# Toward a molecular programming language for algorithmic self-assembly

by

Matthew John Patitz

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Jack H. Lutz, Major Professor
Pavan Aduri
James I. Lathrop
John Mayfield
Gurpur Prabhu

Iowa State University

Ames, Iowa

2010

## TABLE OF CONTENTS

## LIST OF FIGURES

# ACKNOWLEDGMENTS

The path to earning a Ph.D. has been long and difficult, and without the help, support, and guidance of many people I could never have completed it. It is impossible to list all of the contributions from friends, family, and colleagues who helped me along the way, so I will make a small and inadequate list here. For those that I leave out, my gratitude is no less and I look forward to fully expressing it in the years to come, since I hope that the transition to the next stage of my career still keeps us in close contact.

It is often noted that even the longest journey begins with a single step, and for me that single step was initiated by my friend Bob Kortzeborn. One morning at the gym, Bob asked me if I knew why he had gotten his PhD. His answer: “Because it’s the best degree you can get.” That statement and his continual prodding created the seed that eventually grew into my desire to return to school and get my own PhD.

Next, I wish to thank my advisor, Jack Lutz, for introducing me to the study of theoretical computer science, and more specifically to algorithmic self-assembly. It is this research area which captured my imagination and gave me the energy and excitement to throw myself into research. Self-assembly is an extremely exciting research field which has just the right mixture of theory and potential for experimental work to keep me fascinated and eager to always seek out the next line of research, as well as to grind through the mountains of tedious work that can also be required at times. Jack taught me the rigor required to be a theoretician, and that was no easy task.

Additionally, by bringing me into his research group, Jack brought me into contact with some fantastic researchers. Of these, none played a greater part in my education than Scott

Summers, to whom I owe a debt of gratitude that is staggering. Scott has played an enormous number of roles throughout my PhD career, from teacher and mentor to collaborator and co-author to, most importantly, friend. As novice graduate students, Scott and I met and quickly formed a strong bond and team, relying heavily on each other to learn the ropes of academia and the research community, and to bootstrap our research careers. Early on, we proposed an "if and only if" proposition, tying the prospects of our individual graduations together, and at every step of the way Scott has been hard working and generous with his time and energy, guaranteeing our mutual success. Throughout the years, our teamwork has only strengthened as we eventually matured into productive researchers and earned our degrees, and I can only hope that it continues into the future.

Several senior members of our research group, including Dave Doty, Satyadev Nandakumar, and Xiaoyang Gu, as well as my friend and co-author Damien Woods, have provided me with great examples of first class scientists, plus invaluable advice, help, and inspiration, not to mention much needed consolation and support at many points during my career.

As a graduate student, it has been my great privilege and pleasure to work with and become friends with some tremendous faculty members. Among them is John Mayfield, who taught what has probably been my favorite class of all time, and who has since been a great source of information and a creative sounding board for many of my ideas about research outside of my domain of expertise. Gurpur Prabhu, one of the best instructors I've ever met, has been a constant mentor and has been the primary force in molding my teaching philosophy and practices. Beyond that, he worked hard every semester to ensure that I always had the maximum support as a teaching assistant, which was vital to my ability to continue on.

Of course, no such journey is even remotely possible without the support of family. My parents, Dan and Trish, have been some of the most supportive and inspirational people in my life. Their selfless sacrifices throughout my life have opened countless doors for me, and the examples of hard work and dedication to principles they have provided forever shaped me. My sister, Jenni, and brother, Zack, have also given important encouragement and support that kept me going.

Finally, and most importantly, has been the unconditional support and love of my wife, Andrea. When I decided to throw away a promising career to return to school for the uncertain pursuit of goals I didn't yet even firmly understand, without hesitation she stood behind me. Throughout the intervening five years, she has provided unceasing support, all the while working hard to allow me the financial freedom to be back in school as well as giving birth to and helping to raise our wonderful daughter, Josie. She enabled this pursuit and has earned this degree as much as I. For this, I will be forever in her debt. And to Josie, who, on a daily basis, reminds me that no matter how serious and difficult things may seem, there is always room for some carefree giggling and comforting cuddles, I owe a large chunk of my sanity and happiness.

The road has been long, but exceptionally rewarding, mostly because of the people I have met and the relationships formed. Despite the many hardships and complaints I made (numerous, I admit!), I am profoundly happy with the entire journey. It is with the greatest excitement that I look forward to moving on and putting my new knowledge and skills to use.

(The work in this thesis was supported in part by National Science Foundation Grants 0652569 and 0728806.)

# ABSTRACT

Self-assembly is the process whereby relatively simple components autonomously combine to form more complex objects. Nature exhibits self-assembly to form everything from microscopic crystals to living cells to galaxies. With a desire to both form increasingly sophisticated products and to understand the basic components of living systems, scientists have developed and studied artificial self-assembling systems. One such framework is the Tile Assembly Model introduced by Erik Winfree in 1998. In this model, simple two-dimensional square 'tiles' are designed so that they self-assemble into desired shapes. The work in this thesis consists of a series of results which build toward the future goal of designing an abstracted, high-level programming language for designing the molecular components of self-assembling systems which can perform powerful computations and form into intricate structures.

The first two sets of results demonstrate self-assembling systems which perform infinite series of computations that characterize computably enumerable and decidable languages, and exhibit tools for algorithmically generating the necessary sets of tiles. In the next chapter, methods for generating tile sets which self-assemble into complicated shapes, namely a class of discrete self-similar fractal structures, are presented. Next, a software package for graphically designing tile sets, simulating their self-assembly, and debugging designed systems is discussed. Finally, a high-level programming language which abstracts much of the complexity and tedium of designing such systems, while preventing many of the common errors, is presented.

The summation of this body of work presents a broad coverage of the spectrum of desired outputs from artificial self-assembling systems and a progression in the sophistication of tools used to design them. By creating a broader and deeper set of modular tools for designing

self-assembling systems, we hope to increase the complexity which is attainable. These tools provide a solid foundation for future work in both the Tile Assembly Model and explorations into more advanced models.

# CHAPTER 1. OVERVIEW

## 1.1 Introduction

### 1.1.1 The power and promise of self-assembly

Stars, star systems and their remnants, and interstellar clouds are pulled by gravitational forces to spontaneously combine in complex patterns to form galaxies. Water molecules suspended in air whose temperature, currents, and humidity all play a role, autonomously coalesce into intricate crystalline structures known as snowflakes. Large numbers of convoluted but precisely interlocking proteins aimlessly float around in intracellular fluids, eventually bumping into each other and binding to form highly specific viruses. What is the common thread, the tool used by nature in all of these examples? It is a process known as self-assembly.

Self-assembly is the process in which relatively simple components brought into contact with each other experience local interactions guided by basic rules, and combine to form increasingly complex structures. There is no externally guiding force or direction, just the summation of the undirected local interactions. Despite this seemingly haphazard method of assembly, nature harnesses the power of self-assembly, from the nanoscale to galactic scales, to produce myriad structures of mind boggling complexity.

As science has progressed, so has our understanding of many of these naturally occurring self-assembling systems. And as technology has progressed, so has our desire and ability to mimic nature and create artificial self-assembling systems. Mastery of such systems could provide the necessary pieces to make atomically-precise manufacturing, the fabrication of materials and products such that the position of each atom is specified by design, a reality. This could enable the creation of vastly more powerful computing systems, nano-biomedical devices, and a host of additional technological wonders that we can only dream of today. Furthermore, increased understanding of self-assembly will provide valuable insights into the origination and functioning of living systems.

With these goals in mind, scientists from fields such as Biology, Chemistry, Computer Science, and Mathematics, among others, have been pursuing various approaches to studying self-assembly. For example, research is already underway to produce self-assembling molecular scaffoldings onto which complex devices such as circuits can attach [32], as well as for making three-dimensional containers which can be used to transport nanoscale cargo like medicinal drugs to highly specific locations [8]. While these examples display the progress of experimental research into self-assembly, there is also a great need for and amount of work in theoretical studies. Theoretical work helps to steer the experimental work by uncovering basic properties needed to create powerful self-assembling systems as well as showing their mathematical boundaries. One active area of such theoretical work is the Tile Assembly Model.

### 1.1.2 Self-assembly in the Tile Assembly Model

In 1998, Winfree [52] introduced the (abstract) Tile Assembly Model (TAM), which is an effective version of Wang tiling [50, 51] and a mathematical model of DNA self-assembly pioneered by Seeman [46]. In the TAM, the fundamental components are un-rotatable, but translatable "tiles", or 2-dimensional squares, whose sides are labeled with glue "colors" and "strengths." Tiles that are placed next to each other *interact* if the glue colors on their abutting sides match, and they *bind* to each other if the strength on their abutting sides matches and is at least as great as a given "temperature" value. Rothemund and Winfree [45, 44] later refined the model, and Lathrop, Lutz and Summers [36] gave a treatment of the TAM in which equal status is bestowed upon the self-assembly of infinite and finite structures. There are also several generalizations [7, 37, 28] of the TAM.

In the TAM, *assemblies* are aggregations of tiles which bind to each other, and are formed as tiles bind one-by-one to an assembly which begins as a predefined *seed*. The process of tile adsorption is asynchronous and nondeterministic. Despite its deliberate over-simplification, the TAM is a computationally and geometrically expressive model. For instance, Winfree [52] proved that the TAM is computationally universal, and thus can be directed algorithmically.

## 1.2 Programming algorithmic self-assembly

Since its introduction, the TAM has provided a rich breeding ground for research directions and results. In particular, my co-authors and I have generated numerous complex constructions as well as impossibility results [20, 17, 18, 35, 42, 41, 22]. In so doing, we have progressively increased the size and intricacy of the tile assembly systems that we design. Additionally, we have uncovered more of the mathematical principles that can guide self-assembling systems and learned how to think of them in high-level abstractions. The combination of these factors has led us to increasingly approach the design of self-assembling systems in terms of programming - we have learned how to modularize constructions and to treat modules and sub-modules as computing primitives that can be combined in well-defined ways.

In the remainder of this section, I provide an overview of a series of results that display aspects of programming algorithmic self-assembly, from constructions for which we have created computer programs to algorithmically generate the necessary tile sets, to programming tools that we have created and released to the scientific community for designing and creating such tile sets. The general direction of this research is toward further and further encapsulation of the design of self-assembling systems into a programming paradigm, with the eventual goal being a ‘programming language’ which can be used to design a full spectrum of algorithmically self-assembling systems.

## 1.3 Organization of this thesis

Chapter 2 gives a formal definition of the TAM along with some basic examples for clarity. Overviews of the remaining chapters are presented in the following subsections, with a concluding chapter to talk about future direction.

### 1.3.1 Computability and Complexity in Self-Assembly

In Chapter 3 we explore the impact of geometry on computability and complexity in the abstract Tile Assembly Model. Our first main theorem says that there is a roughly quadratic function $f$ such that a set $A \subseteq \mathbb{Z}^+$ is computably enumerable if and only if the set $X_A = \{(f(n), 0) \mid n \in A\}$ – a simple representation of $A$ as a set of points on the $x$-axis – self-assembles in Winfree's sense. In contrast, our second main theorem says that there are decidable sets $D \subseteq \mathbb{Z} \times \mathbb{Z}$ that do *not* self-assemble in Winfree's sense.

Our first main theorem is established by an explicit translation of an arbitrary Turing machine $M$ to a modular tile assembly system $\mathcal{T}_M$, together with a proof that $\mathcal{T}_M$ carries out concurrent simulations of $M$ on all positive integer inputs. The construction for this result was fully implemented, and was the first construction of such complexity that we are aware of to be fully implemented. Our implementation, as well as our proof of correctness, made heavy use of modularization. We developed and released C++ code which can be used to generate tile sets for this construction, given $M$ as input.

### 1.3.2 Self-Assembly of Decidable Sets

The theme of Chapter 4 is computation in the TAM. We first review a simple, well-known tile assembly system (the "wedge construction") that is capable of universal computation. We then extend the wedge construction to prove the following result: if a set of natural numbers is decidable, then it and its complement's canonical two-dimensional representation self-assemble. This leads to a novel characterization of decidable sets of natural numbers in terms of self-assembly. Finally, we show that our characterization is robust with respect to various (restrictive) geometrical constraints.

This work is another example of translating an infinite series of Turing machine computations into a representative assembly in the TAM. Similar to the above work, the construction for this result was fully implemented as a modular collection of sub-tile sets which can be programmatically generated by C++ code that we developed and released. It also provides a few new ‘programming’ tricks, such as overlapping the functionality of multiple modules by ‘passing signals’ through assemblies in well defined ways.

### 1.3.3 Self-Assembly of Discrete Self-Similar Fractals

In Chapter 5, we search for *theoretical* limitations of the TAM, along with techniques to work around such limitations. Specifically, we investigate the self-assembly of fractal shapes in the TAM. We prove that no self-similar fractal weakly self-assembles at temperature 1 in a locally deterministic tile assembly system, and that certain kinds of discrete self-similar fractals do not strictly self-assemble at any temperature. Additionally, we extend the fiber construction of Lathrop, Lutz and Summers [36] to show that any discrete self-similar fractal belonging to a particular class of “nice” discrete self-similar fractals has a fibered version that strictly self-assembles in the TAM.

The construction for this work was also fully implemented, with C++ source code and programs for generating tile sets released to the scientific community. This construction made heavy use of counters of various bases in complex combinations and further expands the tool kit of programming modules available to researchers programming algorithmic self-assembly.

### 1.3.4 Simulation of Self-Assembly in the Abstract Tile Assembly Model with ISU TAS

As previously mentioned, as research has progressed in the aTAM, the self-assembling structures being studied have become progressively more complex. This increasing complexity, along with a need for standardization of definitions and tools among researchers, motivated the development of the Iowa State University Tile Assembly Simulator (ISU TAS). ISU TAS is a graphical simulator and tile set editor for designing and building 2-D and 3-D aTAM tile assembly systems and simulating their self-assembly. Chapter 6 reviews the features and functionality of ISU TAS and describes how it can be used to further research into the complexities of the aTAM.

ISU TAS provides a solid foundation for research into programming in the TAM, and since it is released as open source, is hoped to be extensible for more generic research into programming algorithmic self-assembly.

### 1.3.5 A Domain-Specific Language for Programming in the Tile Assembly Model

In Chapter 7 we introduce a domain-specific language (DSL) for creating sets of tile types for simulations of the aTAM. The language defines objects known as tile templates, which represent related groups of tiles, and a small number of basic operations on tile templates that help to eliminate the error-prone drudgery of enumerating such tile types manually or with low-level constructs of general-purpose programming languages. The language is implemented as a class library in Python (a so-called *internal DSL*), but is presented independently of Python or object-oriented programming, with emphasis on support for a visual editing tool for creating large sets of complex tile types.

This DSL encapsulates many of the concepts central to programming in the TAM and allows them to be abstracted at a much higher level. In this way, it greatly helps to extend the paradigm of programming as it applies to designing self-assembling systems.

# CHAPTER 2. The abstract Tile Assembly Model

In this chapter we give a brief and relatively self-contained introduction to the abstract Tile Assembly Model (aTAM) that is adequate for reading this paper. More formal details and discussion may be found in [52, 45, 44, 36]. Our notation is that of [36]. Note that throughout this paper we use aTAM and TAM interchangeably. Unless explicitly stated, all references to the TAM refer to the aTAM.

We work in the 2-dimensional discrete Euclidean space $\mathbb{Z}^2$. We define the set $U_2 = \{(0,1),(1,0),\ (0,-1),(-1,0)\}$ to be the set of all *unit vectors*, i.e., vectors of length 1, in $\mathbb{Z}^2$. We regard the 4 elements of $U_2$ as (names of the cardinal) *directions* in $\mathbb{Z}^2$ (e.g. North, South, East, and West). We write $[X]^2$ for the set of all 2-element subsets of a set $X$. All *graphs* here are undirected graphs, i.e., ordered pairs $G = (V,E)$, where $V$ is the set of *vertices* and $E \subseteq [V]^2$ is the set of *edges*.

A *grid graph* is a graph $G = (V,E)$ in which $V \subseteq \mathbb{Z}^2$ and every edge $\{\vec{a},\vec{b}\} \in E$ has the property that $\vec{a} - \vec{b} \in U_2$. The *full grid graph* on a set $V \subseteq \mathbb{Z}^2$ is the graph $G^{\#}_V = (V,E)$ in which $E$ contains *every* $\{\vec{a},\vec{b}\} \in [V]^2$ such that $\vec{a} - \vec{b} \in U_2$.

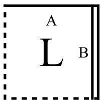


Figure 2.1: Example tile type

Intuitively, a tile type $t$ is a unit square that can be translated, but not rotated, so it has a well-defined "side $\vec{u}$" for each $\vec{u} \in U_2$. Each side $\vec{u}$ is covered with a "glue" of "color" $\text{col}_t(\vec{u})$

and "strength" $\mathrm{str}_t(\vec{u})$ specified by its type $t$. Tiles are depicted as squares whose various sides have a dashed line, one solid line, or two solid lines, indicating whether the glue strengths on those sides are 0, 1, or 2, respectively. Thus, for example, a tile of the type shown in Figure 2.1 has glue of strength 0 on the left and bottom, glue of color 'A' and strength 1 on the top, and glue of color 'B' and strength 2 on the right. This tile also has a label 'L', which plays no formal role.

Two tiles $t$ and $t'$ that are placed at the points $\vec{a}$ and $\vec{a}+\vec{u}$ respectively, *bind* with *strength* $\mathrm{str}_t\left(\vec{u}\right)$ if and only if $\left(\mathrm{col}_t\left(\vec{u}\right), \mathrm{str}_t\left(\vec{u}\right)\right) = \left(\mathrm{col}_{t'}\left(-\vec{u}\right), \mathrm{str}_{t'}\left(-\vec{u}\right)\right)$. In this paper, all glues have strength 0, 1, or 2. Each side's "color" is indicated by an alphanumeric label. An example of a tile set is shown in Figure 2.2.

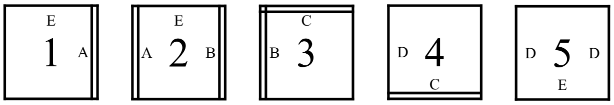


Figure 2.2: Example tile set

Given a set $T$ of tile types and a "temperature" $\tau \in \mathbb{N}$, a *$\tau$-$T$-assembly* is a partial function $\alpha : \mathbb{Z}^2 \dashrightarrow T$ - intuitively, a placement of tiles at some locations in $\mathbb{Z}^2$, with points $\vec{x} \in \mathbb{Z}^2$ at which $\alpha(\vec{x})$ is undefined interpreted to be empty space, so that dom $\alpha$ is the set of points with tiles. The *binding graph of* an assembly $\alpha$ is the grid graph $G_\alpha = (V, E)$, where $V =$ dom $\alpha$, and $\{\vec{m}, \vec{n}\} \in E$ if and only if (1) $\vec{m} - \vec{n} \in U_2$, (2) $\mathrm{col}_{\alpha(\vec{m})}\left(\vec{n} - \vec{m}\right) = \mathrm{col}_{\alpha(\vec{n})}\left(\vec{m} - \vec{n}\right)$, (3)$\mathrm{str}_{\alpha(\vec{m})}\left(\vec{n} - \vec{m}\right) = \mathrm{str}_{\alpha(\vec{n})}\left(\vec{m} - \vec{n}\right)$, and (4) $\mathrm{str}_{\alpha(\vec{m})}\left(\vec{n} - \vec{m}\right) > 0$. An assembly is *$\tau$-stable* if it cannot be broken up into smaller assemblies without breaking bonds of total strength at least $\tau$ (i.e., if every cut of $G_\alpha$ cuts edges, the sum of whose strengths is at least $\tau$). For each $t \in T$, the *$\tau$-$t$-frontier* of $\alpha$, written as $\partial_t^\tau \alpha$, is the set of all locations to which $t$ can be $\tau$-stably added to $\alpha$. For each $t \in T$, we write $\partial_t \alpha$ for the set of all locations to which a tile of type $t$ can be $\tau$-stably added. We write $\partial \alpha$ for the set of all locations to which some tile can be $\tau$-stably added to $\alpha$. We refer to $\partial \alpha$ as the *frontier* of $\alpha$. If $\alpha$ and $\alpha'$ are assemblies, then

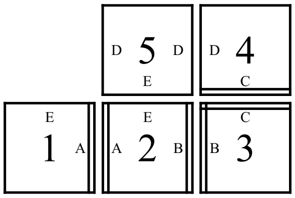


Figure 2.3: Example 2-$T$-assembly, where $T$ is the tile set shown in Figure 2.2. Note that the size of the frontier of this assembly is 1.

$\alpha$ is a *subassembly* of $\alpha'$, and we write $\alpha \sqsubseteq \alpha'$, if dom $\alpha \subseteq$ dom $\alpha'$ and $\alpha(\vec{m}) = \alpha'(\vec{m})$ for all $\vec{m} \in$ dom $\alpha$. $\alpha'$ is a *single-tile extension* of $\alpha$ if $\alpha \sqsubseteq \alpha'$ and dom $\alpha'$ − dom $\alpha$ is a singleton set. In this case, we write $\alpha' = \alpha + (\vec{m} \mapsto t)$, where $\{\vec{m}\} =$ dom $\alpha'$ − dom $\alpha$ and $t = \alpha'(\vec{m})$

Self-assembly begins with a *seed assembly* $\sigma$ and proceeds asynchronously and nondeterministically, with tiles adsorbing one at a time to the existing assembly in any manner that preserves $\tau$-stability at all times. A *tile assembly system* (*TAS*) is an ordered triple $\mathcal{T} = (T, \sigma, \tau)$, where $T$ is a finite set of tile types, $\sigma$ is a seed assembly with finite domain, and $\tau \in \mathbb{N}$ is the temperature. A *generalized tile assembly system* (*GTAS*) is defined similarly, but without the finiteness requirements.

