3.9: ST Bit Gadget look-up. (a.u) For a string S, where the first 3 bits are 001, the states 0_A and 0_B have $|S|^{\frac{1}{3}}$ transition rules changing the state 0_B to a state representing one of the first $|S|^{\frac{1}{3}}$ bits. The state is i_{C0} if the i^{th} bit is 0 or i_{C1} if the i^{th} bit is 1 (a.v) The state 0_{C0} and the state 0_C both represent the same C index so the 0_C state transition to the 0_s. (a.w) For all states not matching the index of 0_C, they transition to x_B, which can be seen as a blank B state. (a.x) The state 0_{Bu} transitions with the state x_B changing to 0_B resetting the bit gadget.

Our bit gadget has nondeterministically looked up each bit indexed by its A and B states. Now, we must compare the bit we just retrieved to the C index via the state in the C column. The states k_{C0} and k_C transition, changing the state k_C to the $0i$ state only when they represent the same k. The same is true for the state k_{C1} except C_k transitions to $1i$.

If they both represent different k, then the state k_C goes to the state B_x. This is the error checking of our system. The B_x states transition with the north state j_{Bu} above it, transitioning B_x to j_B once again. This takes the bit gadget back to its starting configuration and another look-up can occur.

Theorem. For any binary string S with length $n > 0$, there exists a single-transition Tile Automata system Γ, that uniquely assembles an $(2n+2) \times 3$ assembly which represents S with $O(n^{\frac{1}{3}})$ states.

Proof. Let $r = n^{\frac{1}{3}}$, note that we use $O(r)$ index states and look-up states in our system. The number of other states in our system is bounded by a constant, so the total number of states in Γ is $O(n^{\frac{1}{3}})$. We use affinity and transition rules to place tiles and index the columns as described above.

Note that the transition rules to index columns and the transition rules to signal for another section to build, only ever change one of the states involved. The transition rules for the look-up do

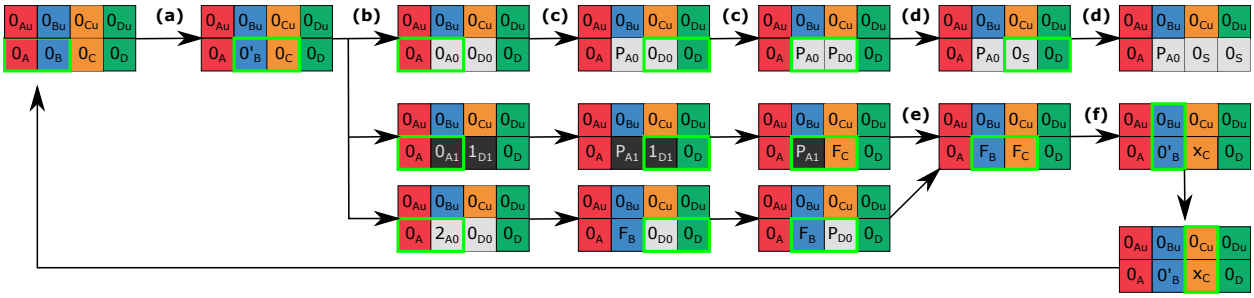


Figure 3.10: Nondeterministic Bit Gadget look-up. (b.a) Once the state A_0 appears in the bit gadget, it transitions with 0_B changing 0_B to $0'_B$. (b.b) The states $0'_B$ and 0_C nondeterministically look-up bits with matching B and C indices. The state $0'_B$ transitions to the look-up state representing the bit retrieved and the bit's A index. The state 0_C transitions to a look-up state representing the D index of the retrieved bit. (b.c) The look-up states transition with the states 0_A and 0_D, respectively. As with the single-transition construction, these may pass or fail. (b.d) When both tests pass, they transition the D look-up state to a symbol state that propagates out. (b.e) If a test fails, the states both go to blank states. (b.f) The blank states then reset using the states to their north.

the same as well. Only the B state changes to a look-up state. When the look-up state transitions with C, if they both represent the same index, C transitions to the symbol state of the retrieved bit. If they have different indices, then the look-up state transitions to Bx. Bx transitions with the index state above it, and resets itself. All of these rules only ever change one tile.

As with Theorem 3.2.7, the northernmost and southernmost rows do not allow other states to attach above/below them, respectively. The states in the A column do not allow tiles to attach on their left. The rightmost column has only Cu and symbol states ($1s$ or $0s$), which do not allow tiles to attach on their right.

The last column to be indexed in this construction is the A column. Bit gadgets also do not begin to transition until the A state is indexed, so no matter the build order, the states that are used will be present before the gadget begins transitioning. Until the A column begins indexing, there is only one step that can take place so we know there are not build orders that result in other terminal assemblies. □

3.2.18 General Nondeterministic Transitions

Using a similar method to the previous sections, we build length-n strings using $O(n^{\frac{1}{4}})$ states. We start by building a pattern of index states with bit gadgets of height 2 and width 4.

3.2.19 Overview

Here, let $r = \lceil n^{\frac{1}{4}} \rceil$. We build index states in the same way as the single-transition system, but instead building starts from the D column. We have four sets of index states: A, B, C, D. The same methods are used to control the next section by transitioning the state $r-1_D$ to $r-1'_D$ when the current section is finished building.

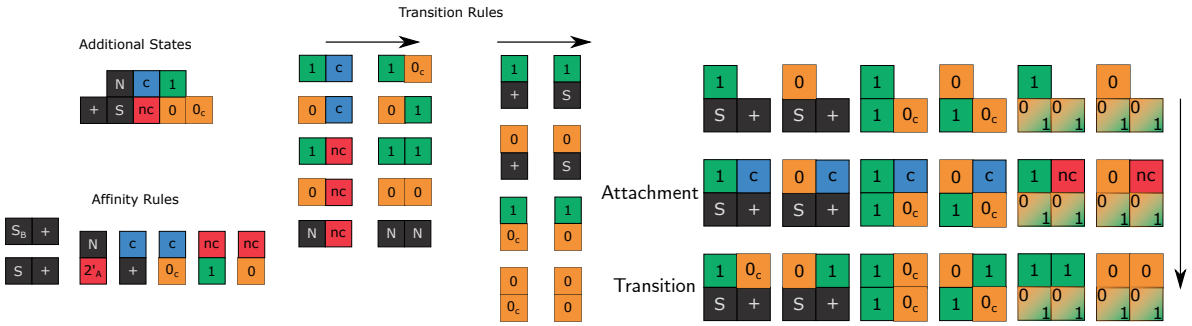
We use a similar look-up method as the previous construction where we nondeterministically retrieve a bit. However, since we are not restricting our rules to be a single-transition system, we may retrieve 2 indices in a single step. We include 2 sets of $O(r)$ look-up states, the A look-up states and the D look-up states. We also include Pass and Fail states $F_B, F_C, P_{A0}, P_{D0}, P_{A1}, P_{D1}$ along with the blank states B_x and C_x. We utilize the same method to build the north and south row.

Let $S(\alpha,\beta,\gamma,\delta)$ be the i^{th} bit of S where $i = \alpha r^3 + \beta r^2 + \gamma r + \delta$. The states β'_B and γ_C have r^2 transitions rules. The process of these transitions is outlined in Figure 3.10. They transition from (β'_B, γ_C) to either $(\alpha_{A0}, \delta_{D0})$ if $S(\alpha,\beta,\gamma,\delta) = 0$, or $(\alpha_{A1}, \delta_{D1})$ if $S(\alpha,\beta,\gamma,\delta) = 1$. After both transitions have happened, we test if the indices match the actual A and D indices. We include the transition rules (α_A, α_{A0}) to (α_A, P_{A0}) and (α_A, α_{A1}) to (α_A, P_{A1}). We refer to this as the bit gadget passing a test. The two states (P_{A0}, P_{D0}) horizontally transition to $(P_{A0}, 0s)$. The $0s$ state then transitions the state δ_D to $0s$ as well as propagating the state to the right side of the assembly. If the compared indices are not equal, then the test fails and the look-up states will transition to the fail states F_B or F_C. These fail states will transition with the states above them, resetting the bit gadget as in the previous system.

Theorem. For any binary string S with length $n > 0$, there exists a Tile Automata system Γ, that uniquely assembles an $(2n+2) \times 4$ assembly which represents S with $\mathcal{O}(n^{\frac{1}{4}})$ states.

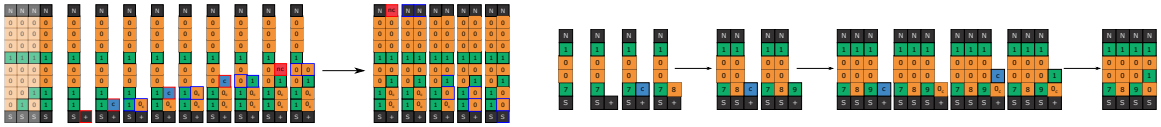
Proof. Let $r = \lceil n^{\frac{1}{4}} \rceil$. We use $O(r)$ index states to build our bit gadgets. We have $O(r)$ look-up states and a constant number of pass, fail, and blank states to perform the look-up.

This system uniquely constructs a $(2n+2) \times 4$ rectangle that represents the string S. Each bit gadget is a constant height and represents a unique bit of the string. In each bit gadget, the test



(a) New states and rules for a binary counter (b) Every case for the half adder.

Figure 3.11: (a) The states used for a binary counter. Note we are augmenting the system used to create strings. (b) The 0/1 tile is not present in the system. It is used in the diagram to show that either a 0 tile or a 1 tile can take that place.



(a) Binary Counter (b) Base-10 Counter

Figure 3.12: (a) The process of the binary counter. (b) A base-10 counter.

ensures that only the correct bit is retrieved and propagated to the right side of the assembly. By the same argument as the previous constructions, this assembly is terminal. A bit gadget does not begin looking up bits until its A column is complete (by transitioning β_B to β'_B) so the system uniquely constructs this rectangle as one move can happen at a time to build the bit gadgets. Once a bit gadget is complete, it does not affect surrounding tiles as transitions only occur between tiles in the same gadget. □

3.2.20 Rectangles

In this section, we show how to use the previous constructions to build $\mathcal{O}(\log n) \times n$ rectangles. All of these constructions rely on using the previous results to encode and print a string, then adding additional states and rules to build a counter.

3.2.21 States

We choose a string and construct a system that creates that string by using the techniques shown in the previous section. We then add states to implement a binary counter that counts up

from the initial string. The states of the system, seen in Figure 3.11a, have two purposes. The north and south states (N and S) are the bounds of the assembly. The plus, carry, and no carry states ('+', 'c', and 'nc') forward the counting. The '1', '0', and '0 with a carry' state make up the number. The counting states and the number states work together as half-adders to compute bits of the number.

3.2.22 Transition Rules / Single-Tile Half-Adder

As the column grows, in order to complete computing the number, each new tile attached in the current column, along with its west neighbor, are used in a half-adder configuration to compute the next bit. Figure 3.11b shows the various cases for this half-adder.

When a bit is going to be computed, the first step is an attachment of a carry tile or a no-carry tile ('c' or 'nc'). A carry tile is attached if the previous bit has a carry, indicated by a tile with a state of plus or '0 with a carry' ('+' or '0c'). A no-carry tile is placed if the previous bit has no-carry, indicated by a tile with a state of '0' or '1'. Next, a transition needs to occur between the newly attached tile and its neighbor to the west. This transition step is the addition between the newly placed tile and the west neighbor. The neighbor does not change states, but the newly placed tile changes into a number state, '0' or '1', that either contains a carry or does not. This transition step completes the half-adder cycle, and the next bit is ready to be computed.