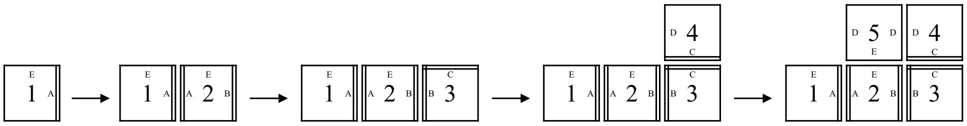


Figure 2.4: An example assembly sequence with respect to the tile set shown in Figure 2.2.

An assembly sequence in a TAS $\mathcal{T}$ is a (finite or infinite) sequence $\vec{\alpha} = (\alpha_0, \alpha_1, \ldots)$ of assemblies in which each $\alpha_{i+1}$ is obtained from $\alpha_i$ by the addition of a single tile. The *result* res$(\vec{\alpha})$ of such an assembly sequence is its unique limiting assembly (This is the last assembly in the sequence if the sequence is finite). Figure 2.4 shows an example of a 2-$T$-assembly sequence (where $T$ is the tile set shown in Figure 2.2), with its result being the right most assembly.

We write $\mathcal{A}[\mathcal{T}]$ for the set of all assemblies that can arise via some assembly sequence in $\mathcal{T}$. An assembly $\alpha$ is *terminal*, and we write $\alpha \in \mathcal{A}_\Box[\mathcal{T}]$, if no tile can be $\tau$-stably added to it. It is clear that $\mathcal{A}[\mathcal{T}] \subseteq \mathcal{A}_\Box[\mathcal{T}]$.

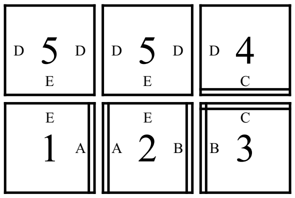


Figure 2.5: An example of a 2-$T$-assembly that is terminal, where $T$ is the tile set shown in Figure 2.2.

The set $\mathcal{A}[\mathcal{T}]$ is partially ordered by the relation $\longrightarrow$ defined by

$$\alpha \longrightarrow \alpha' \iff \begin{array}{l} \text{there is an assembly} \\ \text{sequence } \vec{\alpha} = (\alpha_0, \alpha_1, \ldots) \\ \text{such that } \alpha_0 = \alpha \text{ and} \\ \alpha' = \text{res}(\vec{\alpha}). \end{array}$$

We say that $\mathcal{T}$ is *directed* (a.k.a. *deterministic*, *confluent*, *produces a unique assembly*) if the relation $\longrightarrow$ is directed, i.e., if for all $\alpha, \alpha' \in \mathcal{A}[\mathcal{T}]$, there exists $\alpha'' \in \mathcal{A}[\mathcal{T}]$ such that $\alpha \longrightarrow \alpha''$ and $\alpha' \longrightarrow \alpha''$. It is easy to show that $\mathcal{T}$ is directed if and only if there is a unique terminal assembly $\alpha \in \mathcal{A}[\mathcal{T}]$ such that $\sigma \longrightarrow \alpha$. The reader is cautioned that the term "directed" has also been used for a different, more specialized notion in self-assembly [6]. We interpret "directed" to mean "deterministic", though there are multiple senses in which a TAS may be deterministic or nondeterministic.

A set $X \subseteq \mathbb{Z}^2$ *weakly self-assembles* if there exists a TAS $\mathcal{T} = (T, \sigma, \tau)$ and a set $B \subseteq T$ ($B$ constitutes the "black" tiles) such that $\alpha^{-1}(B) = X$ holds for every assembly $\alpha \in \mathcal{A}_\Box[\mathcal{T}]$. A set $X$ *strictly self-assembles* if there is a TAS $\mathcal{T}$ for which every assembly $\alpha \in \mathcal{A}_{\Box}[\mathcal{T}]$ satisfies $\mathrm{dom}\ \alpha = X$. Note that if $X$ strictly self-assembles, then $X$ weakly self-assembles. (Let all tiles be black.)

For the sake of example, let $\mathcal{T} = (T, \sigma, 2)$ be the tile assembly system where $T$ is the set of tile types given in Figure 2.2 and $\sigma$ is an assembly such that $\sigma(0,0) = t$, where $t \in T$ is the unique tile type with the label '1' (and undefined elsewhere). Then it is clear that the set

$$X = \{(0,0),(1,0),(2,0),(0,1),(1,1),(2,1)\}$$

strictly self-assembles in $\mathcal{T}$. Furthermore, if we define the set $B$ of "black" tiles to be the singleton set containing the tile type labeled '5', then it follows that the set

$$Y = \{(0,1),(1,1)\}$$

weakly self-assembles in $\mathcal{T}$.

## 2.1 A Brief Sketch of Local Determinism

In general, even a directed TAS may have a very large (perhaps uncountably infinite) number of different assembly sequences leading to its terminal assembly. This seems to make it very difficult to prove that a TAS is directed. Fortunately, Soloveichik and Winfree [48] have defined a property, *local determinism*, of assembly sequences and proven the remarkable fact that, if a TAS $\mathcal{T}$ has *any* assembly sequence that is locally deterministic, then $\mathcal{T}$ is directed. We briefly review local determinism below, and refer the reader to [48] for a full definition.

**Notation 1.** For each assembly $\alpha$, each $\vec{m} \in \mathbb{Z}^2$, and each $\vec{u} \in U_2$,

$$\mathrm{str}_{\alpha}(\vec{m}, \vec{u}) = \mathrm{str}_{\alpha(\vec{m})}(\vec{u}) \cdot [\![ \alpha(\vec{m})(\vec{u}) = \alpha(\vec{m}+\vec{u})(-\vec{u}) ]\!],$$

where $[\![\phi]\!]$ is the *Boolean* value of the statement $\phi$. The Boolean value on the right is 0 if $\{\vec{m}, \vec{m}+\vec{u}\} \not\subseteq \mathrm{dom}\ \alpha$.

**Notation 2.** If $\vec{\alpha} = (\alpha_i \mid 0 \leq i < k)$ is an assembly sequence in the TAS $\mathcal{T} = (T, \sigma, \tau)$, and $\vec{m} \in \mathbb{Z}^2$, then the *$\vec{\alpha}$-index* of $\vec{m}$ is

$$i_{\vec{\alpha}}(\vec{m}) = \min\{i \in \mathbb{N} \mid \vec{m} \in \mathrm{dom}\ \alpha_i\}.$$

**Observation 2.1.1.** $\vec{m} \in \mathrm{dom}\ \mathrm{res}(\vec{\alpha}) \Leftrightarrow i_{\vec{\alpha}}(\vec{m}) < \infty$.

**Notation 3.** If $\vec{\alpha} = (\alpha_i \mid 0 \leq i < k)$ is an assembly sequence in the TAS $\mathcal{T} = (T, \sigma, \tau)$, then, for $\vec{m}, \vec{m}' \in \mathbb{Z}^2$,

$$\vec{m} \prec_{\vec{\alpha}} \vec{m}' \Leftrightarrow i_{\vec{\alpha}}(\vec{m}) < i_{\vec{\alpha}}(\vec{m}').$$

**Definition.** (Soloveichik and Winfree [48]) Let $\vec{\alpha} = (\alpha_i \mid 0 \leq i < k)$ be an assembly sequence in the TAS $\mathcal{T} = (T, \sigma, \tau)$, and let $\alpha = \mathrm{res}(\vec{\alpha})$. For each location $\vec{m} \in \mathrm{dom}\ \alpha$, define the following sets of directions.

1. $\mathrm{IN}^{\vec{\alpha}}(\vec{m}) = \left\{\vec{u} \in U_2 \;\middle|\; \vec{m} + \vec{u} \prec_{\vec{\alpha}} \vec{m} \text{ and } \mathrm{str}_{\alpha_{i_{\vec{\alpha}}(\vec{m})}}(\vec{m}, \vec{u}) > 0\right\}$.
2. $\mathrm{OUT}^{\vec{\alpha}}(\vec{m}) = \left\{\vec{u} \in U_2 \;\middle|\; -\vec{u} \in \mathrm{IN}^{\vec{\alpha}}(\vec{m} + \vec{u})\right\}$.

Intuitively, $\mathrm{IN}^{\vec{\alpha}}(\vec{m})$ is the set of sides on which the tile at $\vec{m}$ initially binds in the assembly sequence $\vec{\alpha}$, and $\mathrm{OUT}^{\vec{\alpha}}(\vec{m})$ is the set of sides on which this tile propagates information to future tiles. Note that $\mathrm{IN}^{\vec{\alpha}}(\vec{m}) = \varnothing$ for all $\vec{m} \in \alpha_0$.

**Notation 4.** If $\vec{\alpha} = (\alpha_i \mid 0 \leq i < k)$ is an assembly sequence in the TAS $\mathcal{T} = (T, \sigma, \tau)$, $\alpha = \mathrm{res}(\vec{\alpha})$, and $\vec{m} \in \mathrm{dom}\ \alpha - \mathrm{dom}\ \alpha_0$, then

$$\vec{\alpha} \setminus \vec{m} = \alpha \restriction \left(\mathrm{dom}\ \alpha - \{\vec{m}\} - \left(\vec{m} + \mathrm{OUT}^{\vec{\alpha}}(\vec{m})\right)\right).$$

(Note that $\vec{\alpha} \setminus \vec{m}$ may or may not be a stable assembly.)

**Definition.** (Soloveichik and Winfree [48]). An assembly sequence $\vec{\alpha} = (\alpha_i \mid 0 \leq i < k)$ in the TAS $\mathcal{T} = (T, \sigma, \tau)$, with result $\alpha$ is *locally deterministic* if it has the following three properties.

1. For all $\vec{m} \in \mathrm{dom}\ \alpha - \mathrm{dom}\ \alpha_0$,

$$\sum_{\vec{u} \in \mathrm{IN}^{\vec{\alpha}}(\vec{m})} \mathrm{str}_{\alpha_{i_{\vec{\alpha}}(\vec{m})}}(\vec{m}, \vec{u}) = \tau.$$

2. For all $\vec{m} \in \text{dom}\ \alpha - \text{dom}\ \alpha_0$ and all $t \in T - \{\alpha(\vec{m})\}$, $\vec{m} \notin \partial_t^{\mathcal{T}}(\vec{\alpha} \setminus \vec{m})$.

3. $\alpha \in \mathcal{A}_{\Box}[\mathcal{T}]$.

Intuitively, $\vec{\alpha}$ is locally deterministic if (1) each tile added in $\vec{\alpha}$ "just barely" binds to the assembly; (2) if a tile of type $t_0$ at a location $\vec{m}$ and its immediate "OUT-neighbors" are deleted from the *result* of $\vec{\alpha}$, then no tile of type $t \neq t_0$ can attach itself to the thus-obtained configuration at location $\vec{m}$; and (3) the result of $\vec{\alpha}$ is terminal. See Figures 2.4 and 2.5 for an example of a locally deterministic assembly sequence (where Figure 2.5 shows the terminal assembly).

**Definition.** A TAS $\mathcal{T} = (T, \sigma, \tau)$ is *locally deterministic* if there exists a locally deterministic assembly sequence $\vec{\alpha} = (\alpha_i \mid 0 \leq i < k)$ with $\alpha_0 = \sigma$.

**Lemma 2.1.2.** (Soloveichik and Winfree [48]) If the TAS $\mathcal{T} = (T, \sigma, \tau)$ is locally deterministic, then every assembly sequence $\vec{\alpha} = (\alpha_i \mid 0 \leq i < k)$ in $\mathcal{T}$ is locally deterministic.

**Theorem 2.1.3.** (Soloveichik and Winfree [48]) Every locally deterministic TAS is directed.

*Proof Sketch.* It follows from the definition of local determinism that all tiles placed in an assembly by a locally deterministic tile assembly system have well-defined input and output sides. These input and output sides are shown to be invariant across all possible assembly sequences. By a standard inductive argument, it follows that only a single tile type can bind in each location in all possible assembly sequences. This yields the result that all assembly sequences lead to a single, common terminal assembly and thus the TAS is directed. Note that as a technicality required by the proof technique, every tile in a locally deterministic TAS must bind with exactly strength $\tau$. □

# CHAPTER 3. Computability and Complexity in Self-Assembly

The work in this chapter was performed with co-authors James I. Lathrop, Jack H. Lutz, and Scott M. Summers. It has been published as [35] and [34].

## 3.1 Introduction

Winfree [52] proved that the Tile Assembly Model is computationally universal, i.e., that any Turing machine can be encoded into a finite set of tile types whose self-assembly simulates that Turing machine. The computational universality of the Tile Assembly Model implies that self-assembly can be algorithmically directed, and hence that a very rich set of structures can be formed by self-assembly. However, as we shall see, this computational universality does not seem to imply a simple characterization of the class of structures that can be formed by self-assembly. The difficulty is that self-assembly (like sensor networks, smart materials, and other topics of current research [9, 10]) is a phenomenon in which the *spatial* aspect of computing plays a crucial role. Two processes, namely self-assembly and the computation that directs it, must take place in the same space and time.

This chapter presents two main theorems on the interplay between geometry and computation in tile self-assembly. To explain our first main theorem, define the function $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ by

$$f(n) = \binom{n+1}{2} + (n+1)\lfloor \log n \rfloor + 6n - 2^{1+\lfloor \log(n) \rfloor} + 2.$$

Note that $f$ is a reasonably simple, strictly increasing, roughly quadratic function of $n$. For each set $A \subseteq \mathbb{Z}^+$, the set

$$X_A = \{(f(n), 0) | n \in A\}$$

is thus a straightforward representation of $A$ as a set of points on the positive $x$-axis.

Our first main theorem says that *every* computably enumerable set $A$ of positive integers (decidable or otherwise) self-assembles in the sense that there is a tile assembly system $\mathcal{T}_A$ in which the representation $X_A$ self-assembles. Conversely, the existence of such a tile assembly system implies the computable enumerability of $A$.

In contrast, our second main theorem says that there are *decidable* sets $D \subseteq \mathbb{Z}^2$ that do *not* self-assemble in any tile assembly system. In fact, we exhibit such a set $D$ for which the condition $(m, n) \in D$ is decidable in time polynomial in $|m| + |n|$.

Taken together, our two main theorems indicate that the interaction between geometry and computation in self-assembly is not at all simple. Further investigation of this interaction will improve our understanding of tile self-assembly and, more generally, spatial computation.

The proof of our first main theorem has two features that may be useful in future investigations. First, we give an explicit transformation (essentially a compiler, implemented in C++) of an arbitrary Turing machine $M$ to a tile assembly system $\mathcal{T}_M$ whose self-assembly carries out concurrent simulations of $M$ on (the binary representation of) all positive integer inputs. Second, we prove two lemmas – a pseudoseed lemma and a multiseed lemma – that enable us to reason about tile assembly systems in a modular fashion. This modularity, together with the local determinism method of Soloveichik and Winfree [48], enables us to prove the correctness of $\mathcal{T}_M$.

## 3.2 Pseudoseeds and Multiseeds

This section introduces two conceptual tools that enable us to reason about tile assembly systems in a modular fashion.

The idea of our first tool is intuitive. Suppose that our objective is to design a tile assembly system $\mathcal{T}$ in which a given set $X \subseteq \mathbb{Z}^2$ self-assembles. The set $X$ might have a subset $X^*$ for which it is natural to decompose the design into the following two stages.

(i) Design a TAS $\mathcal{T}_0 = (T_0, \sigma, \tau)$ in which an assembly $\sigma^*$ with domain $X^*$ self-assembles.

(ii) Extend $T_0$ to a tile set $T$ such that $X$ self-assembles in the TAS $\mathcal{T}^* = (T, \sigma^*, \tau)$

We would then like to conclude that $X$ self-assembles in the TAS $\mathcal{T} = (T, \sigma, \tau)$. This will not hold in general, but it does hold if (i) continues to hold with $T$ in place of $T_0$ and $\sigma^*$ is a pseudoseed in the following sense.

**Definition.** Let $\mathcal{T} = (T, \sigma, \tau)$ be a GTAS. A *pseudoseed* of $\mathcal{T}$ is an assembly $\sigma^* \in \mathcal{A}[\mathcal{T}]$ with the property that, if we let $\mathcal{T}^* = (T, \sigma^*, \tau)$, then, for every assembly $\alpha \in \mathcal{A}[\mathcal{T}]$, there exists an assembly $\alpha' \in \mathcal{A}[\mathcal{T}^*]$ such that $\alpha \sqsubseteq \alpha'$.

The following lemma says that the above definition has the desired effect.

**Lemma 3.2.1** (pseudoseed lemma)**.** If $\sigma^*$ is a pseudoseed of a GTAS $\mathcal{T} = (T, \sigma, \tau)$ and $\mathcal{T}^* = (T, \sigma^*, \tau)$, then $\mathcal{A}_\Box[\mathcal{T}] = \mathcal{A}_\Box[\mathcal{T}^*]$.

*Proof.* ("$\subseteq$" direction) Let $\alpha \in \mathcal{A}_\Box[\mathcal{T}]$. This means that $\sigma \to \alpha$. The definition of pseudoseed tells us that there exists $\alpha' \in \mathcal{A}[\mathcal{T}^*]$ such that $\alpha \sqsubseteq \alpha'$. But since $\alpha$ is terminal, we must have $\alpha = \alpha'$, whence $\sigma \sqsubseteq \sigma^* \sqsubseteq \alpha$. It follows, by Rothemund's lemma (Lemma 2 on p. 57 of [44]), that $\sigma^* \to \alpha$.

("$\supseteq$" direction) Let $\alpha \in \mathcal{A}_\Box[\mathcal{T}^*]$. Then we have $\sigma^* \to \alpha$. Moreover, $\sigma \to \sigma^*$ because $\sigma^* \in \mathcal{A}[\mathcal{T}]$. Thus, $\sigma \to \sigma^* \to \alpha$, and it follows, by the transitivity of $\to$, that $\alpha \in \mathcal{A}_\Box[\mathcal{T}]$. □

Note that the pseudoseed lemma entitles us to *reason as though* the self-assembly proceeds in stages, even though this may not actually occur. (E.g., the pseudoseed $\sigma^*$ may itself be infinite, in which case the self-assembly *of* $\sigma^*$ and the self-assembly *from* $\sigma^*$ must occur concurrently.)

Our second tool for modular reasoning is a bit more involved. Suppose that we have a tile set $T$ and list $\sigma_0, \sigma_1, \sigma_2, \ldots$ of seeds that, for each $i$, the TAS $\mathcal{T}_i = (T, \sigma_i, \tau)$ has a desired assembly $\alpha_i$ as its result. If the assemblies $\alpha_0, \alpha_1, \alpha_2, \ldots$ have disjoint domains, then it might be possible for *all* these assemblies to grow from a "multiseed" $\sigma^*$ that has $\sigma_0, \sigma_1, \sigma_2, \ldots$ embedded in it. We now define a sufficient condition for this.

**Definition.** Let $T$ and $T'$ be sets of tile types with $T \subseteq T'$, and let $\vec{\sigma} = (\sigma_i \mid 0 \leq i < k)$ be a sequence of $\tau$-$T$-assemblies, where $k \in \mathbb{Z}^+ \cup \{\infty\}$. A $\vec{\sigma}$-$T$-$T'$-*multiseed* is a $\tau$-$T'$ assembly $\sigma^*$ with the property that, if we write

$$\mathcal{T}^* = (T', \sigma^*, \tau)$$

and

$$\mathcal{T}_i = (T, \sigma_i, \tau)$$

for each $0 \leq i < k$, then the following four conditions hold.

1. For each $i$, $\sigma_i \sqsubseteq \sigma^*$.

2. For each $i \neq j$, $\alpha \in \mathcal{A}[\mathcal{T}_\rangle]$, $\alpha' \in \mathcal{A}[\mathcal{T}_|]$, $\vec{m} \in \text{dom}\ \alpha$, and $\vec{m'} \in \text{dom}\ \alpha'$, $\vec{m} - \vec{m'} \in U_2 \cup \{\vec{/}\} \Rightarrow \vec{m}, \vec{m'} \in \text{dom}\ \sigma^*$. (Recall that $U_2$ is the set of unit vectors in $\mathcal{Z}^2$.)

3. For each $i$, and each $\alpha \in \mathcal{A}[\mathcal{T}_\rangle]$, there exists $\alpha^* \in \mathcal{A}[\mathcal{T}^*]$ such that $\alpha \sqsubseteq \alpha^*$.

4. For each $\alpha^* \in \mathcal{A}[\mathcal{T}^*]$, there exists, for each $0 \leq i < k$, $\alpha_i \in \mathcal{A}[\mathcal{T}_\rangle]$ such that $\alpha^* \sqsubseteq \sigma^* + \sum_{0 \leq i < k} \alpha_i$.

5. For each $\alpha \in \mathcal{A}[\mathcal{T}^*]$, $\alpha^{-1}(T' - T) \subseteq \text{dom}\ \sigma^*$.

Note: In condition 4 we are using the operation + defined as follows. If $\alpha, \alpha' : \mathbb{Z}^2 \dashrightarrow T$ are *consistent*, in the sense that they agree on $\text{dom}\ \alpha \cap \text{dom}\ \alpha'$, then $\alpha + \alpha' : \mathbb{Z}^2 \dashrightarrow T$ is the unique partial function satisfying $\text{dom}\ (\alpha + \alpha') = \text{dom}\ \alpha \cup \text{dom}\ \alpha'$, $\alpha \sqsubseteq \alpha + \alpha'$, and $\alpha' \sqsubseteq \alpha + \alpha'$. This is extended to summations $\sum_{0 \leq i < k} \alpha_i$ in the obvious way. The assemblies being summed in condition 4 are consistent by conditions 1 and 2.

Intuitively, the four conditions in the above definition can be stated as follows.

1. The seeds $\sigma_i$ are embedded in $\sigma^*$.

2. Assemblies in $\mathcal{A}[\mathcal{T}_\rangle]$ and assemblies in $\mathcal{A}[\mathcal{T}_|]$ do not interfere with each other.

3. $\sigma^*$ does not interfere with assemblies in $\mathcal{A}[\mathcal{T}_\rangle]$.

4. $\sigma^*$ does not produce anything other than what its embedded seeds $\sigma_i$ produce.

5. Tile types in $T' - T$ cannot occur outside $\sigma^*$.

The following lemma says that the multiseed definition has the desired effect.

**Lemma 3.2.2** (multiseed lemma)**.** Let $T \subseteq T'$ be sets of tile types, and let $\vec{\sigma} = (\sigma_i \mid 0 \leq i < k)$ be a sequence of $\tau$-$T$-assemblies, where $k \in \mathbb{Z}^+ \cup \{\infty\}$. If $\sigma^*$ is a $\vec{\sigma}$-$T$-$T'$-multiseed of $\mathcal{T}^*$ and $\mathcal{T}_i$ $(0 \leq i < k)$ are defined as in the multiseed definition, then

$$\mathcal{A}_\Box[\mathcal{T}^*] = \left\{ \sigma^* + \sum_{0 \leq i < k} \alpha_i \;\middle|\; \text{each } \alpha_i \in \mathcal{A}_\Box[\mathcal{T}_\rangle] \right\}.$$

*Proof.* ("$\subseteq$" direction) Let $\alpha^* \in \mathcal{A}[\mathcal{T}^*]$. It follows by part (4) of definition 3.2 that $\alpha^* = \sigma^* + \sum_{0 \leq i < k} \alpha_i$, where $\alpha_i \in \mathcal{A}[\mathcal{T}_\rangle]$, Since $\alpha^*$ is terminal, for each $i$, $\alpha_i \in \mathcal{A}_\Box[\mathcal{T}_\rangle]$, whence $\alpha^* \in \left\{ \sigma^* + \sum_{0 \leq i < k} \alpha_i \;\middle|\; \text{each } \alpha_i \in \mathcal{A}_\Box[\mathcal{T}_\rangle] \right\}$.

("$\supseteq$" direction) For each $0 \leq i < k$, let $\alpha_i \in \mathcal{A}_\Box[\mathcal{T}_\rangle]$ and let $\alpha = \sigma^* + \sum_{0 \leq i < k} \alpha_i$ . It suffices to shaw that $\alpha \in \mathcal{A}_\Box[\mathcal{T}^*]$.

Condition 3 of the multiseed definition tells us that, for each $0 \leq i < k$, there is an assembly $\alpha_i^* \in \mathcal{A}[\mathcal{T}^*]$ such that $\alpha_i \sqsubseteq \alpha_i^*$. Since $\sigma^* \sqsubseteq \alpha_i^*$, it follows that

$$\sigma^* + \alpha_i \sqsubseteq \alpha_i^*.$$

By condition 4 of the multiseed definition, there is, for each $0 \leq i < k$ and $0 \leq j < k$, an assembly $\alpha_{ij} \in \mathcal{A}[\mathcal{T}_\rangle]$ such that, for all $0 \leq i < k$,

$$\alpha_i^* \sqsubseteq \sigma^* + \sum_{0 \leq j < k} \alpha_{ij}.$$

For each $0 \leq i < k$, let

$$D_i = \text{dom } \alpha_i^* \cap (\text{dom } \sigma^* \cup \text{dom } \alpha_{ii}),$$

and let

$$\hat{\alpha}_i^* = \alpha_i^* \restriction D_i.$$

By (1), (2), and condition 2 of the multiseed definition, we have

$$\sigma^* + \alpha_i \sqsubseteq \hat{\alpha}_i^*$$

for all $0 \leq i < k$. Now the assemblies $\hat{\alpha}_i^*$ are all consistent with one another, and each $\alpha_i$ is terminal in $\mathcal{A}[\mathcal{T}_\rangle]$, so, by condition 5 of the multiseed definition, we must have

$$\alpha = \sum_{0 \leq i < k} \hat{\alpha}_i^*.$$

For each $0 \leq i < k$, let $\vec{\alpha}^{*i}$ be an assembly sequence from $\sigma^*$ to $\alpha_i^*$, and let $\hat{\vec{\alpha}}^{*i}$ be the sequence obtained from $\vec{\alpha}^{*i}$ by deleting all additions of tiles not in $\sigma^* + \alpha_{ii}$. By condition 2 of the multiseed definition, each $\hat{\vec{\alpha}}^{*i}$ is a $\tau$-T'-assembly sequence from $\sigma^*$ to $\hat{\alpha}_i^*$. By (3), then, if we dovetail the assembly sequences $\hat{\vec{\alpha}}^{*i}(0 \leq i < k)$, we get an assembly sequence $\hat{\vec{\alpha}}^*$ from $\sigma^*$ to $\alpha$, whence $\alpha \in \mathcal{A}[\mathcal{T}^*]$. Since the $\alpha_i$ are terminal, conditions 4 and 5 of the multiseed definition tell us that $\alpha \in \mathcal{A}_\Box[\mathcal{T}^*]$. □

## 3.3 Self-Assembly of Computably Enumerable Sets

In [52], Winfree proved that the Tile Assembly Model is Turing universal in two dimensions. In this section, we prove a stronger result: for every TM $M$, there exists a directed TAS that simulates $M$ on (the binary representation of) *every* input $x \in \mathbb{N}$ in the two dimensional discrete Euclidean plane. We state our result precisely in the following theorem.

**Theorem 3.3.1** (first main theorem)**.** If $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ is defined as in section 1, then, for all $A \subseteq \mathbb{Z}^+$, $A$ is computably enumerable if and only if the set $X_A = \{(f(n), 0) \mid n \in A\}$ self-assembles.

*"⇐" direction.* Let $f$ be as defined in the introduction, and assume the hypothesis. Then there exists a 2-TAS $\mathcal{T} = (T, \sigma, \tau)$ in which the set $X_A$ self-assembles. Let $B$ be the set of "black" tile types given by the definition of self-assembly. Fix some enumeration $\vec{a}_1, \vec{a}_2, \vec{a}_3 \ldots$ of $\mathbb{Z}^2$, and let $M$ be the TM, defined as follows.

```
**Require:** $n \in \mathbb{N}$
  $\alpha := \sigma$
  **while** $(f(n), 0) \notin \text{dom } \alpha$ **do**
    choose the least $j \in \mathbb{N}$ such that $\vec{a}_j \in$
    $\partial^\tau \alpha_i$
    choose $t \in T$ such that $\vec{a}_j \in \partial_t^\tau \alpha_i$
    $\alpha := \alpha + (\vec{a}_j \mapsto t)$
  **end while**
  **if** $\alpha\left((f(n), 0)\right) \in B$ **then**
    accept
  **else**
    reject
  **end if**
```

It is clear from above that $L(M) = A$, whence $A$ is computably enumerable. □

To prove the "⇒" direction, we exhibit, for any TM $M$, a directed TAS $\mathcal{T}_M = (T, \sigma, \tau)$ that correctly simulates $M$ on all inputs $x \in \mathbb{Z}^+$ in the two dimensional discrete Euclidean plane. We describe our construction, and give the complete specification for $T$ in the remainder of this section.

### 3.3.1 Overview of Construction

Intuitively, $\mathcal{T}_M$ self-assembles a "gradually thickening bar", immediately below the positive $x$-axis with upward growths emanating from well-defined intervals of points. For each $x \in \mathbb{Z}^+$, there is an upward growth that simulates $M$ on $x$. If $M$ halts on $x$, then (a portion of) the upward growth associated with the simulation of $M(x)$ eventually stops, and sends a signal down along the right side of the upward growth via a one-tile-wide-path of tiles to the point $(f(x), 0)$, where a black tile is placed. See Figure 3.1 for a finite, yet intuitive snapshot of this infinite process.

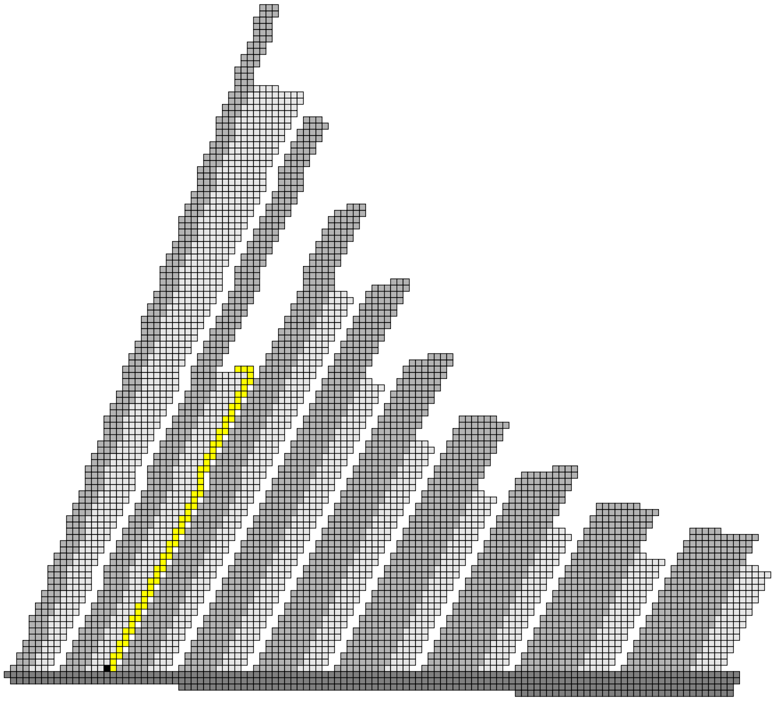


Figure 3.1: Simulation of $M$ on every input $x \in \mathbb{N}$. Notice that $M(2)$ halts - indicated by the black tile along the $x$-axis.

Our tile assembly system $\mathcal{T}_M$ is divided into three modules: the ray, the planter, and the TM module, which control the spacing between successive simulations, the initiation of upward growth, and the actual simulations of $M$ on each positive integer, respectively.

### 3.3.2 Overview Of The Ray Module

The first module in our construction is the ray module (middle shade of gray squares in Figure 3.1). For any $3 \leq w \in \mathbb{Z}^+$, a ray of width $w$ is a fixed-width, periodic, binary counter that repeatedly counts from 0 to $2^w - 1$, such that each integer is counted once, and then immediately copied once before the value of the binary counter is incremented. Essentially, a ray of width $w$ is a discrete line of constant thickness $w$, having a kind of "slope" that depends on $w$ in the following way. In every other row (except for two special cases), the first tile to attach does so on top of the second-to-leftmost tile in the previous row. Thus, a ray of width

$w$ will have a slope of $\frac{2^w}{2^{w-1}-1} = 2 + \frac{2}{2^{w-1}-1}$. This implies that the set of points occupied by properly spaced, consecutive rays of strictly increasing width, will not only be disjoint but the width of the gap in between such rays will increase without limit.

### 3.3.3 Details Of Construction Of The Ray Module

(It is to be understood that, although not explicit in our discussion, tile types in each module are colored with a “module indicator” symbol to prevent erroneous binding between the tile types of different modules.)

The ray module is a tile set of 68 tile types. Although in our construction the ray is not a self-contained tile assembly system, it can be made to be one with a trivial change. For any $3 \leq w \in \mathbb{N}$, a ray of width $w$ is a fixed-width, periodic, binary counter that repeatedly counts from 0 to $2^w - 1$, such that each integer is counted once, and then immediately copied once before the value of the binary counter is incremented. Our ray is based on the binary counter from [45], and thus increments right-to-left, and then copies the previous value left-to-right. However, since we construct the ray to operate on the reverse of each integer (i.e., the LSB of each integer is its left most bit), increments proceed left-to-right and copies right-to-left. The most important feature of the ray is that the first, and hence leftmost, tile to attach in any increment row does so on top of the second-to-leftmost tile in the previous copy row, *unless* the bit pattern of the previous increment row is of the form $1^{w-1}(0+1)$. A detailed example of the former case is shown in Figure 3.2.

In the latter case, the ray simply proceeds without “shifting” to the right by one unit. Copy rows are able to search for the two special bit patterns by using ‘a’ and ‘s’ signals, respectively. For instance, when a copy row begins to self-assemble, and the current value of the counter is less than $2^{w-1}$, an ‘a’ signal is propagated left through the copy row until a 0 bit is encountered in the previous increment row, thus breaking the signal. If no such bit exists, the signal will reach the leftmost tile of the copy row, and cause the next increment row to attach without being shifted one unit to the right. This situation is shown with detail in Figure 3.3.

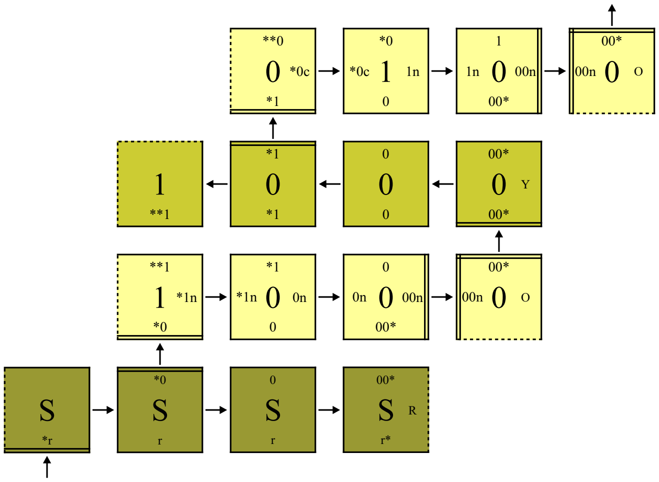


Figure 3.2: The first row of tiles is the initial row in a ray of width 4. The second row of tiles represents the value 1, and is the first increment row. The third row of tiles simply copies the value of the previous increment row.

Otherwise, the next increment row attaches on top of the second-to-leftmost tile in the current copy row. The ‘s’ signal searches for the bit pattern $1^w$ in a similar fashion, shown with detail in Figure 3.4.

Finally, each row in the ray exhibits a particular “color” on the right side of its rightmost tile so as to allow a third (soon to be discussed) module to “know what to do and when to do it.” Each increment row signals orange unless it contains the bit pattern $0^{w-1}(0+1)$, in which case it signals green for $0^{w-1}1$, and indigo otherwise. Each copy row signals yellow. However, if it is the copy row after an increment row with the bit pattern $0^{w-1}1$, then it signals blue. The initial row signals red.

In our construction, the upward growth of every ray is initiated by the planter. This can be seen with detail in Figure 3.5, where the top row of tiles (excluding the leftmost tile) will initiate the self-assembly of a ray having a width of 6. It is clear that our construction has the following properties.

1. A ray having a width of $w$ will have a “slope” of $\frac{2^w}{2^{w-1}-1} = 2 + \frac{2}{2^{w-1}-1}$, which clearly

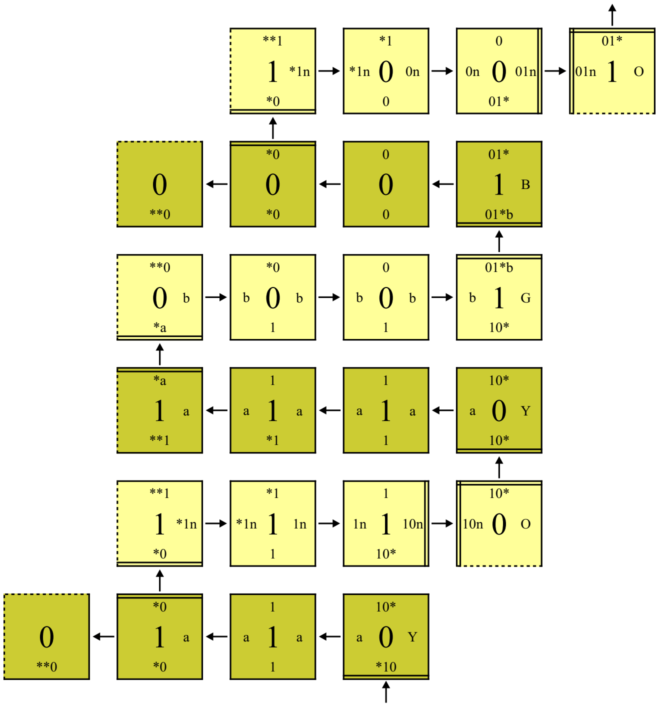


Figure 3.3: If the current value of the binary counter is a power of 2 (the fourth row of tiles), then the increment row that represents this value does not shift to the right by one unit. This situation is detected in the third (copy) row by the 'a' signal, which travels unimpeded right-to-left. The complementary 'b' signal is then sent left-to-right in the subsequent increment row to initiate a green signal.

tends to 2 as $w \to \infty$;

2. for every $3 \leq w \in \mathbb{Z}^+$, there is one and only one ray having a width of $w$;
3. thinner rays appear before thicker rays;
4. the set of points occupied by any ray is disjoint from the set of points occupied by any other ray and
5. the width between successive rays grows without limit.

The following is a list of tile types that make up the ray module.

1. Initial row - add the following tile types:

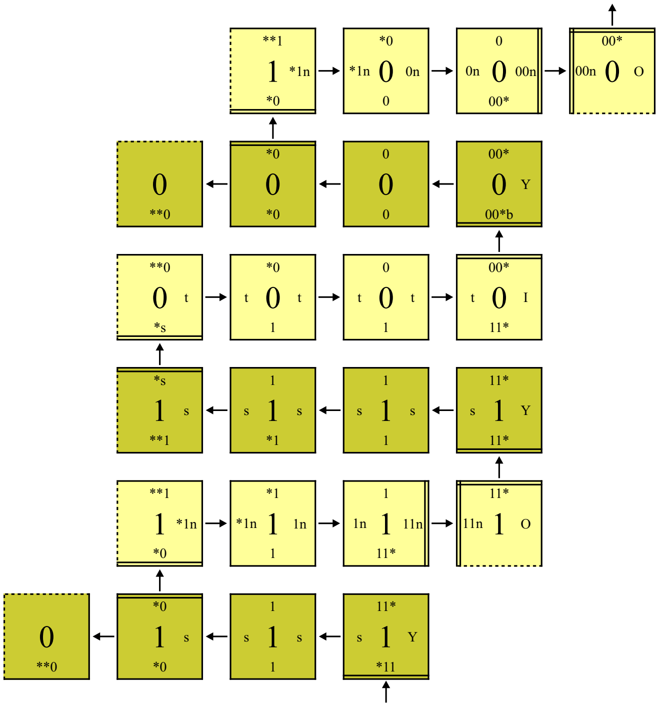
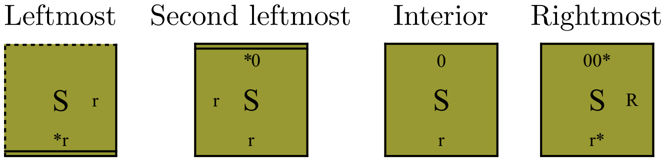
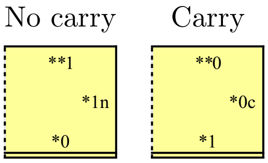


Figure 3.4: If the current value of the binary counter rolls over to all zeros (the fourth row of tiles), then the increment row that represents this value does not shift to the right by one unit. Similar to how the ‘a’ signal detects powers of 2, the ’s’ signal (the third row of tiles) detects when the counter is about to roll over. The ‘t’ signal initiates an indigo signal.



2. Tile types that perform general case increments (left to right).

   (a) Leftmost tile - add the following tile types:

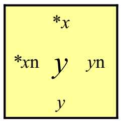


   (b) Second leftmost tile type.

i. If there is no carry bit to propagate, then for all $x, y \in \{0, 1\}$, add the following tile types:

No carry in

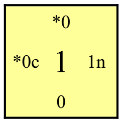
No carry out



ii. If there is a carry bit coming in from the left, add the following tile types:

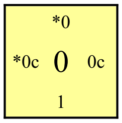
Carry in,

No carry out



Carry in,

Carry out

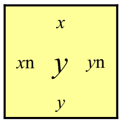


(c) Interior tile type.

i. If there is no carry bit to propagate, then for all $x, y \in \{0, 1\}$, add the following tile types:

No carry in

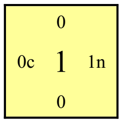
No carry out



ii. If there is a carry bit coming in from the left, add the following tile types:

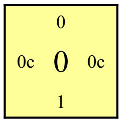
Carry in,

No carry out



Carry in,

Carry out

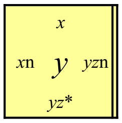


(d) Second rightmost tile type.

i. If there is no carry bit to propagate, then for all $x, y, z \in \{0, 1\}$, add the following tile types:

No carry in

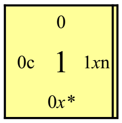
No carry out



ii. If the carry bit being propagated stops at the second most significant bit, then for all $x \in \{0, 1\}$, add the following tile types:

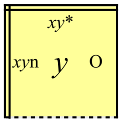
Carry in

No carry out



(e) Rightmost tile type - for all $x, y \in \{0, 1\}$, add the following tile type:

No carry in

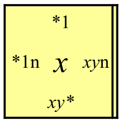


(f) Second right *and* second leftmost tile type (when $w = 3$.)

i. If there is no carry bit to propagate, then for all $x, y \in \{0, 1\}$, add the following tile types:

No carry in,

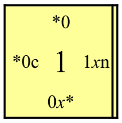
No carry out



ii. If the carry bit stops at this location, then for all $x \in \{0, 1\}$, add the following tile types:

Carry in,

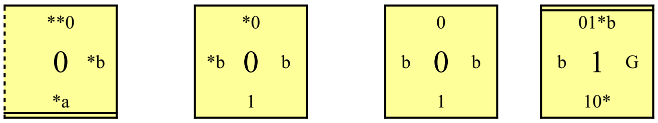
No carry out



3. Tile types that perform special case increments.

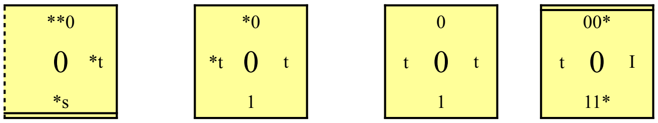
   (a) When the current row represents a power of 2 - add the following tile types:

   Leftmost Second leftmost Interior Rightmost

   (b) When the current row represents 0 - add the following tile types:

   Leftmost Second leftmost Interior Rightmost

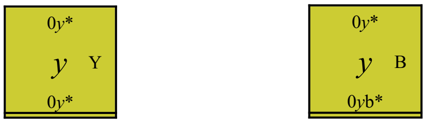
   

4. Tile types that copy the current value of the counter (right to left).

   (a) Rightmost tile type of copy row.