3.2.23 Walls and Stopping

The computation of a column is complete when a no-carry tile is placed next to any tile with a north state. The transition rule changes the no-carry tile into a north state, thus preventing the column from growing any higher. The tiles in the column with a carry then transition to remove the carry information since it is no longer needed for computation. A tile with a carry changes states into a state without the carry. The next column can begin computation when the plus tile transitions into a south tile, thus allowing a new plus tile to be attached. The assembly stops growing to the right when the last column gets stuck in an unfinished state. This column, the stopping column, has carry information in every tile that is unable to transition. When a carry tile is placed next to a north tile, there is no transition rule to change the state of the carry tile, thus preventing any more growth

to the right of the column.

Theorem. For all $n > 0$, there exists a Tile Automata system that uniquely assembles a $\mathscr{O}(\log n) \times n$ rectangle using,

- Deterministic Transition Rules and $\mathscr{O}(\log^{\frac{1}{2}} n)$ states.
- Single-Transition Rules and $\Theta(\log^{\frac{1}{3}} n)$ states.
- Nondeterministic Transition Rules and $\Theta(\log^{\frac{1}{4}} n)$ states.

3.2.24 Arbitrary Bases

Here, we generalize the binary counter process for arbitrary bases. The basic functionality remains the same. The digits of the number are computed one at a time going up the column. If a digit has a carry, then a carry tile attaches to the north, just like the binary counter. If a digit has no carry, then a no-carry tile is attached to the north. The half adder addition step still adds the newly placed carry or no-carry tile with the west neighbor to compute the next digit. This requires adding $O(b)$ counter states to the system, where b is the base.

Theorem. For all $n > 0$, there exists a deterministic Tile Automata system that uniquely assembles a $\mathscr{O}(\frac{\log n}{\log\log n}) \times n$ rectangle using $\Theta\left((\frac{\log n}{\log\log n})^{\frac{1}{2}}\right)$ states.

Proof. Note that all logarithms shown are base 2. There are two cases. Case 1: $n \leq 2^{16}$. Let c be the minimum number of states such that, for all $w < n' < 2^{16}$, a $w \times n'$ rectangle can be uniquely assembled with $\leq c$ states. We utilize c states to uniquely assemble a rectangle of the desired size.

Case 2: $n > 2^{16}$. Let $b = \lceil(\frac{\log n}{\log\log n})^{\frac{1}{2}}\rceil$, and $d = 4\lceil\frac{\log n}{\log\log n}\rceil$. We initialize a variable base counter with value $b^d - n$ represented in base b. The binary counter states then attach to this string, counting up to b^d, for a total length of n. We prove that for $n > 2^{16}$ that $b^d \geq n$. Let $b' = (\frac{\log n}{\log\log n})^{\frac{1}{2}}$ and $d' = \frac{4\log n}{\log\log n}$.

$$\log(b^d) \geq \log(b'^{d'})$$

$$\log(b^d) \geq d' \log(b')$$

$$\log(b^d) \geq d' \log\left(\left(\frac{d'}{4}\right)^{\frac{1}{2}}\right)$$

$$\log(b^d) \geq \frac{d'}{2} \log\left(\frac{d'}{4}\right)$$

$$\log(b^d) \geq \frac{2\log n}{\log\log n} \log\left(\frac{\log n}{\log\log n}\right)$$

$$\log(b^d) \geq \frac{2\log n}{\log\log n} (\log\log n - \log\log\log n)$$

$$\log(b^d) \geq 2\log n - 2\log n \left(\frac{\log\log\log n}{\log\log n}\right)$$

Since $n > 2^{16}$, $\frac{\log\log\log n}{\log\log n} < \frac{1}{2}$.

$$\log(b^d) \geq 2\log n - 2\log n \left(\frac{1}{2}\right)$$

$$\log(b^d) \geq \log n$$

$$2^{\log(b^d)} \geq 2^{\log n}$$

$$b^d \geq n$$

By Corollary 3.2.1, we create the assembly that represents the necessary d-digit base-b string with $d^{\frac{1}{2}} + b = \Theta\left((\frac{\log n}{\log\log n})^{\frac{1}{2}}\right)$ states. The counter that builds off this string requires $\mathscr{O}(b)$ unique states. Therefore, for any $n > 0$ and constant c, there exists a system that uniquely assembles a $\mathscr{O}(\frac{\log n}{\log\log n}) \times n$ rectangle with $\Theta(b) + \Theta(d^{\frac{1}{2}}) + c = \Theta\left((\frac{\log n}{\log\log n})^{\frac{1}{2}}\right)$ states. □

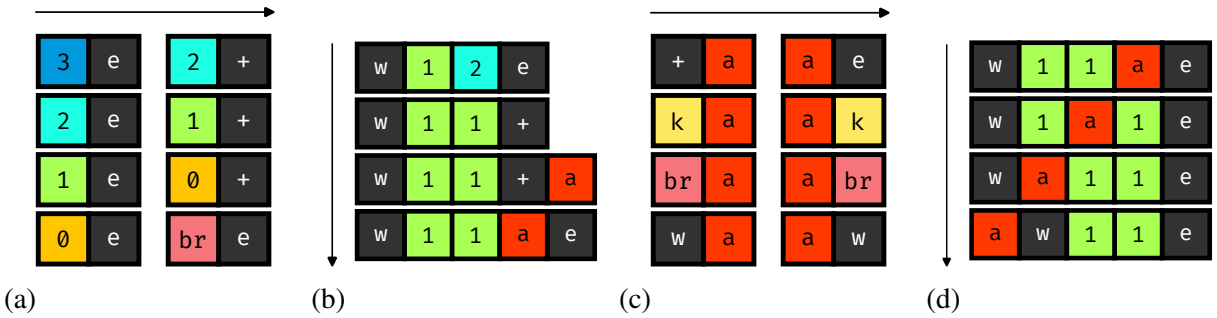


Figure 3.13: Example methods using a base-3 number. (a) Transition rules for decrementing. (b) One iteration of the counter. (c) Transition rules for the additive states. The k state can be any digit. (d) The additive state moving to the left of the assembly via transitions.

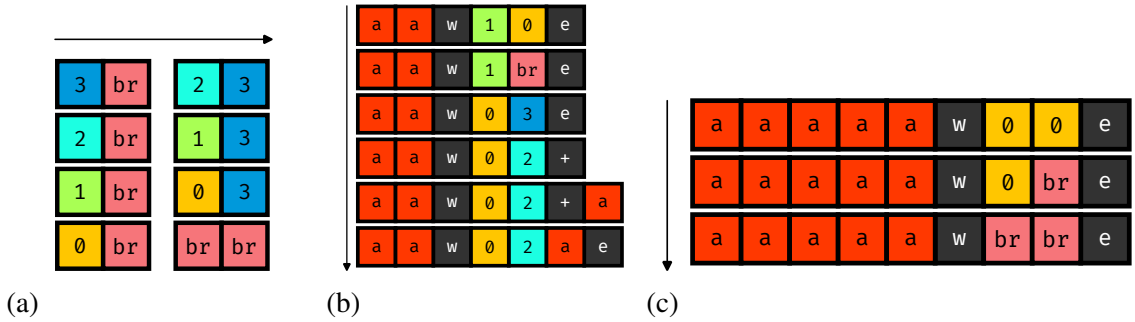


Figure 3.14: Example methods using a base-3 number. (a) Transition rules for borrowing. (b) One iteration of the counter with borrowing. (c) The borrow states reaching w, stopping the assembly.

3.2.25 Constant Height Rectangles

In this section, we investigate bounded-height rectangles. We introduce a constant-height counter by first turning the string construction 90 degrees and adding some modifications. For clarity, we change the north and south caps into west and east caps, respectively. We then add states to count down from the number on the string, and growing the assembly every time the number is decremented. These states include an additive state a, a borrow state br and the attachment state $+$.

The constant height counter takes inspiration from long subtraction. The counter begins with trying to subtract 1 from the first place value. An example of this can be seen in Figure 3.13. If subtraction is possible, the number is decremented and the east tile e transitions into the attachment state $+$ to allow an additive tile to attach. Once attached, the two states $(+,a)$ transition into (a,e), completing one iteration of the counter. The additive state will eventually make its way to the left side of the assembly as the next iteration begins. An example of this is shown in Figure 3.13. Lastly,

if subtraction from the first place value is not possible, meaning the value of the digit is 0, then the system attempts to borrow from the higher place values. The rules for this, as well as an example, are shown in Figure 3.14. If the borrow state, br, reaches the west tile w, then the number is 0 and the counter is finished. The assembly is complete once the counter has finished and the additive states have all made it to the left side of the west tile w. This is shown in Figure 3.14c. The counter can work with any base b by modifying the transition rules and adding $\mathscr{O}(b)$ states. This counter is not freezing.

Theorem. For all $n > 0$, there exists a Tile Automata system that uniquely assembles an $4 \times n$ rectangle with nondeterministic transition rules and $O(\log^{\frac{1}{4}} n)$ states.

Proof. To uniquely assemble a $4 \times n$ rectangle, we construct a constant height counter as described above. By Theorem 3.2.11 we construct a binary string with $O(\log^{\frac{1}{4}} n)$ states. The string is k, written in binary, such that $k = n - s$ where s is the number of tiles needed to assemble the string. Once the string has been assembled, the constant height counter can begin counting down from k and attaching tiles. Once the string reaches 0, k tiles have been added to the assembly. The assembly is now $s + k$ tiles long for a total of n. □

3.2.26 Single-Transition Rules

The constant height counter is easily modified to only use single-transition rules. For every nonsingle-transition rule δ, add one additional state and replace δ with 3 single-transition rules. For example, given a rule $\delta = (\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, d)$. We add another state ω and three rules as follows.

- $\delta_1 = (\mathsf{A}, \mathsf{B}, \mathsf{A}, \omega, d)$
- $\delta_2 = (\mathsf{A}, \omega, \mathsf{C}, \omega, d)$
- $\delta_3 = (\mathsf{C}, \omega, \mathsf{C}, \mathsf{D}, d)$

Theorem. For all $n > 0$, there exists a Tile Automata system that uniquely assembles a $3 \times n$ rectangle with single-transition rules and $O(\log^{\frac{1}{3}} n)$ states.

Proof. We take the constant height counter and make all rules single-transition using the process described above. Then, by Theorem 3.2.10, we construct a binary string with $O(\log^{\frac{1}{3}} n)$ states. □

3.2.27 Deterministic rules

Theorem. For all $n > 0$, there exists a deterministic Tile Automata system that uniquely assembles a $2 \times n$ rectangle with single-transition rules and $\Theta\left((\frac{\log n}{\log\log n})^{\frac{1}{2}}\right)$ states.

Proof. There are two cases, and we use the same ideas as in the proof of Theorem 3.2.13. Case 1 is the same as in Theorem 3.2.13.

Case 2: $n > 2^{16}$. Let $b = \lceil(\frac{\log n}{\log\log n})^{\frac{1}{2}}\rceil$, and $d = 4\lceil\frac{\log n}{\log\log n}\rceil$. By Corollary 3.2.1, a base-b d-digit string can be assembled with $\mathcal{O}(d^{\frac{1}{2}} + b)$ states and single-transition rules at a size of $(n+2) \times 2$. The initial string will be the value $b^d - n$ represented in base b using $\Theta\left((\frac{\log n}{\log\log n})^{\frac{1}{2}}\right)$ states. Since we have an arbitrary base, we need to add $\Theta(b)$ states for the counter to support base b. Therefore, for all $n > 0$, there exists a deterministic Tile Automata system that uniquely assembles a $2 \times n$ rectangle with $\Theta(b) + \Theta\left(d^{\frac{1}{2}}\right) + c = \Theta\left((\frac{\log n}{\log\log n})^{\frac{1}{2}}\right)$ states, where c is a constant from Case 1. □

3.2.28 Deterministic $1 \times n$ lines

By using Theorem 3.2.9 to assemble the initial string of the counter, we achieve $1 \times n$ lines with double transition rules.

Theorem. For all $n > 0$, there exists a deterministic Tile Automata system that uniquely assembles a $1 \times n$ rectangle with $\Theta\left((\frac{\log n}{\log\log n})^{\frac{1}{2}}\right)$ states.

Proof. As with the previous proof, let $b = \lceil(\frac{\log n}{\log\log n})^{\frac{1}{2}}\rceil$, and $d = 4\lceil\frac{\log n}{\log\log n}\rceil$. By Theorem 3.2.9, a base-b d-digit string can be assembled with $\mathcal{O}(d^{\frac{1}{2}} + b)$ states at exact scale. $\Theta(b)$ states are added for the counter to support the arbitrary base b. Therefore, for all $n > 0$, there exists a deterministic Tile Automata system that uniquely assembles a $1 \times n$ rectangle with $\Theta(b) + \Theta\left(d^{\frac{1}{2}}\right) + c = \Theta\left((\frac{\log n}{\log\log n})^{\frac{1}{2}}\right)$ states, where c is a constant from Case 1. □

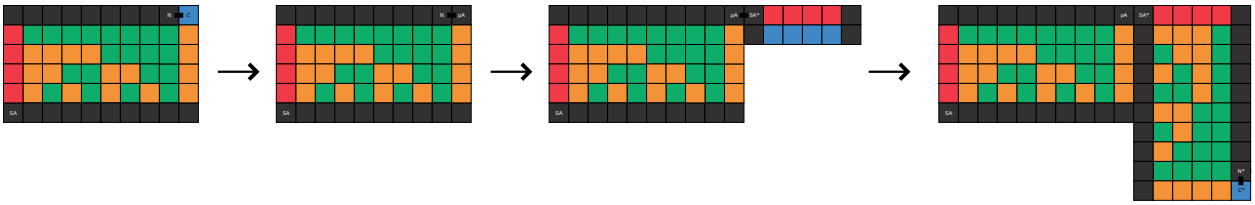


Figure 3.15: The transitions that take place after the first rectangle is built. The carry state transitions to a new state that allows a seed row for the second rectangle to begin growth

3.2.29 Squares

In this section, we utilize the rectangle constructions to build $n \times n$ squares using an optimal number of states.

Let $n' = n - 4\lceil \frac{\log n}{\log\log n} \rceil - 2$, and Γ_0 be a deterministic Tile Automata system that builds an $n' \times (4\lceil \frac{\log n}{\log\log n} \rceil + 2)$ rectangle using the process described in Theorem 3.2.13. Let Γ_1 be a copy of Γ_0 with the affinity and transition rules rotated 90 degrees clockwise, and the state labels appended with the symbol "*1". This system has distinct states from Γ_0, and builds an equivalent rectangle rotated 90 degrees clockwise. We create two more copies of Γ_0 (Γ_2 and Γ_3), and rotate them 180 and 270 degrees, respectively. We append the state labels of Γ_2 and Γ_3 similarly.

We utilize the four systems described above to build a hollow border consisting of the four rectangles, and then adding additional initial states that fill in this border, creating the $n \times n$ square.

We create Γ_n, starting with system Γ_0, and adding all the states, initial states, affinity rules, and transition rules from the other systems ($\Gamma_1, \Gamma_2, \Gamma_3$). The seed states of the other systems are added as initial states to Γ_n. We add a constant number of additional states and transition rules so that the completion of one rectangle allows for the "seeding" of the next.

To Γ_n, we add transition rules so that when the first rectangle (originally built by Γ_0) has built to its final width, a tile on the rightmost column of the rectangle transitions to a new state pA. pA has affinity with the state $S_A * 1$, which originally was the seed state of Γ_1. This allows state $S_A * 1$ to attach to the right side of the rectangle, thus "seeding" Γ_1 and allowing the next rectangle to assemble (Figure 3.15). The same technique is used to seed Γ_2 and Γ_3.

When the construction of the final rectangle (of Γ_3) completes, transition rules propagate a

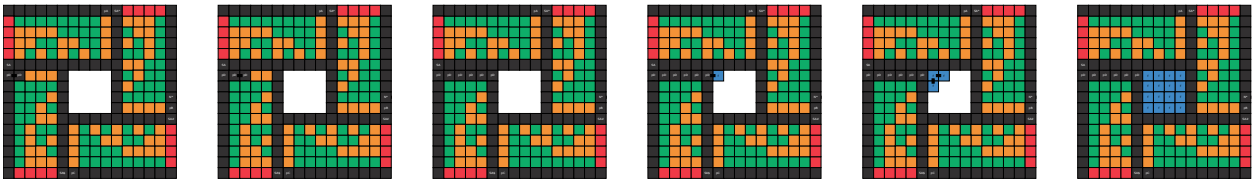


Figure 3.16: Once all four sides of the square finish being built, the *pD* state propagates to the center and allows the light blue tiles to fill in the square

state *pD* towards the center of the square (Figure 3.16). Additionally, we add an initial state *r* that has affinity with itself in every orientation, as well as with state *pD* on its west side. This allows the center of the square to be filled with tiles.

Theorem. For all $n > 0$, there exists a Tile Automata system that uniquely assembles an $n \times n$ square with,

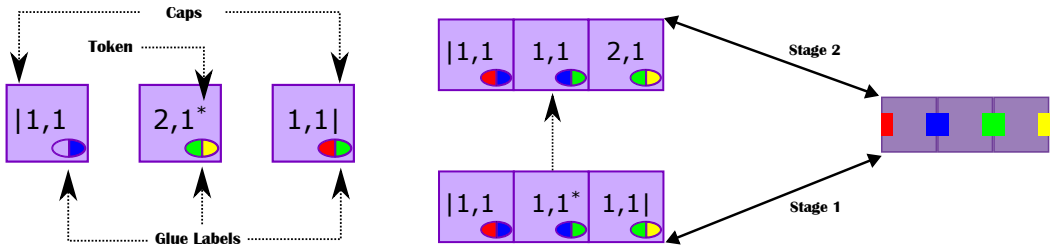
- Deterministic transition rules and $\Theta\left(\left(\frac{\log n}{\log\log n}\right)^{\frac{1}{2}}\right)$ states.
- Single-Transition rules and $\Theta(\log^{\frac{1}{3}} n)$ states.
- Nondeterministic transition rules and $\Theta(\log^{\frac{1}{4}} n)$ states.

3.3 Simulation of Multiple Stages in Single Bin Active Tile Self-Assembly

Two significant and often competing goals within the field of self-assembly are minimizing tile types and minimizing human-mediated experimental operations. The introduction of the Staged Assembly and Single Staged Assembly models, while successful in the former aim, necessitate an increase in mixing operations later. We investigate building optimal lines as a standard benchmark shape and building primitive. We show that a restricted version of the 1D Staged Assembly Model can be simulated by the 1D Freezing Tile Automata model with the added benefits of the complete automation of stages and completion in a single bin while maintaining bin parallelism and a competitive number of states for lines, patterned lines, and context-free grammars.

3.3.1 Simulation of General 1D Staged

In this section, we show how to simulate all 1D staged systems with TA systems. First, we define what simulate means for this system, followed by a high-level overview of our simulation,



(a) Tiles (b) Assemblies

Figure 3.17: (a) Each of our Tile Automata states conceptually represents two glue labels that say which tile type they map to (a glue may be null, as in the leftmost state). They may also contain features such as the left/right cap or the active state token. (b) Assemblies map based on the glue labels on the Tile Automata states. Multiple Tile Automata assemblies represent the same Staged assembly, but sometimes in different stages.

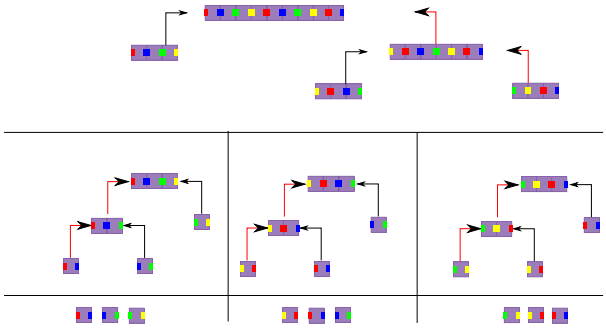
and then the details.

3.3.2 Simulation

Here, we utilize a simplified definition of simulation in which the set of final terminal assemblies, from the *target* staged system to be simulated, is exactly the same, under a mapping function, as the final terminal assemblies of the *source* TA system that is simulating it. This is a standard type of simulation used, and we omit technical definitions in this version. A stronger definition of simulation incorporates *dynamics*, in which assemblies may attach in the target system if and only if they attach in the source system. However, our approach focuses on simulating a restricted set of dynamics that are sufficient to ensure the production of all final (and partial) assemblies. We leave the problem of fully simulating the dynamics of a staged system as future work.

3.3.3 Overview

We create a Tile Automata system with initial tiles representing the initial tile types of the staged system. Each assembly in our Tile Automata system represents an assembly in a specific stage and bin. Each state is a pair consisting of a tile type t and a stage-bin label representing t in that specific stage and bin. Some states will have an *active state token*(*) used to track the progress of the Tile Automata assembly in the assembly tree. We simulate only left- or right-handed



(a) Staged System

(s, b)	Red	Blue	Green	Yellow
$(1,1)$	Term	Used	Used	Term
$(1,2)$	Used	Used	Term	Term
$(1,3)$	Used	Term	Term	Used
$(2,1)$	Term	Term	Used	Used

(b) Glue-Terminal Table

Figure 3.18: (a) Example Staged system to be simulated. (b) Glue-Terminal Table for shown staged system. In the table, s is the stage and b is the bin.

assembly trees based on the parity of the stage number. The logic for the transition rules is described in the full paper using a *Glue-Terminal Table*. Each Tile Automata assembly builds according to the assembly trees of the staged system by having the token "read" the glues to decide if an assembly is terminal in a bin and needs to transition to the next stage.

3.3.4 Glue-Terminal Table

For the simulation to work, we need to know the glues used in each bin of the target system because we cannot "read" the absence of a glue/assembly in self-assembly. However, we can use the Glue-Terminal Table to construct the transition rules. This table stores which glues correspond with each bin.

Definition (Glue-Terminal Table). For a staged system $\Upsilon = (M_{r,b}, T)$, the Glue-Terminal table $GT((s,b),g)$ is a binary $|M_{r,b}| \times G$ table with rows labeled with stage-bin pairs and columns labeled with glues. The entry $GT((s,b),g)$ is true (Used) if there exists at least two producible assemblies in bin b that attach using glue g in stage s. If it is false (Term.), the glue is never used in bin b for stage s.