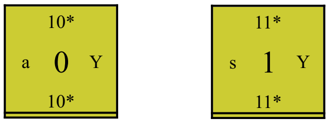
      i. Rightmost tile types that initiate a generic copy row - for all $y \in \{0, 1\}$, add the following tile types:

      Search for bit pattern: $(0 + 1)^{+}0(0 + 1)$ Copy row above bit pattern: $00^{+}1$

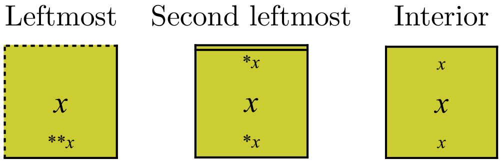
      

      ii. Rightmost tile types that search for a particular bit pattern - add the following tile types:

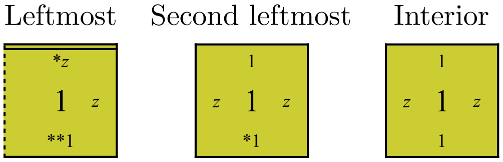
      Search for bit pattern: $11^{+}0$ Search for bit pattern: $11^{+}$

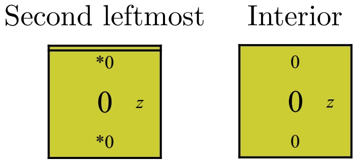
   (b) Copy row interior tile types - for all $x \in \{0, 1\}$, add the following tile types:



(c) Copy row tile types that perform bit pattern search - for all $z \in \{\mathrm{a}, \mathrm{s}\}$ add the following tile types:



(d) Tile types that terminate bit pattern search - for all $z \in \{\mathrm{a}, \mathrm{s}\}$ add the following tile types:



### 3.3.4 Overview Of The Planter Module

The next module is the planter module because it "plants the seeds" from which the ray modules will ultimately grow (the darkest gray squares in Figure 3.1). At the core of the planter module is a log-width, horizontal binary counter that counts every positive integer, starting at 1, in order. A key feature of the binary counter embedded in the planter module is that, after each integer is counted, a number of columns, equal to the current value of the binary counter, plus the number of bits in the binary representation of this value, plus a few extra "dummy" spacing columns, self-assemble. This has the effect of spacing out successive ray modules according to the function $f$ (given in the introduction).

### 3.3.5 Details Of Construction Of The Planter Module

The tile set for the planter consists of 94 tile types. We partition the tile types into four logical subgroups.

The first of the four subgroups is a standard binary counter that counts from 1 to infinity with the LSB of each integer having a $y$-coordinate of -1. The binary counter is based on an infinite fixed-width version of the binary counter in [45].

Suppose that $x \in \mathbb{Z}^+$ be the current value of the binary counter. The second subgroup receives $x$ as input, and then in each subsequent row, subtracts one from the value of the number in the previous row until reaching 0, at which time a final "dummy" row consisting of all zeros self-assembles. This results in the self-assembly of exactly $x + 1$ rows following the binary counter row that represents $x$. The tile types that perform subtraction are based on the optimal binary counter (see [13]). Note that while subtraction is taking place, the current value of the binary counter is "remembered" via the rightmost bits in the east/west edge labels so that the value can be input to the third subgroup. Figure 3.5 shows a detailed example of the subtraction process for $n = 4$.

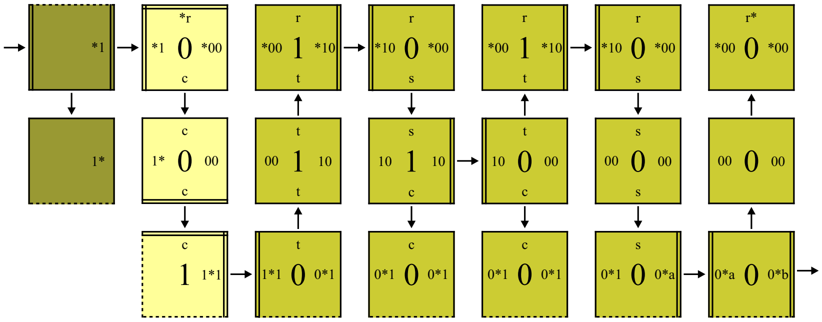


Figure 3.5: Subtraction in the planter. The first column of tiles represents the current value of the binary counter (4). The five rightmost columns of tiles perform subtraction.

Next, the third subgroup of tile types self-assembles into a square of size $\lfloor \lg x \rfloor + 1$ such that its north most edge labels are colored with the bits of $x$. The tile representing the LSB of $x$ is the one that is placed in the upper right corner of the square. Notice that this has the affect of rotating and reflecting $x$ up immediately below the $x$-axis. This is shown with detail in Figure 3.6. Notice that the rightmost tiles in Figure 3.5 bind with the leftmost tiles in Figure 3.6.

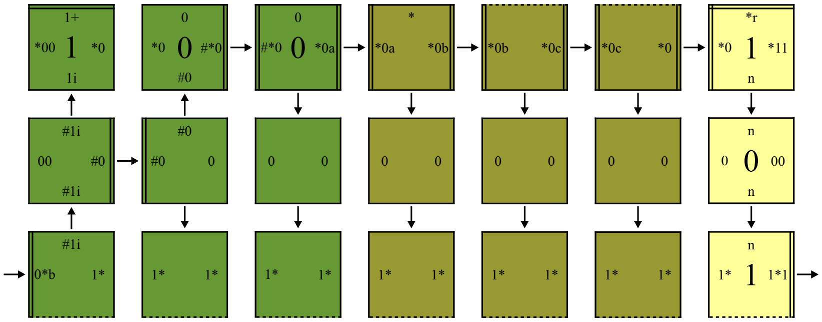
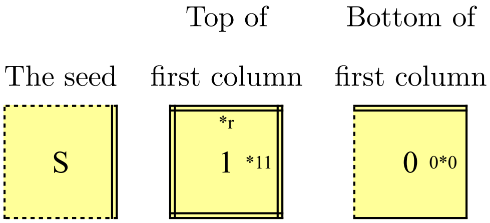


Figure 3.6: Rotation in the planter. The first three columns of tiles rotates and reflects the input (4) up immediately below the $x$-axis. The three subsequent columns of tiles are “dummy” spacing columns, and the final column of tiles represents the next value of the binary counter.

Finally, the fourth subgroup self-assembles three inert “dummy” spacing rows while allowing the current value of the binary counter to pass through. After these spacing rows attach, the next row of the binary counter self-assembles in which 1 is added to $x$. This process is repeated for all $x \in \mathbb{Z}^+$.

The following is the set of tile types that make up the planter module.

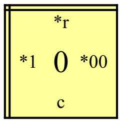
1. Seed tile types.



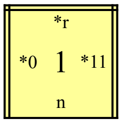
2. Binary counter tile types. The following tile types implement a fixed-width binary counter that counts every positive integer. Self-assembly proceeds naturally from LSB to MSB, which in our construction is top-to-bottom.

   (a) The first (topmost) tile type to attach (depending on whether the current row represents an even or odd integer) - add the following tile types:

Even integer



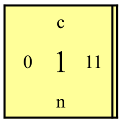
Odd integer



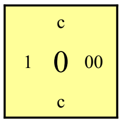
(b) Interior tile types that perform increments; The 'c' and 'n' characters symbolize situations in which there is either a carry bit coming in from the top or there is no carry bit.

i. If there is a carry bit coming in, add the following tile types:

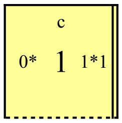
No carry bit out



Carry bit out

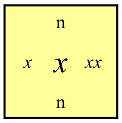


Bottom

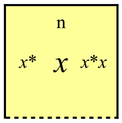


ii. If there is no carry bit coming in, then for all $x \in \{0, 1\}$, add the following tile types:

No carry bit out

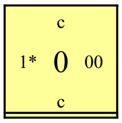


Bottom

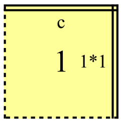


(c) Tile types that increase the number of bits in the binary counter - add the following tile types:

Propagate carry bit past border

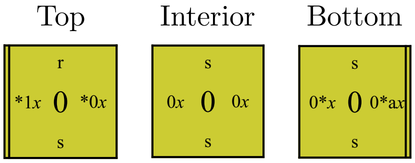


Increase width by 1 unit

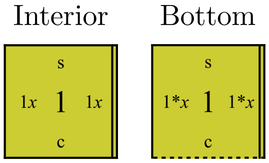


3. Subtraction tile types (a modification of the optimal binary counter given in [13]).

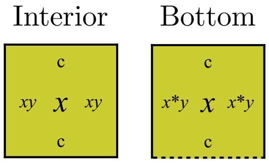
(a) Tile types that search for the least significant 1 bit in the current column - for all $x \in \{0, 1\}$, add the following tile types:



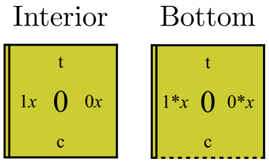
(b) Tile types that find the least significant 1 bit in the current column - for all $x \in \{0,1\}$, add the following tile types:



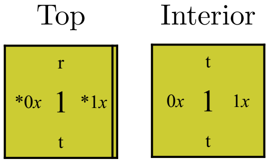
(c) Tile types that copy the least significant bits to the right (bottom) of the least significant 1 bit in the current column - for all $x, y \in \{0,1\}$, add the following tile types:



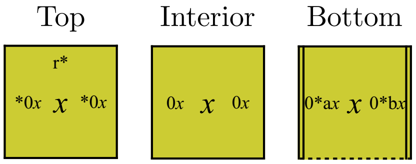
(d) The first tile type to attach in a row that represents an odd integer. Note that all tile types that attach above this tile type in a given column will represent the bit 1 (this is the purpose of the 't' signal) - for all $x \in \{0,1\}$, add the following tile types:



(e) Tile types that convert all the least significant bits to the right(top) of the least significant 1 (in the previous column) to 1 - for all $x \in \{0,1\}$, add the following tile types:



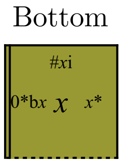
(f) After subtraction reaches 0, one final column is added - for all $x \in \{0, 1\}$, add the following tile types:



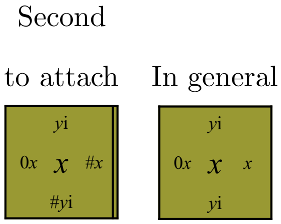
4. Rotation tile types.

(a) Tile types of the initial column.

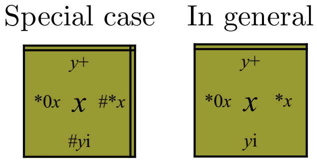
i. The first tile type - for all $x \in \{0, 1\}$, add the following tile types:



ii. Interior tile types - for all $x, y \in \{0, 1\}$, add the following tile types:



iii. Top most tile types - for all $x, y \in \{0, 1\}$, add the following tile types:

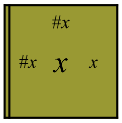


(b) Tile types that shift '#' up and to the right (not the initial or final column) while simultaneously copying the current value of the binary counter (the fourth subgroup) left-to-right, and bottom-to-top.

i. Shift right and then up - for all $x \in \{0, 1\}$, add the following tile types:

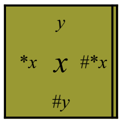
The first tile type

in a column

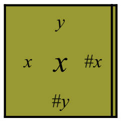


ii. Shift up and then right - for all $x, y \in \{0, 1\}$, add the following tile types:

Topmost

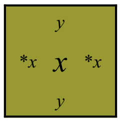


In general

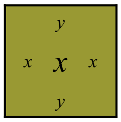


iii. Tile types that do not participate in shifting the '#' token but are above the main diagonal - for all $x, y \in \{0, 1\}$, add the following tile types:

Topmost



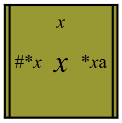
In general



(c) The first tile type to attach in the final column - for all $x \in \{0, 1\}$, add the following tile types:

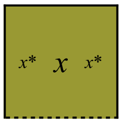
The first tile type

in a column

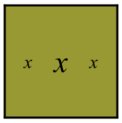


(d) Tile types that do not participate in shifting the '#' token but are below the main diagonal - for all $x \in \{0, 1\}$, add the following tile types:

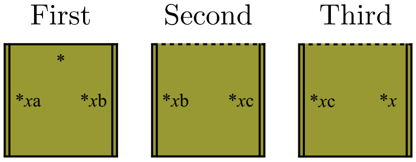
Bottom



In general



(e) Tile types that are responsible for the self-assembly of three inert spacing rows immediately after rotation - for all $x \in \{0, 1\}$, add the following tile types:



### 3.3.6 Overview Of The Computation Module

The final module is, for any TM $M$, an algorithmically generated tile set that, in conjunction with the ray and planter modules, achieves the simulation of $M$ on (the binary representation of) every input $x \in \mathbb{Z}^+$. The simulation of $M$ on $\text{bin}(x)$ proceeds vertically, immediately above the planter, while following the contour defined by the rightmost edge of the ray of width $x + 2$ (Note that by our construction of the planter module, there is one, and only one ray of such width). As with other standard Turing machine constructions (see [52, 45, 48]), each row in our simulation represents a configuration of $M$. However, the frequency with which transitions occur is a novel feature of our construction, and is controlled by "color" signals that are received from the abutting ray module.

### 3.3.7 Details Of Construction Of The Computation Module

Our simulation of $M$ on $x$ proceeds vertically while following the contour defined by the rightmost edge of the ray of width $x+2$. The right edge label of the rightmost tile type in every row of this ray sends a "color" signal that has the potential to initiate the self-assembly of a simulation row (if $M$ halts on $x$, then eventually there will be no simulation rows to influence). The color signal defines the type of simulation row that subsequently self-assembles, whence self-assembly of every simulation row (except "blue" rows) proceeds left-to-right. In fact, all non-blue simulation rows, more or less, simply copy the previous configuration of $M$ up one unit while, in some cases, also performing a shift-to-the-right by one unit.

Simulation begins with a row of tiles that represent the initial configuration of $M$ with input $x$, which is represented by a row of $\lfloor \lg x \rfloor + 2$ red tile types (see below). The initial simulation row is adjacent to the rightmost tile of the initial row of the ray of width $x+2$, and immediately above the $x$-axis. All subsequent simulation rows either copy the configuration of $M$ up to the next row (yellow and indigo rows) or do so while also shifting the contents to the right by one unit (orange rows). However, if a green color signal is received, then the simulation row increases the size of the working tape by a single tile (to the right), and then performs a single computation step in the next (blue) row. Note that when the size of the tape increases, the ray of width $x+2$ does not shift to the right, but the ray of width $x+3$ does. This relationship between successive rays maintains the constant width gap of 2 tiles between the simulation of $M$ on $x$ and the left border of the ray of width $x+3$.

If $M(x)\downarrow$, upward growth of the simulation halts, and a special signal is sent along a one-tile-wide path of tiles left-to-right along the top of the most recent simulation row. When the rightmost tile is encountered, the path continues down along the contour of the rightmost edge of the simulation until reaching the $x$-axis, at which point a black tile type is placed at the location $(f(x), 0)$. Since there is a constant gap (of width 2) between the right border of the simulation of $M$ on $x$ and the left border of the ray of width $x+3$, the halting signal proceeds unimpeded.

The following is the set of tiles that make up the simulation module.

Let $M = (Q, \Sigma, \Gamma, \text{-}, \delta, q_0, q_h)$ be a Turing machine where

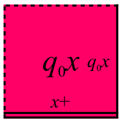
- $Q$ is a finite set of states;
- $\Sigma = \{0, 1\}$ is the input alphabet;
- $\Gamma$ is the tape alphabet;
- $\text{-} \in \Gamma - \Sigma$ is the blank symbol;
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$, is the transition function;
- $q_0 \in Q$ is the start state, and
- $q_h \in Q$ is the halting state.

We make the assumption that $M$ is initially in $q_0$ reading the leftmost symbol of its input $n \in \mathbb{N}$, which is in the leftmost cell on a one-way infinite tape. Further, we stipulate that $M$ never moves left when reading the leftmost symbol on its tape.

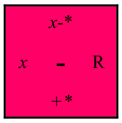
1. Red (seed) tile types.

   (a) For all $x \in \Gamma$, add the following tile types:

Leftmost

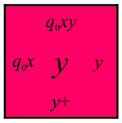


Rightmost

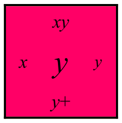


   (b) For all $x, y \in \Gamma$, add the following tile types:

Second leftmost



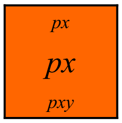
Interior



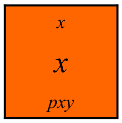
2. Orange (copy) tile types.

   (a) If the location contains, or is next to the tape head, then for all $x, y \in \Gamma$, and for all $p \in Q$, add the following tile types:

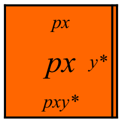
At tape head



Left of tape head

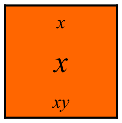


Left of rightmost (with tape head)

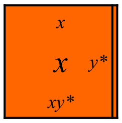


   (b) If the location is neither at or adjacent to the tape head, nor the rightmost location in the row, then for all $x, y \in \Gamma$, add the following tile types:

Copy (no tape head)

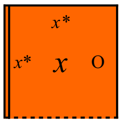


Left of rightmost



(c) Otherwise, for all $x \in \Gamma$, add the following tile types:

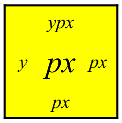
Rightmost



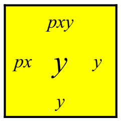
3. Yellow tile types (shift tape contents right one unit).

(a) For all $x, y \in \Gamma$, and for all $p \in Q$, add the following tile types:

Interior (with tape head)

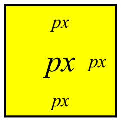


Right of tape head

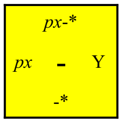


(b) For all $x \in \Gamma$, and for all $p \in Q$, add the following tile types:

Leftmost (with tape head)

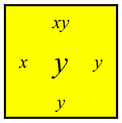


Rightmost and adjacent to tape head



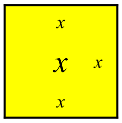
(c) For all $x, y \in \Gamma$, add the following tile types:

Interior

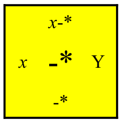


(d) For all $x \in \Gamma$, add the following tile types:

Leftmost (no tape head)



Rightmost

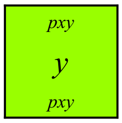


4. Green tile types (pre-transition):

(a) For all $x, y \in \Gamma$, and for all $p \in Q$, add the following tile types:

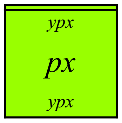
Interior

(left of tape head)



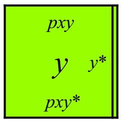
Interior

(with tape head)



Left of rightmost

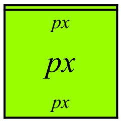
(right of tape head)



(b) For all $x \in \Gamma$, and for all $p \in Q$, add the following tile types:

Leftmost

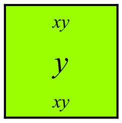
(with tape head)



(c) For all $x, y \in \Gamma$, add the following tile types:

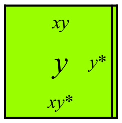
Interior

(no tape head)



Left of

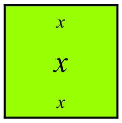
rightmost



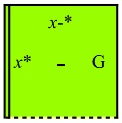
(d) For all $x \in \Gamma$, add the following tile types:

Leftmost

(no tape head)



Rightmost



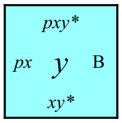
5. Blue tile types (Turing machine transition).

(a) Tile types that do not perform a transition.

i. For all $x, y \in \Gamma$, and for all $p \in Q$, add the following tile types:

Rightmost and

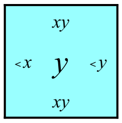
adjacent to tape head



ii. For all $x, y \in \Gamma$, add the following tile types:

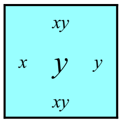
Interior

(left of tape head)



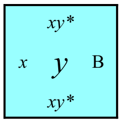
Interior

(right of tape head)



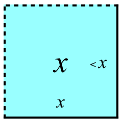
Rightmost and not

adjacent to to tape head



iii. For all $x \in \Gamma$, add the following tile types:

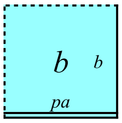
Leftmost



(b) Tile types that perform a right transition. If $\exists a, b \in \Gamma$ such that $(q, b, r) = \delta(p, a)$, then add the following tile types:

Interior and

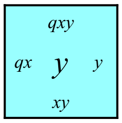
right of tape head



i. For all $x, y \in \Gamma$, add the following tile types:

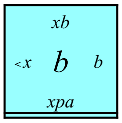
Interior and

right of tape head

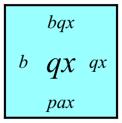


ii. For all $x \in \Gamma$, add the following tile types:

Start of transition at interior



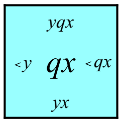
Right of tape head after transition



(c) Tile types that perform a left transition. If $\exists a, b \in \Gamma$ such that $(q, b, L) = \delta(p, a)$, then add the following tile types:

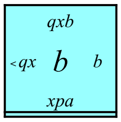
i. For all $x, y \in \Gamma$, add the following tile types:

Interior and left of tape head

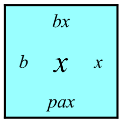


ii. For all $x \in \Gamma$, add the following tile types:

Start of transition at interior



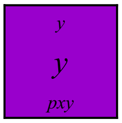
Right of transition



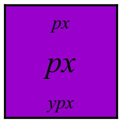
6. Indigo (copy tape contents straight up) tile types.