3.3.5 States and Initial Tiles

A state in our Tile Automata system has the following properties: each state has the first two properties and the second two properties are optional. The first label has sb possible options,

the second has t, and the rest only increase the state space by a constant factor. This results in an upper bound on the states used of $O(sbt)$.

- **Stage-Bin Label.** Each state $(s,i)_t$ is labeled with a pair of integers (s,i) saying the state represents the i^{th} bin in stage s.

- **Glue Labels.** Each state $(s,i)_t$ represents a tile t from the staged system. We say this state has the glue labels of t when defining our affinity rules in Tile Automata. This label also defines our mapping from TA states to staged tiles in both directions.

- **Active State Token.** A state $(s,i)_t^*$ may have an Active State Token *. The token is used to enforce the left/right handed assembly trees by starting on one side of an assembly, and allowing attachment to other states with matching glue and stage-bin labels.

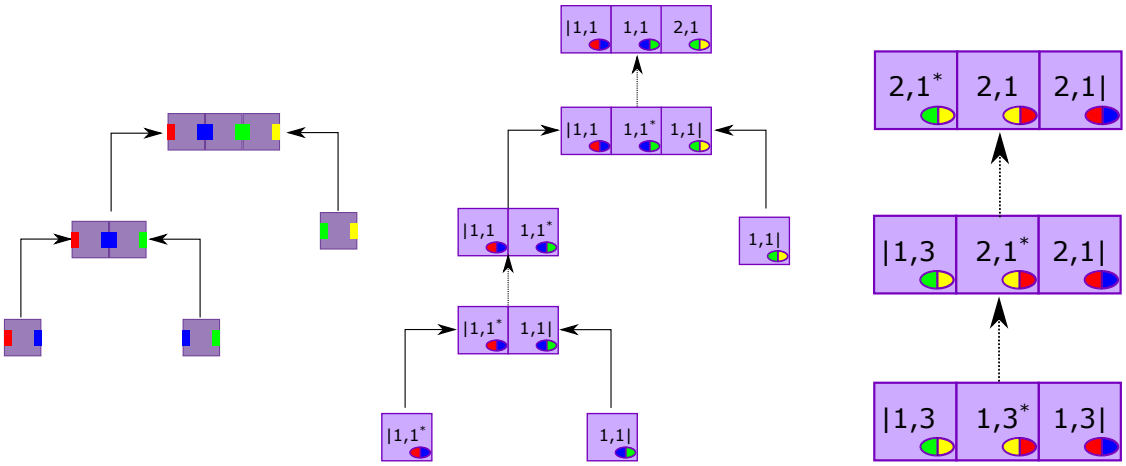
- **Caps.** A state may have a cap on one side, denoted $|s,i)_t$ or $(s,i|_t$. This means that on the side of the cap $|$, there are no affinity rules for that state. Until an assembly is ready to attach, it will have caps on its left and right most tiles.

We create an initial state for each pair $b_{1,i},t$ where $b_{1,i}$ is the i^{th} bin of the first stage and t is a tile input to that bin. If the left glue of the t is used in the $b_{1,i}$, then we include the state $(1,i_t|$, i.e., the right cap state. If the left glue is open, but the right glue is used, the tile is the first in a left-handed assembly tree. In this case, we include the token left cap state $|1,i_t^*)$.

If a tile is terminal in the first bin, we instead include an initial state representing the first bin where the state is consumed. For example, if a tile t is input to bin $(1,i)$ and is terminal, but its right glue is used in an attachment in bin $(2,j)$ (where there's an edge between $(1,i)$ and $(2,j)$), then we instead include an initial state $|2,j_t)$.

3.3.6 Bin Simulation

In any odd stage, we construct every terminal using a sequence of attachments representing a left-handed assembly tree. For even stages, we use a right-handed assembly tree. We control this



(a) $(1,1)$ (b) $(1,3)$

Figure 3.19: (a) Example simulation of an assembly in stage 1. Notice the token moves leftward through the assembly as it builds to enforce a left handed assembly tree. (b) Transition for terminal assembly in bin $(1,3)$. Since the rightmost glue is terminal in bin $(1,3)$ the token changes the stage to 2 and starts moving left to remove the cap.

with the token by defining our affinity rules such that every attachment occurs between one state with the token and one without a cap.

We walk through an example of a bin in the first stage in Figure 3.19a. The token left cap state $|1,1_t^*)$ attaches to the right cap state $(1,1_{t'}|$ if t' attaches to the right of t. These two states then transition. If the right glue of t' is used in the bin, the token moves to that state and removes the cap. This process can then repeat in the bin. Looking at the next tile t'', the right glue is unused, and thus, the assembly is terminal, and the transition should move it to the next stage, now changing directions as outlined in Figure 3.19b. Transition rules are defined in more detail in the full paper. Note that this algorithm is non-deterministic as one bin may output to multiple bins in the next stage, so a pair of states may have multiple transition rules.

Theorem. For any 1D staged system Υ with s stages, b bins, and t tile types, there exists a 1D Freezing Affinity-Strengthening Tile Automata system Γ with $O(sbt)$ states that simulates Υ.

Proof. Consider a staged system $\Upsilon = (M_{r,b}, T)$ with s stages, b bins and t tiles types. Tile Automata system $\Gamma = (\Sigma, \Pi, \Delta)$ which simulates Υ is defined and discussed below.

State complexity $O(sbt)$. Each tile type in Υ requires a unique state in Γ for every bin in

every stage, resulting in $s \cdot b \cdot t$ states. The additional state increase for the token and caps of each state is constant for a total of $O(sbt)$ states.

Freezing and Affinity Strengthening. A state $\sigma_t \in \Sigma$ with tile type $t \in T$ has affinity with a state $\sigma'_t \in \Sigma$ with tile type $t' \in T$ if t attaches to t' in Υ. For every transition rule $\delta \in \Delta$, δ does not alter the tile type a state represents since only the stage, bin, token, or cap are affected.

Every transition rule is freezing and either removes a cap, moves the token forward, or advances to the next stage. Once a state with a tile type t has lost its cap it can never regain it. In a single stage, the token may walk over each tile a maximum of 2 times as both sides of the assembly must be checked to decide if the assembly is terminal. Note that this token walk involves adding an additional distinct state so the tiles do not visit the same state twice.

Simulation. We prove this is a correct simulation by induction on the size of the assemblies. The initial assemblies cover our base case for single tiles in Λ. The tile input in the first stage in Υ ensures each included assembly is in Λ. For the recursive case, assume every assembly $A \in \mathtt{PROD}_\Upsilon$ with $|A| < x$ is simulated. Let b be the bin in which A is produced. A must be produced using two assemblies B and C, each of size $< x$, which are also in bin b. From our assumption, B and C have assemblies representing them- $B', C' \in \mathtt{PROD}_\Gamma(\Lambda)$. Since B and C are produced in the same bin and have matching assemblies B' and C' with matching tokens, they may combine into an assembly A'. A will represent A since it has the same labels.

□

3.3.7 Lines

Using Theorem 3.3.1, we provide an alternate proof from (Caballero et al. 2020) of length-n lines with $O(\log n)$ states.

Corollary. For all $n \in \mathbb{N}$, there exists a freezing Tile Automata system that uniquely assembles a $1 \times n$ line in $O(\log n)$ states.

Proof. In (Erik D Demaine, M. L. Demaine, et al. 2008), it is shown that there exists a staged assembly system that uniquely produces a $1 \times n$ line with 6 tile types, 7 bins, and $O(\log n)$ stages.

From theorem 3.3.1, there exists a Freezing Affinity-Strengthening Tile Automata system Γ with $O(sbt)$ states that simulates any staged system Υ with s stages, b bins and t tile types. Therefore, simulating the staged assembly system from (Erik D Demaine, M. L. Demaine, et al. 2008) can be done with $O(\log n)$ states. □

3.3.8 Patterns

In this section, we augment the Tile Automata model with the concept of a tile's color being based on the current state. For a set of color labels C, this is a partition of the states into $|C|$ sets. We only consider constant-sized C. Thus, the *color* of a tile t is the partition of the tile's state, denoted as $c(t)$.

Definition (Pattern). A pattern P over a set of colors C is a partial mapping of $\mathbb{Z}$ to elements in C. Let $P(z)$ be the color at $z \in \mathbb{Z}$. A scaled pattern P^{hw} is a pattern replacing each pixel within a $1 \times w$ line of pixels.

Definition (Patterned Assemblies). We say a positioned assembly A' represents a pattern P if for each tile $t \in A'$, $c(t) = P(L(t))$ and $dom(A') = dom(P)$. We say a positioned assembly B' represents a pattern P at scale $h \times w$ if it represents the scaled pattern P^{hw}.

A system Γ uniquely assembles a pattern P if it uniquely assembles an assembly A, such that A contains a positioned assembly that represents P.

Definition (Color-Locked). A Tile Automata system is Color-Locked if for every transition rule $\delta = (S_{1a}, S_{2a}, S_{1b}, S_{2b}, d) \in \Delta$, $c(S_{1a}) = c(S_{1b})$ and $c(S_{2a}) = c(S_{2b})$, i.e., tiles are not allowed to change their color.

3.3.9 Context-Free Grammars

A **context-free grammar (CFG)** is a set of recursive rules used to generate patterns of strings that define a given language. A CFG is a quadruple $G = (V, \Upsilon, R, S)$ where V is a finite set of nonterminal symbols (variables), Υ is a finite set of terminal symbols, R is the set of production rules, and $S \in V$ is the start symbol. Assuming the CFG is in Chomsky Normal Form (CNF), the

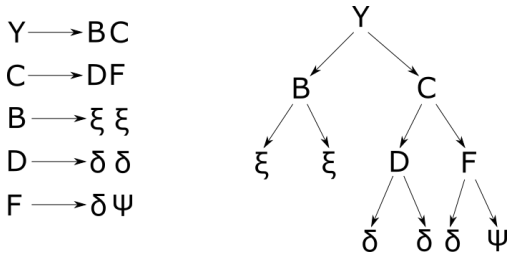


Figure 3.20: A restricted context-free grammar (RCFG) G and its corresponding parse tree that produces a pattern P, $\xi\xi\delta\delta\delta\psi$. This is a deterministic grammar, producing only pattern P.

production rules of the CFGs are in the form $A \to BC$ or $A \to a$, where $A, B, C \in V$ and $a \in \Upsilon$. A CFG derives a string by recursively replacing nonterminal symbols with terminal and nonterminal symbols based on its production rules.

Definition (Minimum Context Free Grammars). We define the size of a grammar G as the number of symbols in the right-hand side rules. Let CF_P be the size of the smallest CNF CFG that produces the singleton language $|P|$.

In this work, we focus on the CFG class used in (Erik D. Demaine et al. 2011), which they name Restricted CFGs. These restricted grammars produce a singleton language, $|L(G)| = 1$, and thus are deterministic. This is the same concept of Context-Free Straight Line grammars from (Benz and Kötzing 2013). Note a Restricted CFG is not necessarily in CNF but any RCFG can be transformed into CNF with only a constant factor increase in rule size. Figure 3.20 presents an example RCFG G and its parse tree that derives a pattern of symbols P, $\xi\xi\delta\delta\delta\psi$. The parse tree shows how internal nodes are nonterminal symbols, and leaf nodes contain a terminal symbol whose in-order traversal derives the output string. Notice that since RCFG G is deterministic, each nonterminal symbol $N \in V$ has a unique subpattern $g(N)$ that is defined by taking N as the start symbol S and applying the production rules. Here, the language or output pattern P of G can be denoted by $L(G) = g(S)$.

3.3.10 Tile Automata Upper Bounds

In (Erik D. Demaine et al. 2011), the authors define the size of a staged assembly system Υ (denoted $|\Upsilon|$) to be the number of edges in its mix graph.

Corollary. For any pattern P, there exists a Freezing Tile Automata system Γ that uniquely assembles P with $O(CF_P)$ states and 1×1 scale. This system is cycle-free, and transition rules do not change the color of tiles.

Proof. In (Erik D. Demaine et al. 2011), the authors show that, given a RCFG G deriving a pattern P, there exists a 1D SSAS Υ that assembles pattern P with B total bins, t tile types, and at 1×1 scale with $|\Upsilon| = O(CF_P)$. Theorem 3.3.1 gives an upper bound based on the number of stages times the number of bins. However, the construction also gives an upper bound of $O(Bt)$ as each state stores the bin and tile it maps to. It follows that there exists a Freezing TA system Γ that uniquely assembles P with $O(CF_P)$ states and 1×1 scale if Γ simulates Υ.

□

CHAPTER IV

CHEMICAL REACTION NETWORKS

Chemical reaction networks aim to be an abstraction of real-world chemical reactions. The idea is to have abstract chemical species float around in a well-mixed solution and compute based on predefined reactions. Our full manuscript can be found in (Alaniz et al. 2022).

4.1 Model Definitions

4.1.1 Basics

Let $\Lambda = s_1, s_2, \ldots, s_{|\Lambda|}$ denote some ordered alphabet of *species*. A configuration over Λ is a length-$|\Lambda|$ vector of non-negative integers, denoting the number of copies of each present species. A *rule* or *reaction* has two multisets, the first containing one or more *reactant* (species) used to create the resulting *product* (species), the second multiset. We represent each rule as an ordered pair of configuration vectors $R = (R_r, R_p)$. R_r contains the minimum counts of each reactant species necessary for reaction R to occur, where reactant species are either *consumed* by the rule in some count or leveraged as *catalysts* (not consumed); in some cases a combination of the two. The product vector R_p has the count of each species *produced* by the *application* of rule R, effectively replacing vector R_r. The species corresponding to the non-zero elements of R_r and R_p are termed *reactants* and *products* of R, respectively.

The *application* vector of R is $R_a = R_p - R_r$, which shows the net change in species counts after applying rule R once. For a configuration C and rule R, we say R is applicable to C if $C[i] \geq R_r[i]$ for all $1 \leq i \leq |\Lambda|$, and we define the *application* of R to C as the configuration $C' = C + R_a$. For a set of rules Γ, a configuration C, and rule $R \in \Gamma$ applicable to C that produces $C' = C + R_a$, we say $C \rightarrow^1_\Gamma C'$, a relation denoting that C can transition to C' by way of a single rule application from

Γ. We further use notation $C \rightsquigarrow_\Gamma C'$ to signify the transitive closure of $\rightarrow_\Gamma^1$ and say C' is *reachable* from C under Γ, i.e., C' can be reached by applying a sequence of applicable rules from Γ to initial configuration C. We use the following notation to depict a rule $R = (R_r, R_p)$:

$$\sum_{i=1}^{|\Lambda|} R_r[i]s_i \rightarrow \sum_{i=1}^{|\Lambda|} R_p[i]s_i$$

For example, a rule turning two copies of species H and one copy of species O into one copy of species W would be written as $2H + O \rightarrow W$.

Definition (Discrete Chemical Reaction Networks). A discrete chemical reaction network (CRN) is an ordered pair (Λ, Γ) where Λ is an ordered alphabet of species, and Γ is a set of rules over Λ.

The primary computational problem we consider in this paper is the *reachability* problem. We consider additional problems in the paper, such as determining if it is possible to produce a given amount of a particular species from an initial configuration of a CRN, as well as universal reachability, which asks if the target configuration is reachable for all reaction application sequences.

Definition (Reachability Problem). Given a CRN (Λ, Γ), an initial configuration I, and a destination configuration D, the *Reachability Problem* is to compute whether or not D is reachable from I with respect to Γ.

Definition (Production Problem). Given a CRN (Λ, Γ), an initial configuration A, a species $s_i \in \Lambda$, and a positive integer k, decide if there exists a reachable configuration B such that $B[i] \geq k$.

Definition (Universal Reachability Problem). Given a CRN (Λ, Γ), an initial configuration I, and a destination configuration D, the *Universal Reachability Problem* is to compute whether or not D is reachable from all configurations M that are reachable from I with respect to Γ.

4.1.2 Primary Restrictions

We consider the reachability problem under several different restrictions defined below.

Definition (Feed-Forward). A CRN (Λ, Γ) is *feed-forward* if Γ permits an ordering on the rules such that the products of any given rule never occur as reactants for earlier rules of the ordering.

Each rule in a system produces some species and consumes others. The following metric places a maximum bound on the number of rules that either produce a given species (j-source) or consume a given species (j-consuming).

Definition (j-source, j-consuming). A species s_i is *consumed* in rule $R = (R_r, R_p)$ if $R_r[i] > R_p[i]$, *produced* if $R_r[i] < R_p[i]$, and is a *catalyst* in rule R if $R_r[i] = R_p[i] > 0$.

A CRN (Λ, Γ) is j-source if for all species $s \in \Lambda$, s is produced in at most j distinct rules in Γ. A CRN (Λ, Γ) is j-consuming if for all species $s \in \Lambda$, s is consumed in at most j distinct rules in Γ. We use the terms *single-source* and *single-consuming* for the special cases of 1-source and 1-consuming CRNs, respectively.

The next concept is a special class of rules that either produce nothing (void rules) or consume nothing (autogenesis rules). This could potentially be motivated by evaporation, the ability to pass through a membrane or the spontaneous appearance of ions in a vacuum.

Definition (Void and Autogenesis rules). A rule $R = (R_r, R_p)$ is a *void* rule if $R_a = R_p - R_r$ has no positive entries. A rule is an *autogenesis* rule if R_a has no negative values.

4.1.3 Additional Restrictions

We also consider the complexity of reachability and production with respect to the *size* of rules.

Definition (Size/Volume). The *size/volume* of a configuration vector C is $\texttt{volume}(C) = \sum C[i]$.

Definition (size-(i, j) rules). A rule $R = (R_r, R_p)$ is said to be a size-(i, j) rule if $(i, j) = (\texttt{volume}(R_r), \texttt{volume}(R_p))$. A reaction is bimolecular if $i = 2$ and unimolecular if $i = 1$.

Definition (Volume Decreasing, Increasing, Preserving). A rule $R = (R_r, R_p)$ of size-(i, j) is said to be volume decreasing if $i > j$, volume increasing if $i < j$, and volume neutral if $i = j$.

A CRN (Λ, Γ) is said to be volume decreasing (respectively increasing, preserving) if all rules in Γ are volume decreasing (respectively increasing, preserving). Note: In previous work, volume preserving has been called *Proper*.

A special subset of CRN systems studied in the literature is Population Protocols, in which agents bump into each other and adjust their state according to rules. This model is equivalent to a CRN that is limited to exactly volume 2 for both the reactants (the two agents that bump into each other) and the products (representing the two new states of the agents after the collision).

Definition (Population Protocols). A CRN (Λ, Γ) in which all rules in Γ are size-$(2,2)$ is called a population protocol.

4.2 Reachability in General CRNs

The main result of this section is PSPACE-completeness of the reachability problem with 2-consuming, 2-source, size $(2,2)$ reactions in Theorem 4.2.2. En route, we prove that production is PSPACE-complete with Theorem 4.2.1. We extend this reduction in two ways. First, in Corollary 4.2.1, we prove our reduction holds for the universal reachability problem, and second, we show the reduction holds for non-monotonic volume with rule sizes of $(1,2)$ and $(2,1)$ in Theorem 4.2.2. We follow with several interesting, yet minor results.

4.2.1 Gadget Reconfiguration Framework

Our main result is based on the motion planning problem through Toggle-Lock and Rotate gadgets (Erik D Demaine, Hearn, et al. 2022). Motion planning is PSPACE-hard with only a rotate gadget and any reversible gadget with interacting tunnels, a class that includes the Toggle-Lock (Erik D Demaine, Grosof, et al. 2018). The motion planning problem considers two input configurations of gadgets, a start location for the agent, and a target location, and asks if the agent can reach the target location. [1]

A gadget consists of a set of labeled ports and states describing the gadget's legal traversals, which may change the state of the gadget. The Toggle-Lock Gadget has an unlocked and locked

[1] In (Erik D Demaine, Hearn, et al. 2022) this problem is called the reachability problem. To avoid confusion, we refer to this as the motion planning problem.

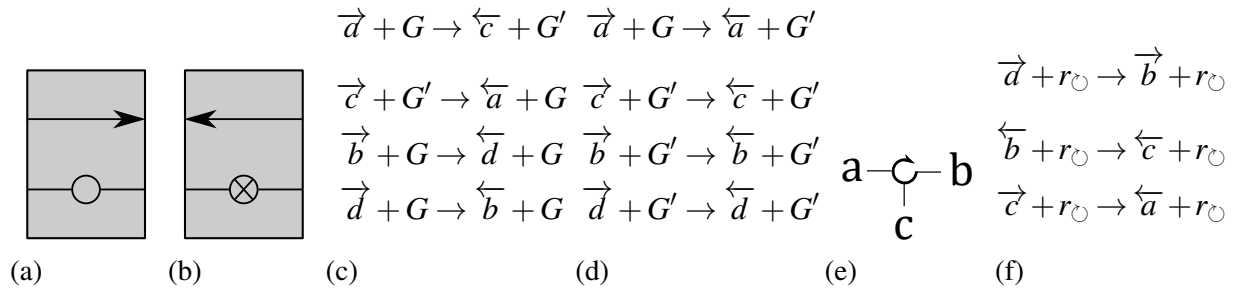


Figure 4.1: (a) Unlocked state of a Toggle-Lock gadget. (b) Locked state of a Toggle-Lock gadget. (c-d) Reactions which implement a single gadget. (c) represents a successful traversal and (d) represents the 'bound-back' reactions. The arrow is incoming or outgoing from port. (e) The rotate gadget. (f) Rules for the rotate gadget.

state, shown in Figure 4.1a and 4.1b, respectively. The top path is directed, and the agent must follow the direction. Traversing this path changes the state of the gadget. Traversal of the bottom tunnel can only occur in the unlocked state; this can be done in either direction and does not change the state of the gadget. The rotate gadget only has one state that sends the agent to the next port going clockwise. A motion planning system consists of a set of gadgets, a set of wires denoting port connections, and an initial signal location.

We are simulating a 0-player gadget framework where the agent makes no choices. The agent is directed down the wire and turns around if the gadget is not traversable in the current state, corresponding to a deterministic model of computation. An early version of this result appeared in the short abstract (Fu et al. 2022), which reduced from a different gadget and used the 1-player framework studied in (Erik D Demaine, Grosof, et al. 2018; Erik D Demaine, Hendrickson, and Lynch 2020; Ani et al. 2022). However, the reduction was not constant source or constant consuming.