(a) For all $x, y \in \Gamma$, and for all $p \in Q$, add the following tile types:

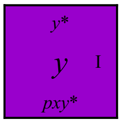
Interior (left of tape head)



Rightmost, and next to tape head



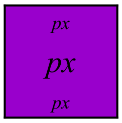
Rightmost, and not next to tape head



(b) For all $x \in \Gamma$, and for all $p \in Q$, add the following tile types:

Leftmost

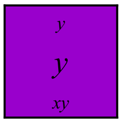
(with tape head)



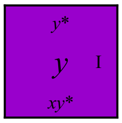
(c) For all $x, y \in \Gamma$, add the following tile types:

Interior without

tape head



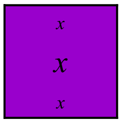
Rightmost



(d) For all $x \in \Gamma$, add the following tile types:

Leftmost

(without tape head)

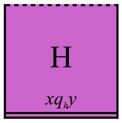


7. Violet (halting) tile types.

(a) For all $x, y \in \Gamma$, add the following tile types:

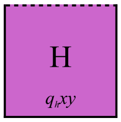
Initial interior

halting tile type



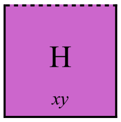
Right of initial

halting tile type



Fill in remainder

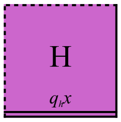
of final row



(b) For all $x \in \Gamma$, add the following tile types:

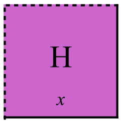
Initial halting

tile type (leftmost)



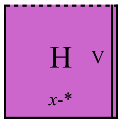
Leftmost tile type

(no tape head)



Second rightmost

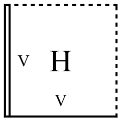
tile type



8. Tile types that snake down the rightmost edge of a halting simulation.

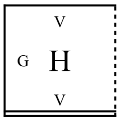
   (a) The first tile type to attach:

Rightmost of halting row

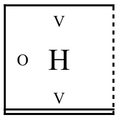


   (b) Grow down:

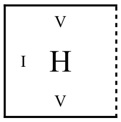
Green



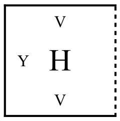
Orange



Indigo

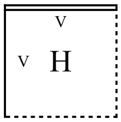


Yellow



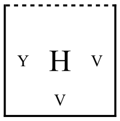
   (c) Initiate left growth:

Grow left

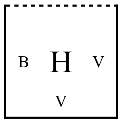


   (d) Grow left one unit:

Bump into yellow row



Bump into blue row



9. If $M$ halts on input $n$ then the following tile type (the only "black" tile type in our construction) is placed at the point $(f(n), 0)$:

Halting indicator

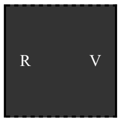


Figure 3.7 shows a trivial example of the simulation of a particular Turing machine $M$ on input 01 (the binary representation of 1, with a leading 0).

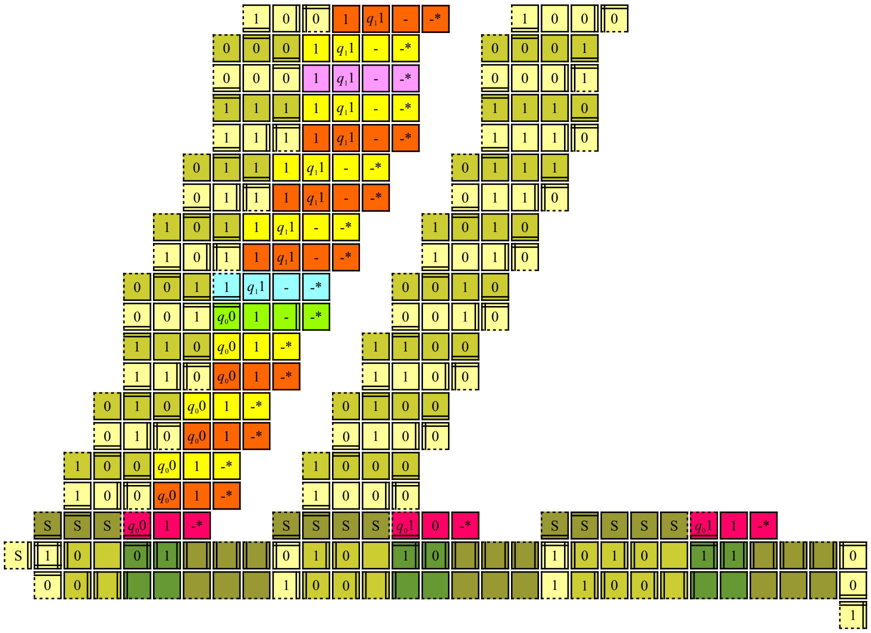

1 0 0 1 $q_1$1 - -* 1 0 0 0
0 0 0 1 $q_1$1 - -* 0 0 0 1
0 0 0 1 $q_1$1 - -* 0 0 0 1
1 1 1 1 $q_1$1 - -* 1 1 1 0
1 1 1 1 $q_1$1 - -* 1 1 1 0
0 1 1 1 $q_1$1 - -* 0 1 1 1
0 1 1 1 $q_1$1 - -* 0 1 1 0
1 0 1 1 $q_1$1 - -* 1 0 1 0
1 0 1 1 $q_1$1 - -* 1 0 1 0
0 0 1 1 $q_1$1 - -* 0 0 1 0
0 0 1 $q_0$0 1 - -* 0 0 1 0
1 1 0 $q_0$0 1 -* 1 1 0 0
1 1 0 $q_0$0 1 -* 1 1 0 0
0 1 0 $q_0$0 1 -* 0 1 0 0
0 1 0 $q_0$0 1 -* 0 1 0 0
1 0 0 $q_0$0 1 -* 1 0 0 0
1 0 0 $q_0$0 1 -* 1 0 0 0
S S S $q_0$0 1 -* S S S S $q_0$1 0 -* S S S S S $q_0$1 1 -*
S 1 0 0 1 0 1 0 1 0 1 0 1 1 0
0 0 1 0 0 1 1 0 0 0
1

Figure 3.7: A closer look at the simulation of $M$ on input 1 (influenced by a ray of width 3). All three modules can be seen in this figure.

### 3.3.8 Proof of Correctness

Let $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ be as in section 1, stipulating that $f(0) = -1$. For each $n \in \mathbb{N}$, define the rectangle

$$Q_n = \{f(n-1)+2, \ldots, f(n)-1\} \times \{-2,-1\},$$

and define the following assemblies.

(i) $\sigma_n$ is the portion of the planter lying in $Q_n$.

(ii) $\rho_n$ is the ray of width $n+2$, translated so that its base is the leftmost $2(n+2)$ tiles in $\sigma_n$.

(iii) $\sigma_n^* = \sigma_n + \rho_n$, where "+" is the operation defined in section 3.

(iv) $\gamma_n$ is the assembly that simulates $M(n)$ as in section 4.4.

(v) $\sigma^*$ is our planter.

(vi) $\alpha_n = \sigma_n^* + \gamma_n$.

(vii) $\alpha = \sigma^* + \sum_{n=0}^{\infty} \alpha_n$.

Let

$$\mathcal{T}_M = (T_M, \sigma, \tau)$$

be our tile assembly system, noting that $\sigma$ consists of a single tile at the origin. Define the following subsets of $\mathcal{T}_M$.

(i) $T_R$ is the set of tile types used in the ray module, together with the benign tile type, all of whose edges are given a binding strength of 1 with no color, appearing in the rectangles $\sigma_n$.

(ii) $T_C$ is the set of tile types in $T_R$, together with those occurring in the computation module.

(iii) $T_P$ is the set of tile types in the planter module.

For each $n \in \mathbb{Z}^+$, define the tile assembly systems

$$\mathcal{T}_{R,n} = (T_R, \sigma_n, \tau),$$

$$\mathcal{T}_{C,n} = (T_C, \sigma_n, \tau),$$

$$\mathcal{T}^*_{C,n} = (T_C, \sigma^*_n, \tau).$$

**Lemma 3.3.2.** For each $n \in \mathbb{Z}^+$, $\mathcal{A}_{\Box}[\mathcal{T}_{\mathcal{R},\backslash}] = \{\sigma^*_n\}$.

*Proof.* Define the infinite, $\tau$-$T_R$-assembly sequence, $\vec{\alpha}$, to be that which is implicit in Figures 3.2, 3.3, and 3.4. It is easy to see from our construction that every tile that binds in $\vec{\alpha}$ does so uniquely. Furthermore, every such tile addition occurs with exactly strength 2. It is clear that $\text{res}(\vec{\alpha})$ is terminal, whence $\vec{\alpha}$ is a locally deterministic assembly sequence. □

**Lemma 3.3.3.** For each $n \in \mathbb{Z}^+$, $\sigma^*_n$ is a pseudoseed of $\mathcal{T}_{C,n}$.

*Proof.* First, note that $\sigma^*_n \in \mathcal{A}[\mathcal{T}_{\mathcal{C},\backslash}]$ because $\sigma^*_n \in \mathcal{A}[\mathcal{T}_{\mathcal{R},\backslash}]$ and $T_R \subset T_C$.

Let $\alpha \in \mathcal{A}[\mathcal{T}_{\mathcal{C},\backslash}]$. Let $\vec{\alpha}'$ be the $\tau$-$T_C$-assembly sequence such that $\text{res}(\vec{\alpha}') = \alpha$. It is clear, since $\vec{\alpha}$ is locally deterministic and because of the use of module indicators in our construction, that when $\vec{\alpha}'$ assigns a tile type to a location in $\rho_n$, it does so in agreement with $\vec{\alpha}$ from Lemma 4. Thus, we can use $\vec{\alpha}$ to "extend" $\text{res}(\vec{\alpha}')$ and thereby fill in the remainder of $\rho_n$. This gives us an assembly $\alpha' \in \mathcal{A}[\mathcal{T}^*_{\mathcal{C},\backslash}]$ such that $\alpha \sqsubseteq \alpha'$. □

**Lemma 3.3.4.** For each $n \in \mathbb{Z}^+$, $\mathcal{A}_{\Box}[\mathcal{T}^*_{\mathcal{C},\backslash}] = \{\alpha_n\}$.

*Proof.* Define the $\tau$-$T_{C,n}$-assembly sequence $\vec{\alpha}_n$ as that which self-assembles each simulation row one at a time, and according to the following rule: if the current row is not "blue," then, by our construction, self-assembly must proceed left to right; however, if the current row is "blue," then $\vec{\alpha}_n$ first attaches the initial tile, via a single $\tau$-strength bond along its south edge, and then the remaining tiles in order of increasing distance from the origin. It is clear from our construction that every tile that binds in $\vec{\alpha}_n$ does so uniquely, and with exactly strength $\tau$. Since $\text{res}(\vec{\alpha}_n)$ is terminal, it follows that $\vec{\alpha}_n$ is locally deterministic. □

**Lemma 3.3.5.** For each $n \in \mathbb{Z}^+$, $\mathcal{A}_{\Box}[\mathcal{T}_{\mathcal{C},\backslash}] = \{\alpha_n\}$.

*Proof.* This follows immediately from Lemma 4, Lemma 5, and the pseudoseed lemma. □

Define the tile assembly system

$$\mathcal{T}_P = (T_P, \sigma, \tau).$$

**Lemma 3.3.6.** $\mathcal{A}_{\Box}[\mathcal{T}_{\mathcal{P}}] = \{\sigma^*\}$.

*Proof.* Define the infinite $\tau$-$T_P$-assembly sequence, $\vec{\alpha}$, to be that which self-assembles $\sigma^*$ one row at a time, and implicit from Figures 3.5 and 3.6. Note that $\vec{\alpha}$ starts from the seed tile located at the origin. It is clear from our construction that every tile that binds via $\vec{\alpha}$ does so uniquely. Furthermore, every such tile addition occurs with exactly strength 2. It is clear that $\text{res}(\vec{\alpha})$ is terminal, whence $\vec{\alpha}$ is a locally deterministic assembly sequence. □

**Lemma 3.3.7.** $\sigma^*$ is a pseudoseed of $\mathcal{T}_M$.

*Proof.* First, note that $\sigma^* \in \mathcal{A}[\mathcal{T}_{\mathcal{M}}]$ because $\sigma^* \in \mathcal{A}[\mathcal{T}_{\mathcal{P}}]$ and $T_P \subset T_M$.

Let $\alpha \in \mathcal{A}[\mathcal{T}_{\mathcal{M}}]$. Let $\vec{\alpha}'$ be the $\tau$-$T_M$-assembly sequence such that $\text{res}(\vec{\alpha}') = \alpha$. It is clear, since $\vec{\alpha}$ is locally deterministic and because of the use of module indicators in our construction, that when $\vec{\alpha}'$ assigns a tile to a location that is in $\sigma^*$, it does so in agreement with $\vec{\alpha}$ from Lemma 8. Thus, we can use $\vec{\alpha}$ to "extend" $\text{res}(\vec{\alpha}')$ and thereby fill in the remainder of $\sigma^*$. This gives us an assembly $\alpha' \in \mathcal{A}[\mathcal{T}^*_{\mathcal{C},\backslash}]$ such that $\alpha \sqsubseteq \alpha'$. □

**Lemma 3.3.8.** $\sigma^*$ is a $\vec{\sigma}$-$T_C$-$T_M$-multiseed.

*Proof.* Let $\vec{\sigma} = (\sigma_n \,|n \in \mathbb{Z}^+)$. By our construction of the planter, it is clear that part (1), of definition 3.2, is satisfied. To see that (2) is satisfied, we appeal to: Lemma 7, our construction of the ray, and the fact that the planter spaces out each $\sigma_n$ sufficiently. Moreover, it is clear that (5) holds because of our use of module indicator symbols in our construction. We now turn our attention to parts (3) and (4).

Let $i \in \mathbb{Z}^+$, and $\alpha \in \mathcal{A}[\mathcal{T}_\rangle]$. Let $\vec{\alpha}'$ be an assembly sequence such that $\alpha = \text{res}(\vec{\alpha}')$. Since (1) holds, we can define the assembly sequence $\vec{\alpha} = (\sigma^*, \ldots, \alpha)$, and let $\alpha^* = \text{res}(\vec{\alpha})$. It is clear that $\alpha^* \in \mathcal{A}[\mathcal{T}_\mathcal{M}^*]$, and $\alpha \sqsubseteq \alpha^*$, whence (3) holds.

Finally, let $\alpha^* \in \mathcal{A}[\mathcal{T}_\mathcal{M}]$. By (1), (2), Lemma 7, and our construction of the planter, it is easy to see that (4) holds.

□

**Lemma 3.3.9.** $\mathcal{A}_\Box[\mathcal{T}_\mathcal{M}] = \{\alpha\}$.

*Proof.* Let $\mathcal{T}_M^* = (T_M, \sigma^*, \tau)$. By Lemma 6, 7, and 9, and the multiseed lemma, $\mathcal{A}_\Box[\mathcal{T}_\mathcal{M}^*] = \{\alpha\}$. It follows by Lemma 8 and the pseudoseed lemma that $\mathcal{A}_\Box[\mathcal{T}_\mathcal{M}] = \{\alpha\}$. □

## 3.4 A Decidable Set That Does Not Self-Assemble

We now show that there are decidable sets $D \subseteq \mathbb{Z}^2$ that do not self-assemble in the Tile Assembly Model.

For each $r \in \mathbb{N}$, let

$$D_r = \{\, (m, n) \in \mathbb{Z}^2 \;\big|\; |m| + |n| = r \}.$$

This set is a "diamond" in $\mathbb{Z}^2$ (i.e., a circle in the taxicab metric) with radius $r$ and center at the origin. For each $A \subseteq \mathbb{N}$, let

$$D_A = \bigcup_{r \in A} D_r.$$

As illustrated in Figure 5.6a, this set is the "system of concentric diamonds" centered at the origin with radii in $A$.

The following trivial observation is used in the proof of Lemma 3.4.2.

**Observation 3.4.1.** $|D_{<r}| = \left|\left\{\, (m, n) \in \mathbb{Z}^2 \;\big|\; |m| + |n| < r \right\}\right| = 2(r-1)^2 + 2r - 1.$

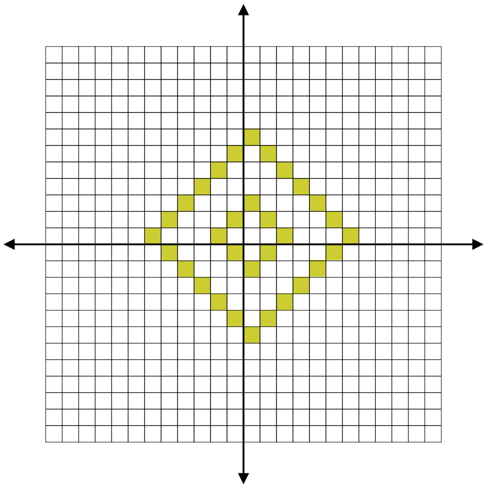


Figure 3.8: The set $D_{\{2,6\}}$

*Proof.*

$$\begin{aligned}
|D_{<r}| &= \left|\left\{ (m,n) \in \mathbb{Z}^2 \;\middle|\; |m| + |n| < r \right\}\right| \\
&= 2r - 1 + 2 \cdot \sum_{i=0}^{r-2} (2i+1) \\
&= 2r - 1 + 2(r-1)^2.
\end{aligned}$$

□

**Lemma 3.4.2.** If $A \subseteq \mathbb{N}$ and $D_A$ self-assembles, then $A \in \text{DTIME}\left(2^{4n}\right)$.

The proof of this lemma exploits the fact that a tile assembly system in which $D_A$ self-assembles must, for sufficiently large $r$, decide the condition $r \in A$ from *inside* the diamond $D_r$.

*Proof.* If $|A| < \infty$ then we are done, so assume otherwise (i.e., $A$ is infinite).

Assume the hypothesis. Then there exists a 2-TAS $\mathcal{T} = (T, \sigma, \tau)$ in which the set $D_A$ self-assembles. Fix some enumeration $\vec{a}_1, \vec{a}_2, \vec{a}_3 \ldots$ of $\mathbb{Z}^2$, and let $M$ be the TM, defined as follows, that simulates the self-assembly of $\mathcal{T}$.

```
Require: $r \in \mathbb{N}$
  $\alpha := \sigma$
  while $D_r \cap \mathrm{dom}\ \alpha = \varnothing$ do
    choose the least $j \in \mathbb{N}$ such that $\vec{a}_j \in \partial\alpha_i$
    choose $t \in T$ such that $\vec{a}_j \in \partial_t \alpha_i$
    $\alpha := \alpha + (\vec{a}_j \mapsto t)$
  end while
  if $\alpha\left(\vec{a}_j\right) \in B$ then
    accept
  else
    reject
  end if
```

Observe that the *while* loop will terminate after at most $|D_{<r}| + 1 - |\mathrm{dom}\ \alpha|$ iterations, with $D_r \cap \mathrm{dom}\ \alpha \neq \varnothing$ whence $M$ halts on all inputs. If $r \in A$, then the weak self-assembly of $D_A$ tells us that for every $\alpha \in \mathcal{A}_{\Box}[\mathcal{T}]$, $\alpha\left(D_r\right) \subseteq B$. Since we have $\vec{a}_j \in D_r$ when the *while* loop terminates, $M$ accepts, and $r \in L(M)$. If $r \in L(M)$, then $\alpha\left(\vec{a}_j\right) \in B$, whence $r \in A$.

By Observation 3.4.1, the *while* loop performs at most $2(r-1)^2 + 2r - 1$ iterations, and in each iteration, we are doing at most $O\left(r^2\right)$ amount of additional work checking $D_r \cap \mathrm{dom}\ \alpha = \varnothing$, and maintaining $\partial^\tau \alpha_i$. Thus, $M$ decides $A$ in $O\left(2^{4n}\right)$ computation steps, where $n = \lfloor \log r \rfloor + 1$ (i.e., the length of $r$).

Note: Implicit in this proof is the fact, proven by Irani, Naor, and Rubinfeld [27], that write-once memory is equivalent to time. □

We now have the following result.

**Theorem 3.4.3** (second main theorem)**.** There is a decidable set $D \subseteq \mathbb{Z}^2$ that does not self-assemble.

*Proof.* By the time hierarchy theorem [26], there is a set $A \subseteq \mathbb{N}$ such that

$$A \in \mathrm{DTIME}\left(2^{5n}\right) - \mathrm{DTIME}\left(2^{4n}\right).$$

Let $D = D_A$. Then $D$ is decidable and, by Lemma 3.4.2, $D$ does not self-assemble. □

Note that both Lemma 3.4.2 and (hence) Theorem 3.4.3 hold for *any* value of $\tau > 0$.