Two ports are connected by a wire. We label each wire with a symbol such as a and include two species $\overrightarrow{a}, \overleftarrow{a}$ to denote the direction moved along the wire. We refer to these as the agent species.

We have two species for each Toggle-Lock that we call the *gate catalysts*. We represent the unlocked gate using the species G, and the locked state with species G'. Traversing the toggle

tunnel requires the gadget to be in the correct state for the toggle. The reaction changes the state of the gate catalyst and the agent species. The bottom tunnel can only be traversed if the gate is in the unlocked state. However, this does not change the state of the gate, so the G species acts a catalyst that must be present to traverse the gadget. Figure 4.1c shows the rules used to implement a toggle lock.

We implement rotate gadgets with a single rotate clockwise species $r_{\circlearrowright}$. The rotate gadget diagram is shown in Figure 4.1e. The signal state changes to the outgoing direction of the next wire in the clockwise ordering of ports using the rules shown in Figure 4.1f. Note that each reaction consumes and creates exactly one agent species yielding the following observation:

Observation. Any reachable configuration in the reduction only contains a single agent species.

4.2.2 Production

We prove production is PSPACE-complete using the framework described above. The target species is the agent species representing the target wire.

Theorem. Production in 2-source, 2-consuming preserving CRNs is PSPACE-complete with only bimolecular reactions.

Proof. Membership in PSPACE for Proper CRNs was shown in (Thachuk and Condon 2012). Given an instance of the motion planning problem with toggle-locks and rotate gadgets, create a CRN ruleset and configuration as described above. Our starting configuration of the CRN encodes the start location and starting states of the gadgets. The target species we wish to produce is the agent species for the target location in the motion planning problem.

The agent may only traverse a toggle-lock gadget if the gate catalyst is in the correct state. The rotate gadget sends the signal to the correct next state. Once the agent reaches the target wire, the agent species representing it is produced. If the agent reaches the target location, then reactions apply representing the path of the agent to produce the target species.

From Observation 4.2.1, there only exists one agent species and the reactions encode only valid traversals through the gadgets. If the target species is produced, then the sequence of reactions to produce it represents the path of the agent though the system of gadgets. Each gate catalyst is only produced and consumed in the reaction that implements the toggle tunnel. Each agent species is directed, so it also is only produced and consumed in one rule.

Note that if an agent ever attempts to cross a gadget in a state that does not allow that traversal, the configuration will no longer have any valid reactions. This is the equivalent to an illegal move in the motion planning problem and the agent cannot progress, so the game is over. □

4.2.3 Reachability

We extend the reduction above to work for reachability by taking advantage of the fact that the toggle-lock is reversible. Once we reach the target species we flip the rotate catalyst to allow the agent to move counterclockwise through a rotate gadget, allowing us to undo all the changes to the gadget states. When the agent reaches the starting location again, the system is in the initial configuration except with the opposite rotate catalyst. The reconfiguration problem for the 1-player version of the motion planning framework was shown to be PSPACE-complete using a similar technique where the agent changes the state of the final gadget then undoes all of its previous movements (Ani et al. 2022).

Theorem. Reachability in 2-source, 2-consuming preserving CRNs is PSPACE-complete with only bimolecular reactions.

Proof. The reduction from Theorem 4.2.1 can be extended to show the reachability problem is PSPACE-hard as well. We add an additional species $r_{\circlearrowleft}$ which is the rotate counterclockwise catalyst. The target configuration is the same as the initial configuration except with the counterclockwise catalyst $r_{\circlearrowleft}$. If the target wire is a, we add the rule $\overrightarrow{a} + r_{\circlearrowleft} \to \overleftarrow{a} + r_{\circlearrowleft}$, which changes the direction of the rotate catalyst and turns the agent around on the wire. Since the toggle lock gadget is reversible, the agent will undo all of its moves and return to the start configuration. Since the gadget system has the property of being reversible, this backwards traversal is possible. □

This reduction extends to the universal reachability problem since there is only a single reaction at each step that can be performed.

Corollary. Universal reachability in 2-source, 2-consuming preserving CRNs is PSPACE-complete with only bimolecular reactions.

Proof. From Observation 4.2.1, we know there only exists one agent state in the system at a time. Since the system is single-consuming, the agent state is only consumed in a single rule. These two points mean there only exists a single move sequence, so if the target configuration is reachable, it is universally reachable. □

4.2.4 Non-Monotone Volume

In this section we extend the reduction to utilize smaller rules. We show PSPACE-hardness of production and reachability when allowing both $(1,2)$ and $(2,1)$ rules. The CRN has both volume increasing and decreasing rules, which means it is non-monotone, and thus, these problems are not known to be in PSPACE. To prove this, we add an intermediate species for each reaction. We replace the reaction $\overrightarrow{a} + G \to \overrightarrow{c} + G'$ with the two reactions $\overrightarrow{a} + G \to \overrightarrow{aGc}$ and $\overrightarrow{aGc} \to \overrightarrow{c} + G'$.

Corollary. Production and reachability in 2-source, 2-consuming CRNs is PSPACE-hard with rules of size $(2,1)$ and $(1,2)$.

Proof. The reduction behaves the same way as in Theorems 4.2.1 and 4.2.2, however, here, we either have a single agent species, or a single intermediate species. □

4.2.5 Unary Encoded Volume

When a system is volume-increasing or volume-decreasing, the reaction sequence length is polynomial in the volume of the system, so we achieve the following theorem.

Theorem. Reachability is in NP for Volume Increasing or Volume decreasing CRNs, with the volumes encoded in unary.

Proof. When the volume of a system is strictly increasing or decreasing, any reaction sequence between I and D is bounded by of size $\leq |I - D|$. The sequence can then be given as a 'yes' certificate for reachability. □

4.2.6 Unimolecular Reactions

Unimolecular reactions are of the form $A \to B$, i.e. preserving rules of size 1. If we are limited to only this type of reaction production is NL-complete. The NL-hardness result works for reachablility as well.

Theorem. Reachability and production in CRNs is NL-hard with rules of size $(1,1)$.

Proof. Given a directed graph G we create a set of species and reactions as follows. For each node $v \in G$ we create a species. For each edge $(a,b) \in E$ we create a reaction $a \to b$.

We reduce from the directed path problem: given two nodes $s,t \in G$, does there exist a path between s and t? Let our initial configuration be a single copy of the species representing s and our target configuration be a single copy of t. At each step the species will represent the current node in the path to reach t if and only if there exists a path. □

Theorem. Production with rules of size $(1,1)$ is NL-complete

Proof. Non-deterministically select a species i with a positive count in the initial configuration, check if the target species is reachable from i, if yes then the target species is producible. Checking reachability is in NL since the reactions can be viewed as directed edges. NL-hardness comes from Theorem 4.2.4 □

4.3 Reachability in Feed-Forward CRNs

Having established PSPACE-completeness for general CRNs, we consider the feed-forward restriction in which rule sets do not have cycles. Feed-forward CRNs are motivated in that they allow functional composition of CRNs (Vasić et al. 2022). We characterize the complexity of reachability for feed-forward CRNs under the assumption that the system does not contain either void or autogenesis rules, leaving a focused consideration of void and autogenesis rules for Section 4.4.

We first show NP-completeness of feed-forward systems in Section 4.3.1, even in the case of size-$(2,2)$ rules (i.e., bimolecular rules / Population Protocols), while at the same time being only 2-source and 2-consuming. We then show a polynomial-time solution to reachability in Section 4.3.4 for any feed-forward system that is either 1-source or 1-consuming, thus giving a complete characterization of feed-forward reachability.

4.3.1 NP-completeness for Bimolecular Reactions

In this section, we show that reachability (and production) in feed-forward systems is NP-complete for bimolecular reactions (rules of size at most $(2,2)$), even when the CRN is 2-source and 2-consuming. This hardness result is tight because a decrease to either 1-source or 1-consuming, as shown in Section 4.3.4, implies a polynomial time solution to reachability. We start with proof of membership in NP, followed by NP-hardness from a reduction from the Hamiltonian Path problem.

4.3.2 NP Membership

A key property of feed-forward systems is that any sequence of rule applications can be reordered such that all system rules are applied consecutively. This new ordering is still a valid sequence of applicable rules that reaches the same final configuration.

Definition (Ordered Application). A sequence of reactions is an ordered application if all the applications of any given rule take place right after each other in a contiguous sequence. An example of this is $R_1, R_1, \ldots R_1, R_2 \ldots R_2, R_3$.

Lemma. Let $C = (\Lambda, \Gamma)$ be a feed-forward CRN with a feed-forward ordering $F = \{R_0, R_1, \ldots R_{|R|-1}\}$ over Γ. Given configurations c, c', and a sequence of reactions $S = \{\ldots R_j, R_i \ldots\}$ in Γ that converts c to c' where $i < j$, then the sequence $S' = \{\ldots R_i, R_j \ldots\}$, i.e., the sequence obtained by swapping the two rules R_i and R_j, also transforms $c \to c'$.

Proof. Let X denote the configuration obtained by applying the rules of S up to just before the application of rule R_j. Let $R_j = (R_r^j, R_p^j)$ and $R_i = (R_r^i, R_p^i)$. As S is a valid sequence of rule applications, $X - R_r^j$ is a non-negative vector. Due to the feed-forward ordering F, the reactants R_r^i

do not occur as products in R_p^j, and thus $X - R_r^j - R_r^i$ is also non-negative, implying that rule R_i and R_j can be applied in either order. □

Corollary. A configuration D is reachable from a configuration I with a feed-forward CRN if and only if D is reachable from I by an ordered application of rules.

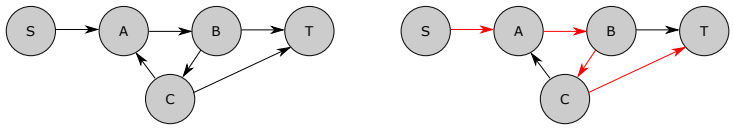
Lemma. The reachability problem is in NP for feed-forward CRNs that do not use autogenesis rules.

Proof. We utilize the ordered application as a polynomial-sized certificate from Corollary 4.3.1. While the sequence length (number of total rule applications) could be exponential, the sequence can be stored as a sequence of ordered pairs denoting each rule type in the sequence, along with a binary encoding of the number of times the rule is to be applied. The number of times a given rule type is applied is limited by the volume of the configuration right before such applications since there are no autogenesis rules. Since the volume of the initial configuration is small enough to be encoded in a polynomial number of bits, this property is preserved through the application of each rule type, implying the number of applications of each rule type is encoded in a polynomial bounded number of bits. Further, the result of applying a given number of iterations of a particular rule can be computed in poly-logarithmic time in the number of applications, which means the certificate can be checked in polynomial time. □

4.3.3 NP-Hardness

We now show the reachability problem is NP-complete for feed-forward CRNs even for size $(2,2)$-rules (bimolecular reactions, Population Protocols) and for 2-source, 2-consuming systems. We show this by a reduction from the Directed Hamiltonian Path problem with vertices of in-degree and out-degree of at most 2 (Plesnik 1979). For each vertex X in the graph $G = (V,E)$, we include $2+|V|$ states: an initial state X, a visited state X^v, and $|V|$ signal states X_i^*. We encode the edges of the graph in the rules as follows,

$$\text{Rules } R = \left\{ \begin{array}{c|c|c} S_i^* + A \to S^v + A_{i+1}^* & B_i^* + C \to B^v + C_{i+1}^* & B_i^* + T \to B^v + T_{i+1}^* \\ A_i^* + B \to A^v + B_{i+1}^* & C_i^* + A \to C^v + A_{i+1}^* & C_i^* + T \to C^v + T_{i+1}^* \end{array} \right\}$$



(a) Graph (b) Hamiltonian Path

$$\begin{array}{ccccc} \{S_0^* & A & B & C & T\} \\ \{S^v & A_1^* & B & C & T\} \\ \{S^v & A^v & B_2^* & C & T\} \\ \{S^v & A^v & B^v & C_3^* & T\} \\ \{S^v & A^v & B^v & C^v & T^4\} \end{array}$$

(c) Configurations

Figure 4.2: Our starting configuration $c = \{S_0^*, A, B, C, T\}$. Our goal configuration is $c' = \{S^v, A^v, B^v, C^v, T^4\}$. Each vertex must be changed to the visited state to reach the target, and the T must be the last vertex.

Definition (HAMPATH). Given a graph $G = \{V, E\}$ and two nodes $s, t \in V$, does there exist a path from s to t that visits each node precisely once?

An example reduction is shown in Figure 4.2. Given this reduction, any Hamiltonian path of graph G has a corresponding sequence of rules that end with every vertex, other than T, represented with the *visited* state, and T represented with the signal state matching the count of the vertices. Conversely, the only way to reach such a configuration corresponds directly to a Hamiltonian path of G from S to T, yielding the following result:

Theorem. Reachability is NP-complete for feed-forward CRNs with size $(2,2)$ rules that are 2-source and 2-consuming.

Proof. If there exists a Hamiltonian path P in G, there exists a sequence of rules r_P, each moving the * to a subsequent agent matching the sequence of vertices in P and further setting the previous agent to the visited state. Such a rule sequence ends with a single visited state for each vertex in the graph other than T and a $T|V|$ state.

Conversely, if a sequence of rules r that results in the target configuration exists, then a Hamilton path exists as each rule changes the agent location to the visited state, which can no longer change state, and moves the * to the next agent. The sequence of * species in the configuration represents the Hamilton path.

Finally, note that this system is feed-forward since an agent starts in the initial state, changes to a signal state, then to the visited state. We can ensure the ordering since the signal states are numbered. □

This reduction, with minor modification, can be adapted to achieve the following additional results.

Corollary. Reachability is NP-complete for feed-forward CRNs that are 2-source and 2-consuming with all rules of size $(2,1)$.

Proof. This is obtained by simply removing the visited states in the previous reduction. □

Corollary. Reachability is NP-complete for feed-forward CRNs that are 2-source and 2-consuming and with all rules of size $(1,2)$.

Proof. This follows from the above corollary combined with Lemma 4.3.7 (defined in Section 4.3.4). □

Corollary. Production is NP-complete for feed-forward CRNs with either size $(2,2)$ rules, size $(2,1)$ rules, or size $(1,2)$ rules, that are 2-source and 2-consuming.

Proof. In the previous reduction, the species $T_{|V|}$ is only producible if it is reached after reaching all other species, implying the previous reduction above holds for the production problem. □

Species-based Restrictions. Our definition of consuming and source is based on all species in the system. If this rule is relaxed to mean j-consuming/k-source per species, then we show the problem is NP-hard by a reduction from the 3-Dimensional Matching (3DM) problem.

Definition (Three Dimensional Matching Problem (3DM)). The 3DM problem takes as input a hypergraph $H = (X,Y,Z,T)$ where X,Y,Z are three disjoint sets, and $T \subseteq X \times Y \times Z$ is a set of *hyperedges*. The output is whether or not there exists a subset of T that covers all vertices in H without any overlap.

Corollary. Reachability in CRNs with each species being k-consuming/1-source or 1-consuming/k-source is NP-complete even with only one species being different than the others and the system being feed-forward without void/autogenesis rules.

Proof. We reduce from the 3DM problem, which is hard even when each vertex is covered by at most 3 hyper edges. Let $H = (X,Y,Z,T)$ be an input to the 3DM problem. From this, create an input to the reachability problem as follows. Let $\Lambda = \{S_v | v \in X \bigcup Y \bigcup Z\} \bigcup \{a\}$, and let $\Gamma = \{S_x + S_y + S_z \to a | (x,y,z) \in T\}$. The initial configuration I is the configuration in which each species has count 1 except species a with count 0. Let D be the configuration in which each species has count 0 except a, which has count $|X| = |Y| = |Z|$. Then D is reachable from I under (Λ, Γ) if and only if H has a three-dimensional mapping.

Species a is never consumed, but is produced by all $k = |X| = |Y| = |Z|$ of the rules. Thus, it is 0-consuming/k-source. All other species are never produced, and are consumed in 3 different rules. Thus, they are 3-consuming/0-source. Finally, since species A is never consumed, any ordering of the rules is feed-forward, and there are no void or autogenesis rules used. By Lemma 4.3.2, the problem is in NP. □

4.3.4 Feed-Forward, Single-Consuming/Single-Source

In this section, we establish Theorem 4.3.2 that shows the reachability problem is polynomial-time solvable for feed-forward, single-source rule sets that do not use void rules. We then extend this into Theorem 4.3.3 to show that reachability in feed-forward, single-consuming systems without autogenesis rules is also polynomial time solvable. Lastly, we give Corollary 4.3.6 that states the reachability problem for feed-forward, 1-source, and 1-consuming rule sets is polynomial-time solvable.

We now introduce some machinery for constructing an efficient algorithm for solving reachability in the special case of feed-forward, single-source rule sets with no void rules.

Definition (Leaf Rule.). A rule $R \in \Gamma$ is a *leaf rule* for Γ if the products of R do not occur as reactants within any *other* rule of the system Γ (however, the products of a leaf rule may occur as reactants within the same leaf rule). Note that a leaf rule could also be a void rule.

Lemma. A feed-forward, non-empty rule set has at least one leaf rule.

Proof. The final rule in the feed-forward ordering must be a leaf rule, as its product may not occur as a reactant for any previous rule of the system, which includes all rules other than itself. □

Lemma. If Γ is a feed-forward, single-source rule set without void rules, then so is any subset of Γ.

Proof. This follows from the definitions of feed-forward, single-source, and void rules. □

Definition (Pruned Configuration.). Consider a configuration I, a target configuration D, and feed-forward rule set Γ with non-void leaf rule $R = (R_r, R_p)$. If there exists a non-negative integer x such that $D(i) \setminus xR_a(i) = I(i)$, for all i where $R_p(i) \neq 0$, and R is applicable to $D \setminus xR_a$, we say that D is prunable towards I with respect to rule R, and define the *pruning* of D towards I with respect to R to be the configuration $\texttt{Prune}(D,I,R) = D \setminus xR_a$. If no such non-negative integer x exists, then we say that D is inconsistent with I for rule R. Note that the integer x, and thus the configuration $\texttt{Prune}(D,I,R) = D \setminus xR_a$, are unique as long as R is not a void leaf rule.

Lemma. If a configuration D is inconsistent with a configuration I for any leaf rule $R \in \Gamma$, then D is not reachable from I with a rule set Γ.

Proof. As R is a leaf rule, the counts of the product species in rule R are only affected by rule R among the rules of Γ. Therefore, if D is reachable from I, it must be possible to generate the counts of these species specified by D by some number of applications x of rule R. If no such integer exists, which is the definition of inconsistent, then the species counts for R's products cannot equal the counts specified by D, making D unreachable. □

Lemma. For a rule set Γ and configuration D that is consistent with I for non-void leaf rule $R \in \Gamma$, then D is reachable from I with a rule set Γ if and only if $D' = \texttt{Prune}(D,I,R)$ is reachable from configuration I with a rule set $\Gamma \setminus R$.

Proof. We first show that if $D' = \texttt{Prune}(D,I,R)$ is reachable with a rule set $\Gamma \setminus R$, then so is D with a rule set Γ. This is because once D' is reached, we know from the definition of $D' = \texttt{Prune}(D,I,R)$

that there exists a non-negative integer x such that x applications of rule R to configuration D' yields D, implying that D is reachable.

For the other direction, suppose we can reach D. From the sequence of rule applications that reaches D, there must be exactly some non-negative integer x applications of rule R. Create a modified sequence of configurations by omitting these x operations, and you get configuration $D' = \texttt{Prune}(D, I, R)$ by application of rules from $\Gamma \setminus R$, meaning D' is reachable from $\Gamma \setminus R$. □

Theorem. The reachability problem is solvable in polynomial time for a rule set Γ that is feed-forward, single-source, and without void rules.

Proof. Let I denote the starting configuration, D denote the destination configuration, and (Λ, Γ) be a feed-forward CRN without void rules for a given reachability instance. The following recursive algorithm solves reachability for a feed-forward, single-source rule set with no void rules.

As a base case, if the input system has 0 rules, then D is reachable if and only if $I = D$. Otherwise, identify a leaf rule R from Γ, which must exist by Lemma 4.3.3. Check if D is consistent with I for rule R. If not, return false, which is the correct answer by Lemma 4.3.5. If it is, then let $D' = \texttt{Prune}(D, I, R)$ denote the pruning of D towards I for rule R and return the result of recursively solving reachability with initial configuration I, ruleset $\Gamma \setminus R$, and destination configuration D', which is a valid input to this algorithm as $\Gamma \setminus R$ is assured to be a feed-forward, single-source rule set without void rules by Lemma 4.3.4, and is assured to yield the correct result by Lemma 4.3.6. In total, this algorithm executes $|\Gamma|$ prune operations, where each prune operation can be computed by a constant number of arithmetic operations on the volume of species in I and D. □

We now consider the case of reachability in feed-forward, single-consuming systems without autogenesis rules. Our approach is to *reverse* the given rule set Γ and apply our algorithm for theorem 4.3.2.

Definition. For a rule set Γ, let $\overleftarrow{\Gamma}$ be the *reverse* of Γ where $\overleftarrow{\Gamma} = \{(a,b) | (b,a) \in \Gamma\}$.

Lemma. For any two configurations A, B and rule set Γ, B is reachable from A in Γ if and only if A is reachable from B in $\overleftarrow{\Gamma}$.

Theorem. The reachability problem is solvable in polynomial time for a ruleset Γ that is feed-forward, single-consuming, and without autogenesis rules.

Proof. Given a CRN (Λ, Γ) where Γ is feed-forward, single-consuming, and without autogenesis rules, along with initial configuration I, and destination D, generate rule set $\overleftarrow{\Gamma}$. Note that $\overleftarrow{\Gamma}$ must be single-source as Γ is single-consuming, and must have no void rules since Γ has no autogenesis rules, and must be feed-forward since Γ is feed-forward. We can therefore determine if I is reachable from D under $\overleftarrow{\Gamma}$ in polynomial time by Theorem 4.3.2, which gives the answer to our original reachability problem by Lemma 4.3.7. □

Here we consider the case of reachability in feed-forward, 1-source, and 1-consuming systems with no further restrictions on the rule set. A single-consuming rule set will contain at most one void rule, allowing void rules to be considered when pruning a configuration.

Corollary. The reachability problem is solvable in polynomial time for a rule set Γ that is feed-forward, 1-source, and 1-consuming with no further restrictions on the rule set.