## 3.5 Conclusion

Our first main theorem says that, for every computably enumerable set $A \subseteq \mathbb{Z}^+$, the representation $X_A = \{(f(n), 0) | n \in A\}$ self-assembles. This representation of $A$ is somewhat sparse along the $x$-axis, because our $f$ grows quadratically. A linear function $f$ would give a more compact representation of $A$. We conjecture that our first main theorem does *not* hold for any linear function, but we do not know how to prove this.

Let $D$ be the set presented in the proof of our second main theorem. It is easy to see that the condition $(m, n) \in D$ is decidable in time polynomial in $|m| + |n|$, but $|m| + |n|$ is exponential in the length of the binary representation of $(m, n)$, so this only tells us that $D \in \mathrm{E} = \mathrm{DTIME}\left(2^{\mathrm{linear}}\right)$. Is there a set $D \subseteq \mathbb{Z}^2$ such that $D \in \mathrm{P}$, and $D$ does not self-assemble?

More generally, we hope that our results lead to further research illuminating the interplay between geometry and computation in self-assembly.

# CHAPTER 4. Self-Assembly of Decidable Sets

The work in this chapter was performed with co-author Scott M. Summers. It has been published as [41] and [39].

## 4.1 Introduction

In this chapter, we explore the notion of *computation* in the TAM–what is it, and how is it accomplished? Despite its deliberate over-simplification, the TAM is a computationally expressive model. For instance, Winfree proved [52] that in two or more spatial dimensions, the TAM is capable of Turing-universal computation at temperature 2. On the other hand, Adleman, Kari, Kari, Reishus, and Sosík [4] established that the TAM is universal (in two spatial dimensions) with respect to *non-deterministic* temperature 1 tile assembly systems. Doty, Patitz and Summers then conjectured [21] that any temperature 1 tile assembly system that produces a unique 2-dimensional terminal assembly must necessarily produce a "computationally very simple" shape, i.e., that the TAM is *not* Turing universal at temperature 1 with respect to deterministic self-assembly in two spatial dimensions. In a recent result, Fu and Schweller [24] proved that the TAM is in fact Turing universal at temperature 1 in three spatial dimensions with respect to deterministic self-assembly systems.

Note that the universality of the TAM implies that it is possible to construct, for any Turing machine $M$ and any input string $w$, a finite assembly system (i.e., finite set of tile types) that tiles the first quadrant and encodes the set of all configurations that $M$ goes through when processing the input string $w$. In other words, the process of self-assembly can (1) be directed algorithmically, and (2) be used to evaluate computable functions.

One can also regard the process of self-assembly itself as computation that takes as input an initial configuration of tiles (usually taken to be taken a single tile) and produces output in the form of some particular connected shape, and *nothing* else (i.e., *strict* self-assembly [36]). The self-assembly of shapes, and their associated Kolmogorov (shape) complexity, was studied extensively by Soloveichik and Winfree in [48], where they proved the counter-intuitive fact that sometimes fewer tile types are required to self-assemble a "scaled-up" version of a particular shape than the actual un-scaled shape.

Another flavor of computation in the TAM is the self-assembly of a computationally interesting set (or pattern) on top of a much larger possibly "less interesting" set that is used for auxiliary computations (so-called *weak* self-assembly). We say that a set of points $A$ weakly self-assembles if there is a finite tile assembly system that places "black" tiles on, and only on, the points that are in $A$. One can also view weak self-assembly as the process of a tile system "painting" a picture of the set $A$ onto a much larger canvas of tiles. It is clear that if $A$ weakly self-assembles, then $A$ is necessarily computably enumerable. Moreover, Lathrop, Lutz, Patitz and Summers [34] discovered that the converse of the previous statement holds in the following sense. If the set $A$ is computably enumerable, then a "simple" two-dimensional representation of $A$, as points along the $x$-axis, weakly self-assembles. This result is interesting because its proof requires the simulation of a particular Turing machine $M$ on *infinitely* many inputs (i.e., for all $x \in \mathbb{N}$) in the two-dimensional discrete Euclidean plane $\mathbb{Z}^2$.

In this chapter, we continue the work of Lathrop, Lutz, Patitz and Summers [34]. Specifically, we focus our attention on the weak self-assembly of canonical two-dimensional representations of decidable sets in the TAM. We first reproduce Winfree's proof of the universality of the TAM [52] in the form of a simple construction called the "wedge construction." The wedge construction self-assembles the *computation history* of an arbitrary TM $M$ on input $w$ in the space to the right of the $y$-axis, above the $x$-axis, and above the line $y = x - |w| - 2$. We then prove our first main result, which follows from a straight-forward extension of the wedge construction and gives a new characterization of decidable languages of natural numbers in terms of (weak) self-assembly. That is, we prove that a set $A \subseteq \mathbb{N}$ is decidable if and only if $A \times \{0\}$ and $A^c \times \{0\}$ weakly self-assemble. Technically speaking, our characterization is (exactly) the first main theorem from Lathrop, Lutz, Patitz and Summers [34] with "computably enumerable" replaced by "decidable," and $f(n) = n$. Finally, we show that our construction is robust with respect to certain types of geometrical constraints, i.e., our characterization can be carried out in the set of integer lattice points lying *above* the $x$-axis yet *below* the line $y = \frac{1}{a} \cdot x$, for any $a \in \mathbb{N}$.

## 4.2 The Wedge Construction

In this section, we review the "wedge construction" - a simple, well-known technique used to carry out the simulation of an arbitrary Turing machine on some binary string in the first quadrant of the discrete Euclidean plane. We will later extend the wedge construction to prove a new characterization of decidable languages.

**Lemma 4.2.1** (The wedge construction)**.** For every single-tape Turing machine $M$ and input $w \in \{0,1\}^*$, there exists a tile assembly system $\mathcal{T}_{M(w)}$, which simulates $M$ on $w$ in the following way.

1. $\mathcal{T}_{M(w)}$ simulates the computation of $M(w)$, with the configuration of $M(w)$ after $n$ steps represented by the line $y = n$ in the terminal assembly of $\mathcal{T}_{M(w)}$,

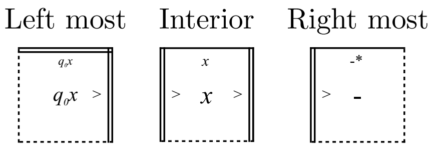
2. if $M$ halts on $w$ after $k$ steps, then the line $y = k + 1$ in the terminal assembly of $\mathcal{T}_{M(w)}$ contains one and only one "halting" tile that binds via a single strength-2 bond on its south edge, and

3. $\mathcal{T}_{M(w)}$ is locally deterministic, and therefore directed.

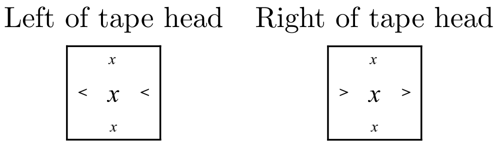
*Proof.* Our proof is by construction. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ be a Turing machine and $w \in \{0,1\}^*$. Assume, without loss of generality, that $M$ is a Turing machine having a one-way infinite-to-the-right tape such that the tape head of $M$ never attempts to move left while reading the left most tape cell. We define the finite set of tile types $T_{M(w)}$ as follows.

**Definition of $T_{M(w)}$:**

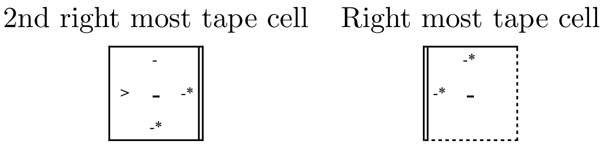
1. For all $x \in \Gamma$, add the seed row tile types:



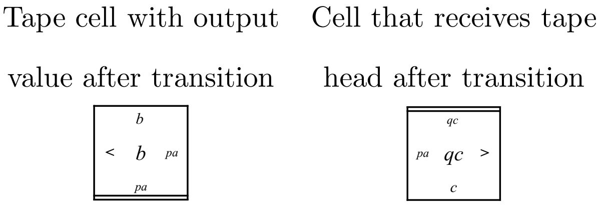
2. For all $x \in \Gamma$, add the tile types:



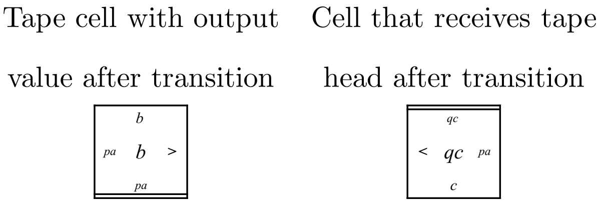
3. Add the following two tile types that grow the tape to the right:



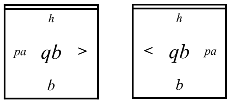
4. For all $p, q \in Q$, and all $a, b, c \in \Gamma$ satisfying $(q, b, \mathrm{R}) = \delta(p, a)$ and $q \notin \{q_{\mathrm{accept}}, q_{\mathrm{reject}}\}$ (i.e. for each transition moving the tape head to the right into a non-accepting state), add the tile types:



5. For all $p, q \in Q$, and all $a, b, c \in \Gamma$ satisfying $(q, b, \mathrm{L}) = \delta(p, a)$ and $q \notin \{q_{\mathrm{accept}}, q_{\mathrm{reject}}\}$ (i.e. for each transition moving the tape head to the left into a non-accepting state), add the tile types:



6. For all $p \in Q$, $a, b \in \Gamma$, and all $h \in \{\text{reject}, \text{accept}\}$ satisfying $\delta(q,b) \in \{q_{\text{accept}}, q_{\text{reject}}\} \times \Gamma \times \{\text{L}, \text{R}\}$, with $h = \text{accept}$ if $\delta(q,b) = q_{\text{accept}}$ and $h = \text{reject}$ otherwise (i.e. for each transition moving the tape head into a halting state), add the tile types:



**Definition of $\sigma_w$:** We now define the finite seed assembly $\sigma_w$ of $\mathcal{T}_{M(w)}$. Let $s_{\text{left}}$, $s_{\text{interior}}$ and $s_{\text{right}}$ be the "Left most," "Interior," and "Right most" tile types, respectively, defined above in the first group of "seed row" tile types. Define $\sigma_w$ as follows. $\sigma_w(0,0) = s_{\text{left}}$, where each occurrence of $x$ in $s_{\text{left}}$ is replaced by $w[0]$ (i.e., the first bit of $w$); for all $0 < i < |w|$, $\sigma_w(i,0) = s_{\text{interior}}$, with each occurrence of $x$ in $s_{\text{interior}}$ replaced by $w[i]$; let $\sigma_w(0,|w|) = s_{\text{right}}$; finally, let $\sigma_w$ be undefined at all other points in $\mathbb{Z}^2$.

Note that $\mathcal{T}_{M(w)}$ satisfies property (1) of the conclusion of the lemma–this can be easily verified from the above definition of $T_{M(w)}$ and is intuitively portrayed in Figure 4.1. We now argue that $\mathcal{T}_{M(w)}$ is locally deterministic. To do so, we first define an assembly sequence $\vec{\alpha}$, leading to a terminal assembly $\alpha = \text{res}(\vec{\alpha})$, in which (1) the $j^{\text{th}}$ configuration $C_j$ of $M$ is encoded in the row $R_j = (\{0, \ldots, |w| - 1 + j\} \times \{j\})$, and (2) $\vec{\alpha}$ self-assembles $C_i$ in its entirety before $C_j$ if $i < j$. By the way we defined the tile types of $T_{M(w)}$, it is easy to see that every tile that binds in $\vec{\alpha}$ does so deterministically, and with exactly strength-2 (either one strength-2 bond or two strength-1 bonds), whence $\mathcal{T}_{M(w)}$ is locally deterministic. The lemma follows by the addition of two tile types, each having strength-2 glues on their south edges, labeled with "accept" and "reject" respectively. □

The above "wedge" construction can be used to prove the following undecidability result.

**Corollary 4.2.2.** The language defined as $A = \{\mathcal{T} \mid \mathcal{T} \text{ is a TAS such that } \mathcal{T} \text{ is locally deterministic}\}$ is undecidable.

*Proof.* We will prove a stronger result: $A$ is $\Pi^1_0$-complete (recall that the set $\Pi^1_0$ is the set of

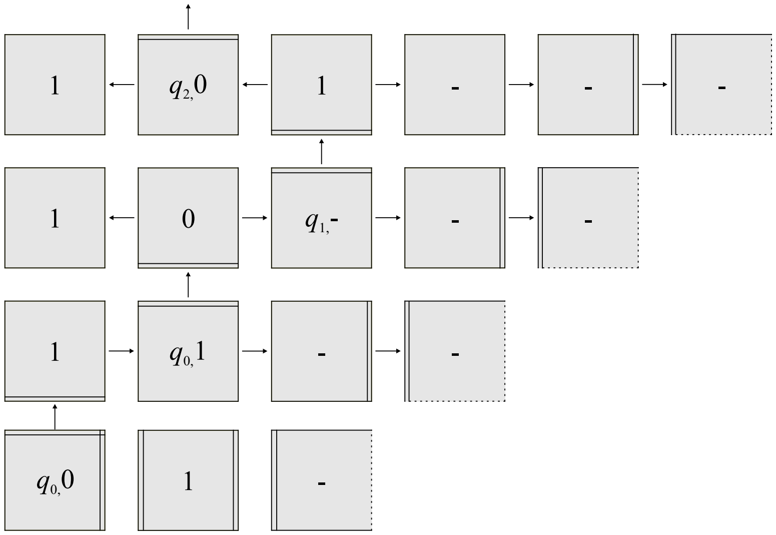


Figure 4.1: Example of the first four rows of a sample wedge construction which is simulating a Turing machine $M$ on the input string '01.'

all computably enumerable languages). To see that $A \in \Pi^1_0$, first note that

$$A = \left\{ \mathcal{T} \;\middle|\; \begin{array}{l} \text{for each } n \in \mathbb{N}, \vec{\alpha} \text{ is an assembly sequence in } \mathcal{T} \text{ of length } n, \\ \text{and } \vec{\alpha} \text{ satisfies the first two conditions of local determinism} \end{array} \right\}.$$

Checking whether a finite assembly sequence in some tile assembly sequence satisfies the first two conditions of local determinism is a decidable condition, whence $A \in \Pi^1_0$.

Next, to show that $A$ is $\Pi^1_0$-hard, we will exhibit a many-one reduction from the complement of the halting problem $H^c$ to $A$. Our reduction $F$ takes as input a Turing machine $M$ and a binary string $w \in \{0,1\}^*$, and outputs the tile assembly system $\mathcal{T}_{M(w)}$ modified as follows: each "final halting tile type" is replaced by two distinct tile types that each have a strength-2 glue on their south edge but share the same glue label (either "accept" or "reject"). This clearly breaks the second condition of local determinism! Note that we could also modify $\mathcal{T}_{M(w)}$ such that the final halting tiles bind with strength $3 > \tau = 2$, thus breaking the first condition of local determinism. It is clear that $F$ is a reduction, seeing as how if $M$ never halts on $w$, then $\mathcal{T}_{M(w)}$ remains locally deterministic (because no halting tiles ever attach). However, if $M$ halts on $w$, the one of the newly added halting tiles will non-deterministically attach breaking the local determinism of $\mathcal{T}_{M(w)}$. □

## 4.3 A New Characterization of Decidable Languages

We now turn our attention to the self-assembly of decidable sets of positive integers. We will extend the wedge construction from the previous section in order to prove that, for every decidable set $\{0\} \times -A$, there exists a directed TAS $\mathcal{T}_A = (T_A, \sigma, \tau)$ in which $\{0\} \times -A$ weakly self-assembles. Our proof relies on the following observation.

**Observation 4.3.1.** If $A \subseteq \mathbb{N}$ is a decidable set, then there is a TM $M$, such that for every $w \in A$, $M$ halts on $w$.

This means that we can essentially "stack" wedge constructions one on top of the other. Intuitively, our main construction is the "self-assembly version" of the following enumerator.

```
while $1 \leq w < \infty$ do
  simulate $M$ on the binary representation of $w$
  if $M$ accepts then
    output 1
  else
    output 0
  end if
  $w := w + 1$
end while
```

Just as the above enumerator prints the characteristic sequence of $A$, our construction will self-assemble a canonical two-dimensional representation of the characteristic sequence of $A$ as points along the negative $y$-axis.

### 4.3.1 Main Construction: Self-Assembly of 2-Dimensional Representations of Decidable Languages

In this section we present the full construction of the tile assembly system $\mathcal{T}_A$, and in the next section we provide a higher-level description of the behavior of our tile system. Note that we used the Tile Set Designer graphical interface to the TAM DSL [19] to design the tile set for this construction and the ISU TAS simulator [38] to simulate it. Both software packages are available for download from `http://www.cs.iastate.edu/~lnsa`.

**Lemma 4.3.2** (Main construction)**.** Let $A \subseteq \mathbb{N}$ be a decidable set. There exists a directed, singly-seeded, tile assembly system $\mathcal{T}_A = (T_A, \sigma, 2)$ in which the set $\{0\} \times -A$ weakly self-assembles.

*Proof.* Our proof is by construction. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ be a Turing machine with blank symbol '-' $\in \Gamma$ and $L(M) = A$. Assume, without loss of generality, that $M$ is a total Turing machine having a one-way infinite-to-the-right tape such that the tape head of $M$ never attempts to move left while reading the left most tape cell. We give the full specification of $T_A$ and $\sigma$ below.

**Simulation tiles:** Throughout our construction of simulation tiles, every tile takes as input, and ultimately outputs, six pieces of information along its south and north edges, respectively:

1. The symbol stored in the tape cell represented by this tile,
2. the current state of the Turing machine that is being simulated,
3. the direction of movement of the tape head,
4. the value of the bit that is embedded into this tile,
5. the significance of the aforementioned bit, and
6. a miscellaneous signal.

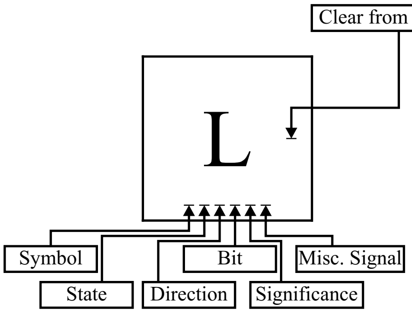
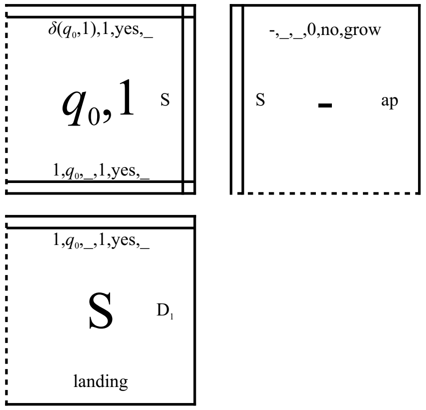


Figure 4.2: Each tile that contributes to the simulation of the Turing machine takes as input six pieces of information. We omit discussion of the east and west glue labels for our tile types because the type of information that is being taken as input and produced as output is always clear from the context.

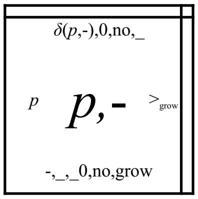
This situation is illustrated in Figure 4.2.

Note that some or all of these six pieces of information might not be relevant at certain times during the assembly process. Therefore, we use underscores (i.e., "place holder" values) to denote the absence of values of some of the signals when they are not required. The following list of tile types encode the logic of the Turing machine $M$ that decides $A$.

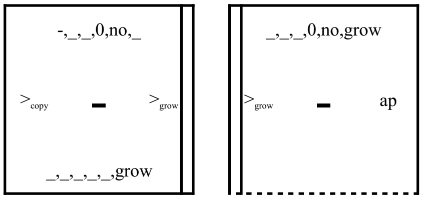
1. The following tile types appear only near the seed tile type (the tile having the 'S' label).



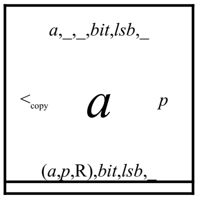
2. The following tile type only appears near the seed and receives the tape head from the left. For all $p \in Q - \{q_{\text{accept}}, q_{\text{reject}}\}$, add the following tile type.



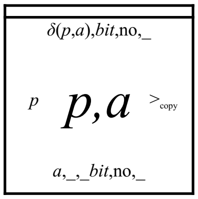
3. The following tile types "grow" the tape one cell to the right.



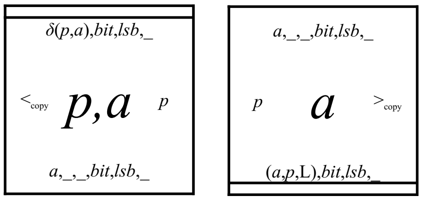
4. The following tile types move the tape head right. For all For all $p \in Q - \{q_{\text{accept}}, q_{\text{reject}}\}$, $a \in \Gamma$, $bit \in \{0, 1\}$, and $lsb \in \{\text{yes}, \text{no}\}$, add the following tile types. As noted above, the three pieces of information passed upward are the current symbol in this particular tape cell, and the current value of the bit of this column along with its significance.



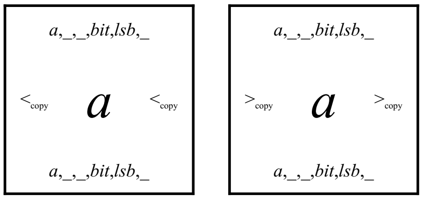
5. The following tile type receives the tape head from the left. For all For all $p \in Q - \{q_{\text{accept}}, q_{\text{reject}}\}$, $a \in \Gamma$, and $bit \in \{0, 1\}$, add the following tile types.



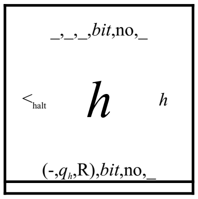
6. The following tile types receive the tape head from the right and move the tape head left (respectively). For all For all $p \in Q - \{q_{\text{accept}}, q_{\text{reject}}\}$, $a \in \Gamma$, $bit \in \{0, 1\}$, and $lsb \in \{\text{yes}, \text{no}\}$, add the following tile types.



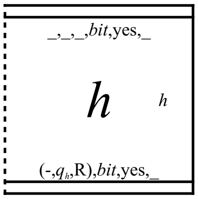
7. The following tile types copy the contents of the tape to the left and right of the tape head up to the next row (respectively). For all $a \in \Gamma$, $bit \in \{0, 1\}$, $lsb \in \{\text{yes}, \text{no}\}$, add the following tile types.



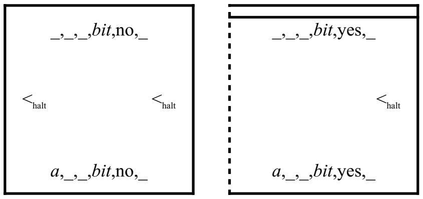
8. The following tile type halts the Turing machine when the tape head is not reading the left most tape cell. For all $h \in \{\text{accept}, \text{reject}\}$, and $bit \in \{0, 1\}$, add the following tile type.



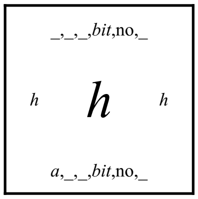
9. The following tile type halts the Turing machine when the tape head *is* reading the left most tape cell. For all $h \in \{\text{accept}, \text{reject}\}$, and $bit \in \{0, 1\}$, add the following tile type.



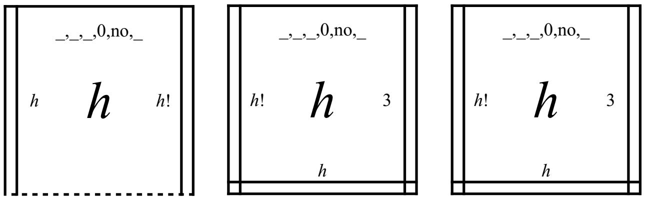
10. The following tile types search (to the left of the tape head) for the left most tape cell after halting. For all $a \in \Gamma$, and $bit \in \{0, 1\}$, add the following tile types.



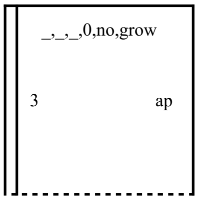
11. The following tile type transfers the halting state (either accept or reject) to the right enroute to the negative $y$-axis. For all $a \in \Gamma$ and $bit \in \{0, 1\}$, add the following tile type.



12. The following tile types prepare to send the one-tile-wide path containing the halting state (either accept or reject) down to the negative $y$-axis. For all $h \in \{q_{\text{accept}}, q_{\text{reject}}\}$, add the following tile types.



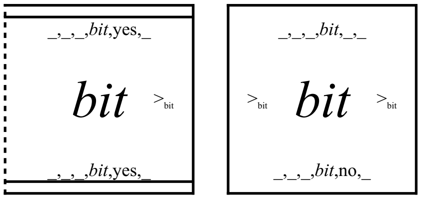
13. The following tile type is the right most tile to attach in any halting row. Add the following tile type.



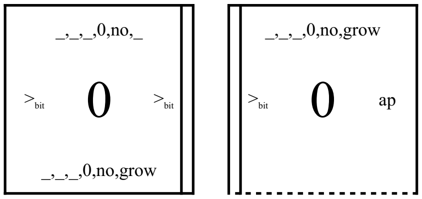
The following tile types "extract" the bits that are embedded within each simulation of the Turing machine. This is the final step before the count is incremented by one and used as input to the next simulation.

1. The following tile types initiate and carry out the bit extraction procedure starting from the least significant bit and working toward the right edge of the previous row

of the assembly. For all $bit \in \{0,1\}$, add the following tile types.

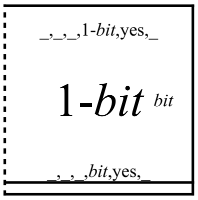


2. The following tile types extract the final two bits. The tile type on the left extracts the final bit of the previous row, and the tile type on the right side adds a dummy bit to maintain the geometry of the right most edge of the assembly. Add the following tile types.

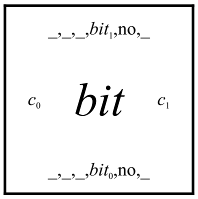


The following tile types self-assemble on top of the bit-extraction row and increments the value of these bits by 1.

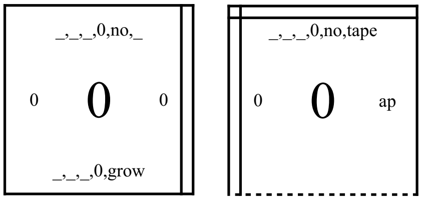
1. The following tile type initiates the increment process starting from the least significant bit. For all $bit \in \{0,1\}$, add the following tile type.



2. The following tile type performs the bulk of the increment procedure. For all $bit_0 \in \{0,1\}$ and $c_0 \in \{0,1\}$, add the following tile type, where $c_1 = \frac{\lfloor bit_0+c_0 \rfloor}{2}$ and $bit_1 = (bit_0 + c_0) \mod 2$.

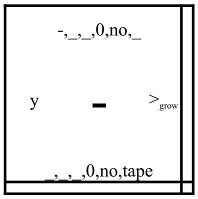


3. The following tile types are the final two tile types to attach in any increment row. Since the right edge of the construction grows faster than the length of the binary integers, the bits will always be 0. Add the following tile types.

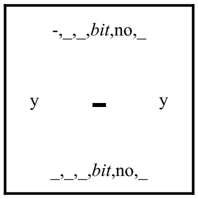


The following list of tile types build the initial configuration of the “next” simulation of $M$.

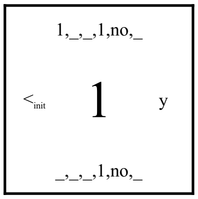
1. The following tile type assembles the right most tape cell (it always contains a blank) with a 0 bit embedded in it. Note that the ‘y’ signal is used to search for the right most 1 bit in the previous row. Add the following tile type



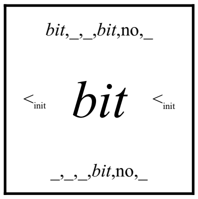
2. The following tile type continues the search (via the ‘y’ signal) for the right most 1 bit in the previous row. While doing so, blank symbols are encoded in the tape and 0 bits are also embedded. For all *bit* $\in \{0,1\}$, add the following tile type.



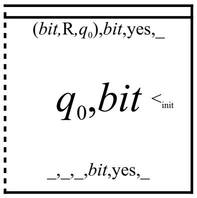
3. The following tile type attaches directly above the right most 1 bit in the previous row. In this case, the 1 bit is encoded in the tape and the embedded bit is 1. Note that this never occurs in the least significant bit. We terminate our search by changing the ‘y’ signal to $<_{\text{init}}$. Add the following tile types.



4. The following tile type initializes the contents of the tape to the left of the right most 1 bit. Here, we simply encode each cell with the appropriate bit from the previous row. Add the following tile type.

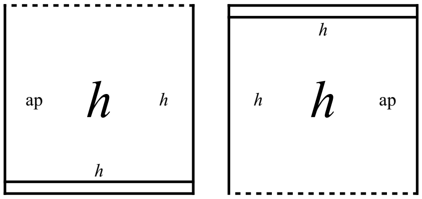


5. This tile type starts the next simulation of $M$. For all $bit \in \{0, 1\}$, add the following tile type.

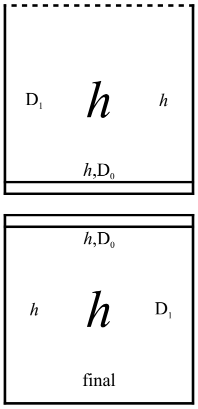


**"Decision path" tiles:** All of the previous tile types contribute to the simulation of $M$ on every input $x \in \mathbb{N}$. We complete our construction by adding the tile types that self-assemble one-tile-wide "decision paths" that result in the placement of black (accept) or non-black (reject) tiles on the negative $y$-axis.

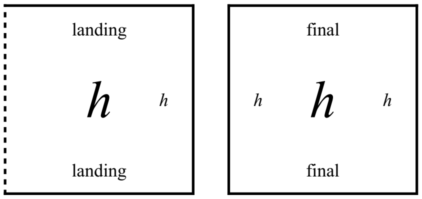
1. The following tile types initiate the one-tile-wide "decision path." This path starts from the right edge of a halting row (see above tile types) and carries the answer to the question, "did the Turing machine accept or reject?" Note that the geometry of the existing assembly guides the path down to the appropriate point on the negative $y$-axis. For each $h \in \{\text{accept}, \text{reject}\}$, add the following tile types.



2. The following tile types allow the decision paths to “turn” the corner and thus head straight for the appropriate point on the negative $y$-axis. We use the $\mathrm{D}_0$ and $\mathrm{D}_1$ signals to accomplish this task. For each $h \in \{\text{accept}, \text{reject}\}$, add the following tile types.



3. The following tile types assemble the final segment of each decision path. The tile type that is placed on the negative $y$-axis is the left most tile below. For each $h \in \{\text{accept}, \text{reject}\}$, add the following tile types.



Let $T_A$ be the set of *all* of the tile types that are defined above. Let $\sigma$ be the seed assembly consisting of the unique tile type having the label ‘S’ placed at the origin and undefined at every other point $\vec{0} \neq \vec{x} \in \mathbb{Z}^2$. Finally, define the tile assembly system $\mathcal{T}_A = (T_A, \sigma, 2)$. A routine local determinism argument can be used to show that $\mathcal{T}_A$ is locally deterministic and therefore directed.

Choosing the set $B$ (a.k.a., the set of "black" tiles) to be the singleton set containing the left most tile type in the last pair of tile types defined above, where $h =$ accept, proves that $\{0\} \times -A$ weakly self-assembles. $\square$

### 4.3.2 Discussion of Proof of Lemma 4.3.2

This section gives a high-level, intuitive description of the constructive proof of Lemma 4.3.2. Note that $\mathcal{T}_A$ is a singly-seeded, two-dimensional temperature 2 tile assembly system. The seed tile type is the unique tile type having a label of 'S.' We place the seed tile at the point $(0, 0)$.

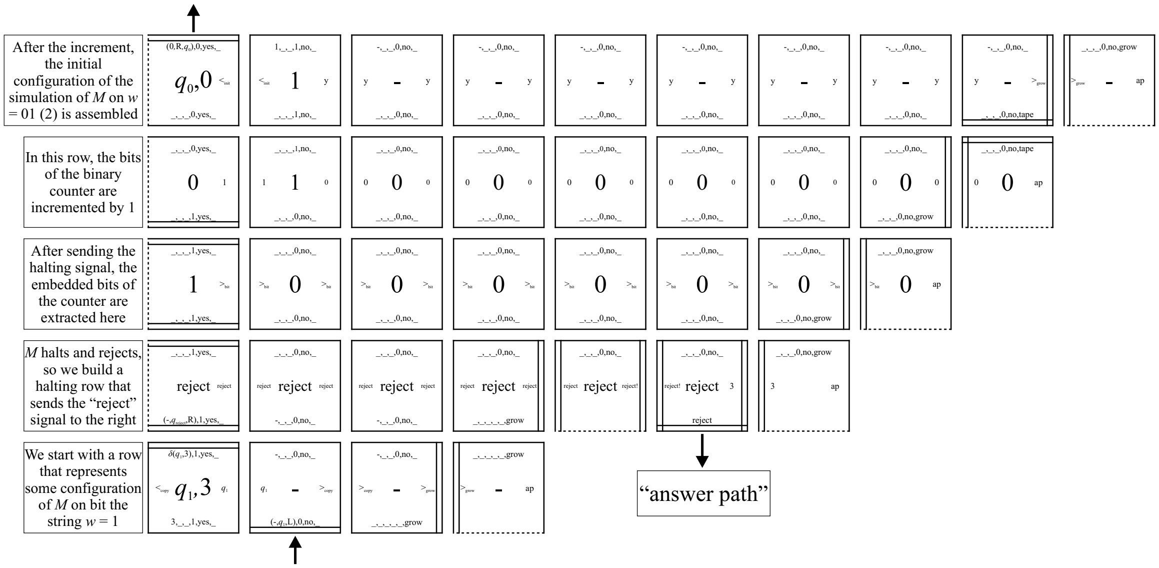
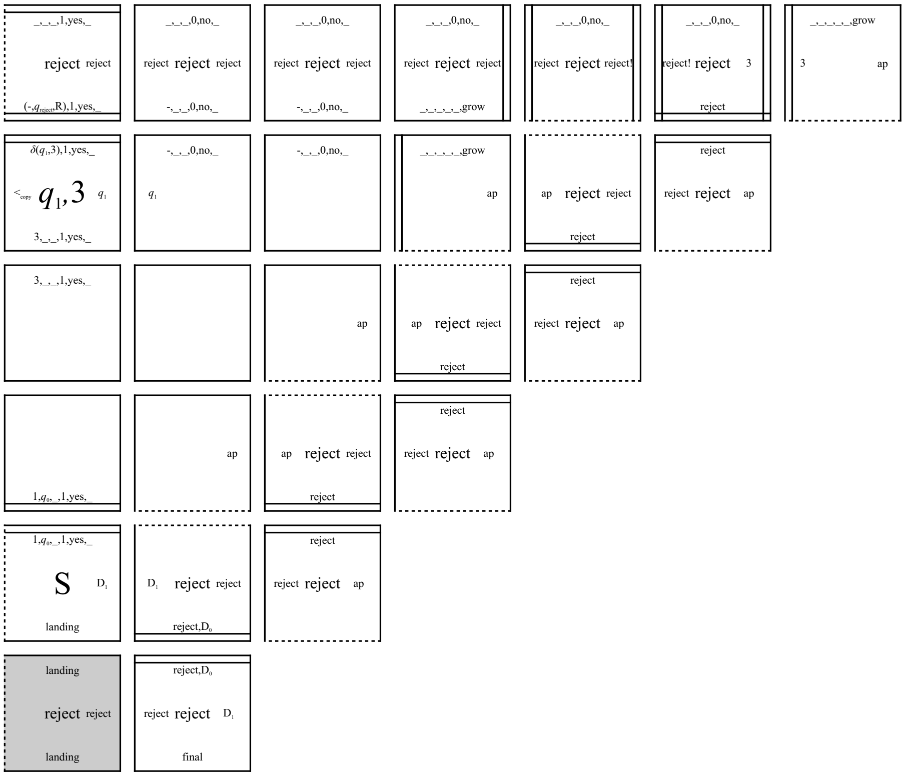

After the increment, the initial configuration of the simulation of M on w = 01 (2) is assembled
$q_0$,0
1
-
In this row, the bits of the binary counter are incremented by 1
0
1
After sending the halting signal, the embedded bits of the counter are extracted here
M halts and rejects, so we build a halting row that sends the "reject" signal to the right
reject
We start with a row that represents some configuration of M on bit the string w = 1
$q_1$,3
"answer path"
ap

Figure 4.3: Trivial example of the "halt-extract-increment-initialize" procedure carried out by our construction.

**Definition.** In our construction, for all $n \in \mathbb{N}$, there exists a one-tile-wide path that carries the answer to the question "is $n \in A$?" to the point $(0, -n)$. We call such a path a *decision path*.

The tile assembly system $\mathcal{T}_A$ consists of two logical modules. The first of these modules carries out the simulation of $M$ on the binary representation of $1 \leq n \in \mathbb{N}$. In order to simulate

$M$ on the binary representation of every natural number, we embed a kind of log width binary counter into the tiles of $\mathcal{T}_A$. Thus, each tile must "remember" (and possibly modify) the value and significance of a particular bit in the binary counter. Note that, because of the way we embedded the counter, $\mathcal{T}_A$ actually simulates $M$ on *the reverse of* the binary representation of every natural number. Tiles must also keep track of information pertaining to the simulation of $M$ on a given input. The information that each tile is responsible for is shown visually in Figure 4.2.



This is the black tile placed at (0,-1) representing the fact that $M$ rejects input $w = 1$

Figure 4.4: Trivial example of the one-tile-wide "decision path" that carries the answer to the question "is $1 \in L(M)$?" down to the appropriate point on the negative $y$-axis. Note that the $D_0$ and $D_1$ signals allow each decision path to turn the "corner" and proceed left toward the negative $y$-axis.

The assembly process starts by $\mathcal{T}_A$ simulating $M$ on the binary representation of 1. In general, after the simulation of $M$ on $i$ but *before* the simulation of $M$ on $i + 1$, $\mathcal{T}_A$ performs the following tasks.

1. The answer to the question "Does $M$ accept or reject $i$?" is propagated via a one-tile-wide "decision path" down to the point $(0, -i)$ (discussed below),

2. the bits of the embedded binary counter are extracted in the row immediately above the row of tiles representing the halting configuration of $M$ on $i$,

3. in the row immediately above the row in which the bits of the binary counter were extracted, the value of the binary counter is incremented by 1, and finally,

4. the simulation of $M$ on $i+1$ is initialized and proceeds in the same fashion as that of $M$ on $i$.

We give a concrete example of this four-step procedure in Figure 4.3.

The second component of $\mathcal{T}_A$ is a small group of tile types that carry the "accept" and "reject" signals to the appropriate location on the negative $y$-axis via a one-tile-wide path of tiles. The geometry of the existing assembly "guides" these decision paths to the correct location. Each decision path originates from the halting tile and proceeds down toward the $x$-axis. In order for each path to turn the corner and proceed toward their final destination somewhere on the negative $y$-axis, we propagate "diagonal" signals (i.e., $\mathrm{D}_0$ and $\mathrm{D}_1$) down and to the right into the fourth quadrant. An example of this process is illustrated in Figure 4.4.

Note that the simulation component of $\mathcal{T}_A$ can self-assemble in the absence of the decision path component whereas the latter requires the former to self-assemble. Figure 4.5 shows the "flow" of information from the simulation components (the light grey spaces) to their respective halting rows (dark grey horizontal rows) down to the appropriate point on the negative $y$-axis via a decision path (the dark grey paths that snake down and to the left along the assembly).

### 4.3.3 First Main Theorem

The following technical result is a primitive self-assembly simulator.

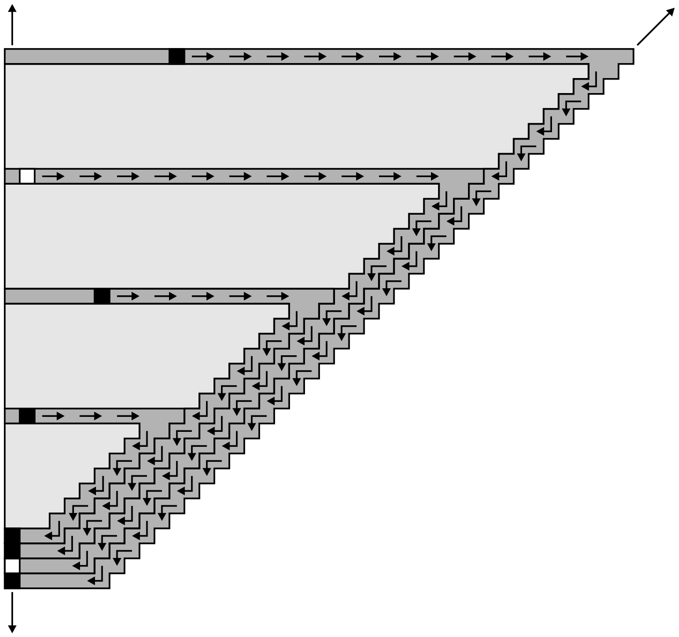


Figure 4.5: Intuitive depiction of the behavior of $\mathcal{T}_A$. The dark grey (horizontal) rows are halting rows. The arrows represent one-tile-wide paths of tiles that carry the answer to the question “is $n \in A$?” down to the negative $y$-axis.

**Lemma 4.3.3.** Let $A \subseteq \mathbb{Z}^2$. If $A$ weakly self-assembles, then there exists a TM $M_A$ with $L\left(M_A\right) = A$.

*Proof.* Assume that $A$ weakly self-assembles. Then there exists a TAS $\mathcal{T} = (T, \sigma, \tau)$ in which the set $A$ weakly self-assembles. Let $B$ be the set of "black" tile types given in the definition of weak self-assembly. Fix some enumeration $\vec{a}_1, \vec{a}_2, \vec{a}_3 \ldots$ of $\mathbb{Z}^2$, and let $M_A$ be the TM, defined as follows.

```
Require: $\vec{v} \in \mathbb{Z}^2$
  $\alpha := \sigma$
  while $\vec{v} \notin \operatorname{dom} \alpha$ do
    choose the least $j \in \mathbb{N}$ such that some tile can be added to $\alpha$ at $\vec{a}_j$
    choose some $t \in T$ that can be added to $\alpha$ at $\vec{a}_j$
    add $t$ to $\alpha$ at $\vec{a}_j$
  end while
  if $\alpha\left(\vec{v}\right) \in B$ then
    accept
  else
    reject
  end if
```

It is routine to verify that $M_A$ accepts $A$. □

**Lemma 4.3.4.** Let $A \subseteq \mathbb{N}$. If $\{0\} \times -A$ and $\{0\} \times (-A)^c$ weakly self-assemble, then $A$ is decidable.