Proof. Let I denote the initial configuration, let D denote the target configuration, and let (Λ, Γ) be a feed-forward, 1-source, and 1-consuming CRN for a given reachability instance. With a single-consuming rule set Γ, any rule R, void or otherwise, may be used to prune D if there exists some non-negative integer x such that $D(i) \setminus xR_a(i) = I(i)$. By the definition of single-consuming, the integer x and the configuration $\texttt{Prune}(D, I, R) = D(i) \setminus xR_a(i)$ remain unique.

It follows that the recursive algorithm used in Theorem 4.3.2 solves reachability for a feed-forward, single-source, and single-consuming rule set by allowing void rules when pruning. □

4.4 Void and Autogenesis Rules

In our consideration of feed-forward CRNs, we omitted two classes of rules: *void* rules that consume reactants without creating any products and *autogenesis* rules that create products without consuming any reactants. One reason for separating these rules is that their lack of conservation of mass might mean they are not feasible in some experimental settings. Another important reason is that their inclusion alone substantially impacts the complexity of problems such as reachability.

In this section, we explore the reachability question in the scenario where *all* rules are void rules or, conversely, all rules are autogenesis rules. We show that void rules (or autogenesis rules) alone imply the NP-completeness of reachability, even if such systems are both feed-forward and 0-source. We specifically show NP-completeness for size $(3,0)$ void rules. We then explore the complexity of reachability with size $(2,0)$ void rules and provide a polynomial time solution when the system's volume is encoded in unary. We further show a polynomial time solution for binary encoded volume for a restricted class of *bipartite* $(2,0)$ CRNs. We leave the remaining general case of reachability with $(2,0)$ void rules as an open question.

We note by Lemma 4.3.6, that we may prove results for void only rules, and they are equivalent for autogenesis rules.

4.4.1 $(3,0)$ void rules / $(0,3)$ autogenesis rules

We show that reachability is NP-complete by a reduction from the 3-Dimensional Matching (3DM) problem (Definition 4.3.3).

Theorem. Reachability in CRNs is NP-complete with only rules of size $(3,0)$.

Proof. We reduce from the 3DM problem. Let $H = (X,Y,Z,T)$ be an input to the 3DM problem. From this, create an input to the reachability problem as follows. Let $\Lambda = \{S_v | v \in X \bigcup Y \bigcup Z\}$, and let $\Gamma = \{S_x + S_y + S_z \to \varnothing | (x,y,z) \in T\}$. Let configuration I be the configuration in which each species has count 1 and let D be the configuration in which each species has count 0. Then D is reachable from I under (Λ,Γ) if and only if H has a three-dimensional mapping. □

Corollary. Reachability for CRNs with only rules of size $(0,3)$ is NP-complete.

Proof. We show this by reduction from the reachability problem with size $(3,0)$ rules. Consider an instance of the reachability problem with an input of a CRN (Λ,Γ), an initial configuration I, and a destination configuration D in which rules in Γ are exclusively sized $(3,0)$ void rules. Let $\Lambda' = \Lambda$, $\Gamma' = \overleftarrow{\Gamma}$, initial configuration $I' = D$, and target configuration $D' = I$. Observe that Γ' consists of rules only of size $(0,3)$. By Lemma 4.3.7, D' is reachable from I' under ruleset Γ' if and only if D is reachable from I under ruleset Γ. □

4.4.2 $(2,0)$ rules with Unary Encoding

Just as we have connected three-dimensional mapping with size $(3,0)$ void rules to show NP-completeness, we can reduce in the other direction for size $(2,0)$ rules to provide an efficient solution for unary encoded volumes.

Theorem. Reachability in CRNs is in P with rules of size $(2,0)$ if configuration counts are encoded in unary.

Proof. We show this by reducing reachability in this scenario to the two-dimensional matching problem, which has an established polynomial time solution. We first consider the configuration $X = I - D$, creating a graph from this configuration. For each non-zero count species $X(i) > 0$, $X(i)$ vertices are added to the graph of type i. For each rule $i + j \to \emptyset$, we add edges to the graph connecting all vertices of type i to all vertices of type j. Then we have that X can reach the empty configuration if and only if the created graph has a perfect two-dimensional mapping, and thus I reaches D if and only if such a matching exists. □

4.4.3 $(2,0)$ rules with Binary Encoding

We now consider $(2,0)$ with binary encoded species counts, which permits a potentially exponential configuration volume, making the algorithm of Theorem 4.4.2 no longer polynomial time. In this scenario, we consider a new restriction in which the CRN rules are bipartite:

Definition. Bipartite CRN. A bipartite CRN (Λ,Γ) is one in which the species Λ can be partitioned into two disjoint sets Λ_1 and Λ_2 such that for each rule $R \in \Gamma$, there are at most 2 reactants of R and they do not occur within the same partition of Λ.

Determining if a CRN is bipartite can be solved in polynomial time by a bipartite graph detection algorithm. If the CRN is bipartite, we reduce the problem to the maximum flow problem.

Theorem. Reachability is polynomial-time solvable for bipartite CRNs with $(2,0)$ rules.

Proof. We show this by reducing reachability for a $(2,0)$ rule bipartite CRN to the maximum network flow problem. Consider an input $(2,0)$-rule bipartite CRN (Λ,Γ) with partitions Λ_1 and

Λ_2, input configuration I, and output configuration D. From this, generate a max-flow instance as follows: for each $s \in \Lambda$, let the network contain a corresponding vertex v_s. For each rule, $a + b \to \emptyset \in \Gamma$, add an infinite capacity edge between vertices v_a and v_b. For each $x_i \in \Lambda_1$, add an edge from the source vertex to vertex v_{x_i} of capacity $I[i] - D[i]$, and for each $y_i \in \Lambda_2$, add an edge from v_{y_i} to the sink vertex of capacity $I[i] - D[i]$. The maximum-flow of this network is equal to the configuration volume $I - D$ if and only if D is reachable from I under Γ, and therefore reachability can be computed in polynomial time using the Edmonds-Karp maximum-flow algorithm. □

Although this algorithm only works with bipartite CRNs, we conjecture that the problem is in P and leave it as an important open question related to general matching in weighted graphs.

Conjecture. Reachability is polynomial-time solvable for CRNs with $(2,0)$ rules.

REFERENCES

Adleman, Leonard et al. (2001). "Running time and program size for self-assembled squares". In: *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pp. 740–748.

Adleman, Leonard M. et al. (2002). "Combinatorial optimization problems in self-assembly". In: *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pp. 23–32.

Alaniz, Robert M. et al. (2022). *Reachability in Restricted Chemical Reaction Networks*. DOI: 10.48550/ARXIV.2211.12603. URL: https://arxiv.org/abs/2211.12603.

Ani, Joshua et al. (2022). "Traversability, reconfiguration, and reachability in the gadget framework". In: *Intl. Conf. and Work. on Algorithms and Comp.*

Benz, Florian and Timo Kötzing (2013). "An effective heuristic for the smallest grammar problem". In: *Proc. of the 15th annual conf. on genetic and evolutionary computation*, pp. 487–494.

Caballero, David et al. (2020). "Verification and Computation in Restricted Tile Automata". In: *26th Inter. Conf. on DNA Computing and Molecular Programming*. Vol. 174. DNA'20, 10:1–10:18. ISBN: 978-3-95977-163-4.

Cantu, Angel A. et al. (2019). "Covert Computation in Self-Assembled Circuits". In: *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Vol. 132. Leibniz International Proceedings in Informatics (LIPIcs), 31:1–31:14. ISBN: 978-3-95977-109-2.

Cheng, Qi et al. (2005). “Complexities for Generalized Models of Self-Assembly”. In: *SIAM Journal on Computing* 34, pp. 1493–1515.

Demaine, Erik D, Martin L Demaine, et al. (2008). “Staged self-assembly: nanomanufacture of arbitrary shapes with O (1) glues”. In: *Natural Computing* 7.3, pp. 347–370.

Demaine, Erik D, Isaac Grosof, et al. (2018). “Computational Complexity of Motion Planning of a Robot through Simple Gadgets”. In: *9th Intl. Conf. on Fun with Algorithms (FUN 2018)*.

Demaine, Erik D, Robert A Hearn, et al. (2022). “PSPACE-Completeness of Reversible Deterministic Systems”. In: *arXiv preprint arXiv:2207.07229*.

Demaine, Erik D, Dylan H Hendrickson, and Jayson Lynch (2020). “Toward a General Complexity Theory of Motion Planning: Characterizing Which Gadgets Make Games Hard”. In: *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*.

Demaine, Erik D. et al. (2011). “One-dimensional staged self-assembly”. In: *Proceedings of the 17th international conference on DNA computing and molecular programming*. DNA’11. Pasadena, CA, pp. 100–114.

Fu, Bin et al. (2022). “Reachability in Population Protocols”. In: *The Japan Conference on Discrete and Computational Geometry, Graphs, Games (JCDCG^3)*.

Kari, Lila et al. (Apr. 2016). “Binary Pattern Tile Set Synthesis Is NP-Hard”. In: *Algorithmica* 78.1, pp. 1–46. DOI: 10.1007/s00453-016-0154-7. URL: https://doi.org/10.1007/s00453-016-0154-7.

Keenan, Alexandra et al. (Mar. 2016). “Fast arithmetic in algorithmic self-assembly”. In: *Natural Computing* 15.1, pp. 115–128. ISSN: 1572-9796.

Plesnik, J (1979). “The np-completeness of the hamiltonian cycle problem in planar diagraphs with degree bound two”. In: *Information Processing Letters*.

Thachuk, Chris and Anne Condon (2012). “Space and energy efficient computation with DNA strand displacement systems”. In: *International Workshop on DNA-Based Computers*.

Vasić, Marko et al. (2022). “Programming and training rate-independent chemical reaction networks”. In: *Proceedings of the National Academy of Sciences* 119.24, e2111552119.

Vollmer, Heribert (1999). *Introduction to Circuit Complexity*. Springer Berlin Heidelberg. DOI: 10.1007/978-3-662-03927-4. URL: https://doi.org/10.1007/978-3-662-03927-4.

Winfree, Erik (1998). “Algorithmic self-assembly of DNA”. English. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-02-23. PhD thesis, p. 132. ISBN: 978-0-591-96599-5. URL: https://go.openathens.net/redirector/utrgv.edu?url=https://www.proquest.com/dissertations-theses/algorithmic-self-assembly-dna/docview/304420561/se-2.

Woods, Damien et al. (Mar. 2019). “Diverse and robust molecular algorithms using reprogrammable DNA self-assembly”. In: *Nature* 567.7748, pp. 366–372. DOI: 10.1038/s41586-019-1014-9. URL: https://doi.org/10.1038/s41586-019-1014-9.

BIOGRAPHICAL SKETCH

Andrew Ray Rodriguez is a proud Hispanic born and raised in the Rio Grande Valley. He graduated from high school in 2017 with an associate's degree in computer science from South Texas College. In 2021, he graduated from the University of Texas at Austin with his bachelor's in computer science. He then went on to join the master's program at the University of Texas Rio Grande Valley studying computer science. He joined the Algorithmic Self-Assembly Research Group (ASARG) at UTRGV under Dr. Robert Schweller and Dr. Tim Wylie, and thus began his research career. He graduated with his master's in computer science in 2023. He can be reached at aroday23@gmail.com.