*Proof.* Assume the hypothesis. Then by Lemma 4.3.3, there exist TMs $M_{\{0\}\times -A}$ and $M_{\{0\}\times(-A)^c\times-\mathbb{N}}$ satisfying $L\left(M_{\{0\}\times -A}\right) = \{0\} \times -A$, and $L\left(M_{\{0\}\times(-A)^c}\right) = \{0\} \times (-A)^c$, respectively. Now define the TM $M$ as follows.

```
Require: $n \in \mathbb{N}$
  Simulate both $M_{\{0\}\times -A}$ and $M_{\{0\}\times(-A)^c}$ on input $(0,-n)$ in parallel.
  if $M_{\{0\}\times -A}$ accepts then
    accept
  end if
  if $M_{\{0\}\times(-A)^c}$ accepts then
    reject
  end if
```

It is clear that $M$ is a decider for $A$. □

**Lemma 4.3.5.** Let $A \subseteq \mathbb{N}$. If the set $A$ is decidable, then $\{0\} \times -A$ and $\{0\} \times (-A)^c$ weakly self-assemble.

*Proof.* This follows immediately from Lemma 4.3.2. □

We now have the machinery to prove our main result.

**Theorem 4.3.6** (first main theorem)**.** Let $A \subseteq \mathbb{N}$. The set $A$ is decidable if and only if $\{0\} \times -A$ and $\{0\} \times (-A)^c$ weakly self-assemble. Furthermore, a single tile set suffices in that the choice of $B$ (the set of "black" tiles) determines whether $\{0\} \times -A$ or $\{0\} \times (-A)^c$ self-assembles.

*Proof.* This follows from Lemmas 4.3.4 and 4.3.5. □

In the next section, we will analyze the space requirements of this assembly and present a series of constructions which require less space, but at a price of increasing the tile set (a.k.a., program-size [45]) complexity.

## 4.4 Space Requirements of the Self-Assembly of Decidable Sets

In the proof of Theorem 5.3.7, we exhibited a directed TAS whose (infinite) terminal assembly requires two full quadrants of the plane. This leads one to ask the natural question:

is it possible to do any better than two quadrants? In other words, does Theorem 5.3.7 hold if only one quadrant of space, or less, is allowed? By investigating this, we hope to develop interesting “tile programming tricks”.

If $A \in \mathrm{DSPACE}(n)$, then it is possible to modify our construction to weakly self-assemble $A \times \{0\}$ using only one quadrant of space by making the tape for each computation $M(n)$ exactly $n-1$ tiles wide, then self-assembling a path with the result directly down the right side of the assembly to the location $(n, 0)$. However, in the case that $A \notin \mathrm{DSPACE}\left(O(n)\right)$, this particular construction technique does not suffice to self-assemble $A \times \{0\}$ within only one quadrant because we happen to use one tile to represent each tape cell and furthermore we grow the Turing machine tape to the right once per computation step. In the remainder of this section we will discuss why this is the case, as well as alternative constructions which do suffice even for such complex languages, and their space requirements.

### 4.4.1 The Impact of the Decision Paths

It is clear that the portions of the assembly of our main construction, which initiate and perform each of the infinite series of computations, are contained within a wedge-shaped portion of a single quadrant. However, the decision paths end up being the portion of the assembly which force the additional space requirement of a second quadrant for sufficiently complex languages.

If the language $A \notin \mathrm{DSPACE}\left(O(n)\right)$, then there exists some $n$ for which the computation $M(n)$ requires more than $O(n)$ amount of space, and therefore the assembly simulating $M(n)$ requires at least one row of tiles whose width is at least $n$ tiles. Since the row of tiles forming the computation will already be in place and cannot be removed to allow the decision path through, in order for the decision path emanating from this computation to terminate in the correct location, $(n, 0)$, it will have to go around that row, passing through an $x$-coordinate greater than $n$, then turn left at some point. (This ignores the possibility of the decision paths traveling down the left side of the assembly since they would therefore already be using

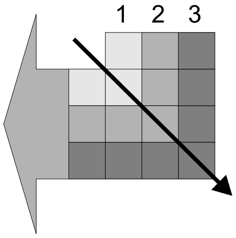


Figure 4.6: Example of a series of paths which make a left turn. Notice that if path “2” assembles after path “1”, in order to make the turn it must be translated one additional coordinate downward. This must occur for each of the infinite series of paths which follow.

a second quadrant.) Additionally, each of the infinite number of subsequent decision paths will then have to assemble around that path. Figure 4.6 depicts the situation where the first three paths in an infinite series of paths make a left turn. Figure 4.5 exemplifies how the diagonal along the right side of our construction makes a series of “left-turns.” Note that each subsequent path must be translated one additional coordinate downward, and therefore, since there must be an infinite number of subsequent paths, eventually the decision path for some $m \in \mathbb{N}$ must place a tile on the $x$-axis in a location other than $(m, 0)$, or otherwise block another decision path. It is for this reason that the Main Construction requires the use of two quadrants for a language $A$ of sufficient space complexity.

### 4.4.2 Squeezing More Information Into Each Decision Path

As shown above, the decision paths of the Main Construction form a virtual bottleneck which force the assembly into a second quadrant. This is due to the fact that each computation has its own, unique, decision path. However, if we “compress” more information into the decision paths, we can further reduce the space requirements. In fact, with a tradeoff in tile set complexity, where complexity is measured as the size of the tile set required, not only can one quadrant be made sufficient, but an arbitrarily small slice of a single quadrant can suffice.

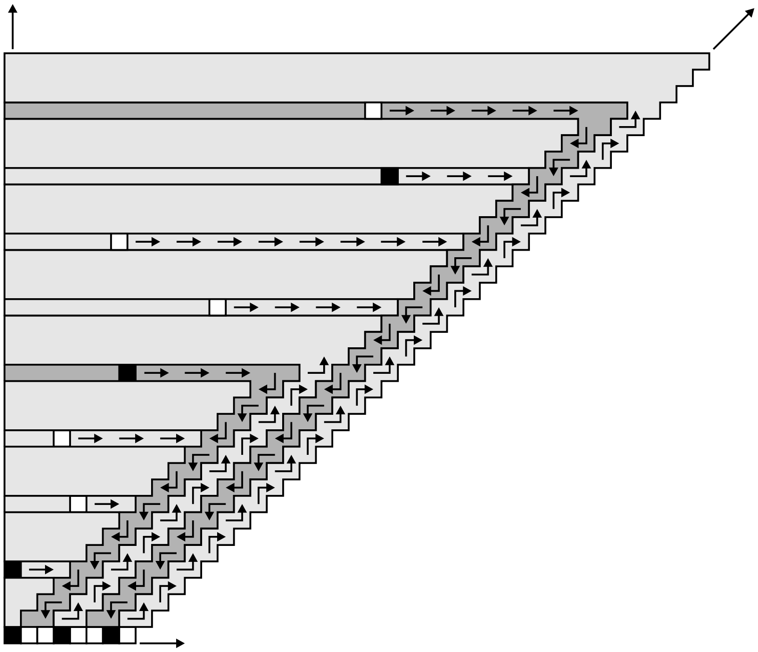


Figure 4.7: The single quadrant construction.

#### 4.4.2.1 A Single-Quadrant Construction

For our next construction, instead of creating a decision path for each individual computation, we let four computations complete (i.e. $M(n)$, $M(n+1)$, $M(n+2)$, and $M(n+3)$) and then create a single decision path which collects and transmits all four answers to their correct locations on the $x$-axis. Figure 4.7 shows a high-level overview of this construction.

Essentially, an additional counter value which cycles from 0 to 3 and increments at the completion of each computation is passed upward through the right side of each computation. This allows the completion of each fourth computation to initiate the growth of a decision path which moves downward, collecting the previous three results until the single path contains four results. It then continues to the $x$-axis where it deposits all four results and initiates the growth of an upward growing path which eventually allows the next set of results to be propagated downward. Note that a main function of the upward growing path is to propagate a marker denoting which row is immediately above the $x$-axis to the right side of the assembly, and thus signal the next downward growing path when it needs to "unpack" its results.

This "compression" of four results into each result path obviates the need for the assembly to grow into a second quadrant, and it does so with only a trivial increase in tile set complexity, which comes mostly from the $2^2 + 2^3 + 2^4$ tile types which must be added to allow the collection and transmission of 4 results in each decision path.

#### 4.4.2.2 Constructions Using Arbitrarily Thin "Pie" Slices of One Quadrant

We now demonstrate how we can further exploit the method of compressing increasingly larger sets of results into each decision path to create constructions that use arbitrarily thin (but infinite) pie-shaped slices of a single quadrant. These constructions will be accompanied by a quickly growing trade off in tile set complexity which we will also analyze. Note that we merely sketch these constructions and leave open the problem of implementation, and perhaps further optimization with regards to space and tile set complexity.

First, note that a standard wedge construction can be modified so that the leftmost tile of each row is $s$ positions further to the right than the leftmost tile of the row immediately below it. This is done by:

1. Initiating the growth of each row at the leftmost position and growing from left to right.
2. Within each tile representing the $n$th tape cell from the left of the tape, encode the contents of tape cells $(n - s, \ldots, n)$, including the state information if one of those cells contains the tape head.

This works because the tile representing the tape cell that must receive the tape head after a left or right moving transition always assembles either directly above or above and to the right of the tile representing the tape head in the previous row, and therefore that information is available as necessary to each proceeding row. One additional difference from the standard wedge construction is that, if a computation enters a halting state while the tape head is on the $n^{\text{th}}$ tape cell from the left, $n/s$ additional rows (in which no computation is performed) will need to assemble before the information that it halted reaches the leftmost edge and the assembly of that simulation can terminate.

By using such a "slanted" wedge construction in the construction of Section 4.4.2.1 and modifying the decision paths to conform to the desired slope and to accumulate and carry $2s + 4$ results to the $x$-axis, (along with a few other trivial modifications) an assembly which requires only the portion of the first quadrant lying below the line $y = x/s$ can form which weakly self-assembles $A \times \{0\}$. Two examples of such constructions and their desired behaviors can be seen in Figure 4.8.

We conjecture that the tradeoff for shrinking the assembly into such arbitrarily small "pie slices" of the first quadrant is an exponential increase in tile set complexity. In fact, for a given $s$, the size of the tile set required for our proposed assembly would be $O(\gamma^{s+1})$ where $\gamma = |\Gamma|$ (and $\Gamma$ is the tape alphabet of the Turing machine M being simulated). Note that the tile complexity of our original two-quadrant construction is $O(\gamma^1)$ (this bound also holds for our one-quadrant construction as well).

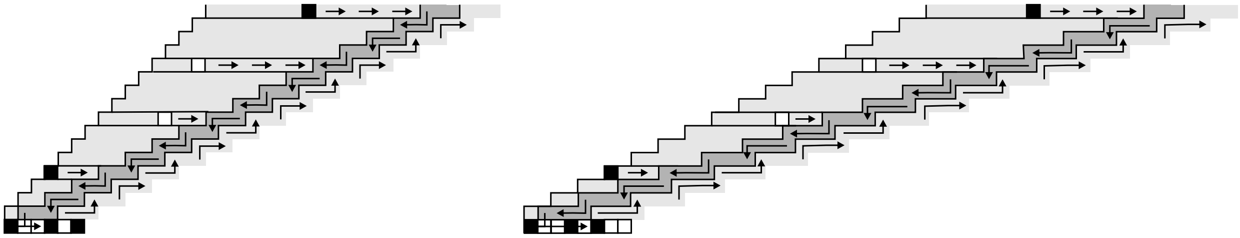


Figure 4.8: Examples of assemblies which weakly self-assembly the set $A \times \{0\}$ within the portions of Quadrant 1 (approximately) underneath the lines $y = x$ (left) and $y = x/2$ (right), corresponding to $s = 1$ and $s = 2$, respectively. Note that the constructions must combine 6 (for the left) and 8 (for the right) signals into each decision path.

## 4.5 Conclusion

In this chapter, we investigated the self-assembly of decidable sets of natural numbers in the TAM. We proved that, for every decidable language $A \subseteq \mathbb{N}$, $A \times \{0\}$ and $A^c \times \{0\}$ weakly self-assemble. This implied a novel characterization of decidable sets in terms of self-assembly. We then analyzed the space requirements of our construction and showed that, in general, for decidable languages, our construction requires two quadrants of space. This led us to the presentation of slightly modified constructions which required, first, only one quadrant of space, and then increasingly smaller portions of a single quadrant. However, this decrease in space was accompanied a "proportional" increase in the size of the tile set complexity.

Additional examples of the trade off between the space complexity of languages and the amount of space required to self-assemble their characteristic sequences in the TAM include the fact that one *spatial* dimension is sufficient to self-assemble $A \times \{0\}$ if and only if $A$ is a regular language over a unary alphabet, and our speculation that if $A$ is a regular language over a binary alphabet that only one *fractal* dimension would be required. Many open problems related to such relationships remain, such as the implementation and potential optimization of our "pie-slice" constructions, and a proof of our above speculation about regular languages over binary alphabets. We hope that continued research in this direction will further extend these results, exposing, more and more, the rich interconnectedness between geometry and computation in the TAM.

# CHAPTER 5. Self-Assembly of Discrete Self-Similar Fractals

The work in this chapter was performed with co-author Scott M. Summers. It has been published as [42] and [40].

## 5.1 Introduction

Strict self-assembly is the self-assembly of a particular connected shape and nothing else. Note that strict self-assembly is a special case of weak self-assembly, where the underlying canvas (onto which a shape $X \subseteq \mathbb{Z}^2$ is painted) is the shape $X$ itself. We say that the *tile complexity of a shape* $X$ is the size of the tile set in which $X$ strictly self-assembles. For the case of the strict self-assembly of finite shapes, the tile complexity of the shape becomes an important factor because every finite shape trivially (but perhaps not feasibly) strictly self-assembles via a spanning-tree construction. Numerous lower bounds on the tile complexity for the strict self-assembly of particular classes of shapes have been established. For instance, Rothemund and Winfree [45] proved that there exist very small tile sets in which extremely large squares strictly self-assemble. In 2002, Adleman, Cheng, Goel, Huang, Kempe, Moisset de Espanés and Rothemund [5] exhibited polynomial-time algorithms capable of finding the "minimum" tile assembly system (with respect to tile complexity) in which tree shapes and squares strictly self-assemble. Moreover, Soloveichik and Winfree [48] discovered the remarkable fact that, if one is not concerned with the scale of an "algorithmically describable" finite shape $X$, then there is always a small tile set in which $X$ strictly self-assembles.

For the case of strict self-assembly of infinite shapes, the power of the TAM has only recently been investigated. In this case, we ignore tile-complexity and are thus primarily concerned with the question of whether or not there is *any* tile assembly system in which a particular infinite shape strictly self-assembles. Note that, unlike for the case of the strict self-assembly of finite shapes, the tile complexity of an infinite shape $X$ cannot be a function of $|X|$. Much of the previous work in this area has focused on the strict self-assembly of discrete fractal structures. In particular, Lathrop, Lutz and Summers [36] established that self-similar tree shapes do not strictly self-assemble in the TAM. A "fiber construction" is also given in [36], in which a non-trivial (infinite) fractal structure resembling the standard discrete Sierpinski triangle strictly self-assembles. Moreover, Kautz and Lathrop [31] define an infinite class of "numerically self-similar" discrete self-similar fractals (to which the Sierpinski triangle and the Sierpinski carpet belong), and give a general construction for generating tile assembly systems in which such numerically self-similar fractals weakly self-assemble.

In this chapter, we search for (1) *theoretical* limitations of the TAM, with respect to the strict self-assembly of infinite shapes, and (2) techniques that allow one to "work around" such limitations. Specifically, we investigate the strict self-assembly of fractal shapes in the TAM. We are interested in fractal shapes because of their geometric aperiodicity and their frequent occurrence in natural biological systems. We prove three main results: two negative and one positive. Our first negative (i.e., impossibility) result says that no self-similar fractal weakly self-assembles in the TAM at temperature 1 in a locally deterministic tile assembly system. In our second impossibility result, we exhibit a class of discrete self-similar fractals, to which the standard discrete Sierpinski triangle belongs, that do not strictly self-assemble in the TAM (at *any* temperature). Finally, in our positive result, we use simple modified counters to extend the fiber construction from Lathrop, Lutz and Summers [36] to a particular class of "nice" discrete self-similar fractals.

## 5.2 Preliminaries

### 5.2.1 Discrete Self-Similar Fractals

In this subsection we introduce discrete self-similar fractals, and a kind of discrete fractal dimension called zeta-dimension [16].

**Definition.** Let $1 < c \in \mathbb{N}$, and $X \subsetneq \mathbb{N}^2$ (we do not consider $\mathbb{N}^2$ to be a self-similar fractal). We say that $X$ is a *$c$-discrete self-similar fractal*, if there is a set $V \subseteq \{0, \ldots, c-1\} \times \{0, \ldots, c-1\}$ with $|V| > c$, and

$$V \notin \{\{(i,i) \mid 0 \leq i < c\}, \{(i,0) \mid 0 \leq i < c\}, \{(0,i) \mid 0 \leq i < c\}\},$$

such that

$$X = \bigcup_{i=0}^{\infty} X_i,$$

where $X_i$ is the $i^{\text{th}}$ *stage* satisfying $X_0 = \{(0,0)\}$, and $X_{i+1} = X_i \cup \left(X_i + c^i V\right)$. In this case, we say that $V$ *generates* $X$.

Intuitively, we define $c$-discrete self-similar fractals as follows (demonstrated in Figure 5.1). Begin with a $c$ x $c$ square where the $0^{th}$ stage is defined simply by the point $(0,0)$. The first stage, which we call the *generator*, is defined by selecting additional locations in the $c$ x $c$ square. Every subsequent stage $i$ is formed by treating the stage $i-1$ as a magnified version of the point $(0,0)$ in the generator and creating one full copy of stage $i-1$ for every other point in the generator, then treating those copies as the magnified versions of their respective points in the generator and translating them accordingly.

**Definition.** $X \subsetneq \mathbb{N}^2$. We say that $X$ is a *discrete self-similar fractal* if it is a $c$-discrete self-similar fractal for some $c \in \mathbb{N}$.

In this chapter, we are concerned with the following class of self-similar fractals.

**Definition.** A *nice discrete self-similar fractal* is a discrete self-similar fractal such that $(\{0, \ldots, c-1\} \times \{0\}) \cup (\{0\} \times \{0, \ldots, c-1\}) \subseteq V$, and $G^{\#}_{V}$ is connected.

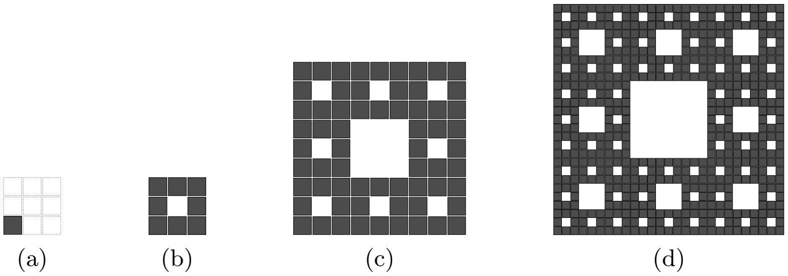


Figure 5.1: The first four stages of the discrete Sierpinski carpet ($X_0$, $X_1 = V$, $X_2$, and $X_3$ are shown in (a), (b), (c), and (d) respectively).

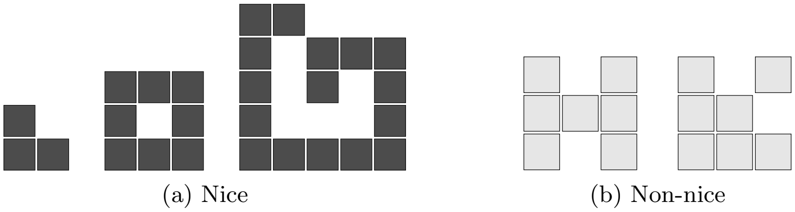


Figure 5.2: The generators of discrete self-similar fractals. The fractals in (a) are nice, whereas (b) shows two non-nice fractals.

See Figure 5.2 for examples of nice discrete self-similar fractals.

The most commonly used dimension for discrete fractals is zeta-dimension, which we use in this chapter.

**Definition.** (Doty, Gu, Lutz, Mayordomo, and Moser [16]) For each set $A \subseteq \mathbb{Z}^2$, the *zeta-dimension* of $A$ is

$$\mathrm{Dim}_\zeta(A) = \limsup_{n \to \infty} \frac{\log |A_{\leq n}|}{\log n},$$

where $A_{\leq n} = \{(k,l) \in A \mid |k| + |l| \leq n\}$.

It is clear that $0 \leq \mathrm{Dim}_\zeta(A) \leq 2$ for all $A \subseteq \mathbb{Z}^2$.

## 5.3 Impossibility Results

In this section, we explore the theoretical limitations of the Tile Assembly Model with respect to the self-assembly of fractal shapes. We are specifically interested in the self-assembly of discrete fractal structures because of their geometrically aperiodic structure. First, we establish that no discrete self-similar fractal weakly self-assembles at temperature $\tau = 1$ in a locally deterministic tile assembly system. Second, we exhibit a class $\mathcal{C}$ of discrete self-similar fractals, and prove that if $F \in \mathcal{C}$, then $F$ does not strictly self-assemble in the TAM.

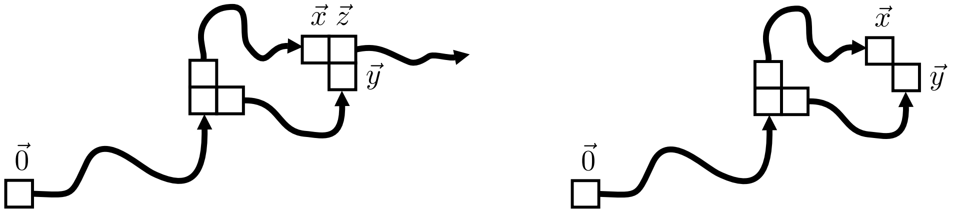
Throughout this section, we assume (without loss of generality) that $T$ is a finite set of tile types, and $\sigma$ is an assembly that places a single tile at the origin.

The following theorem is evidence in support of the claim that locally deterministic temperature 1 tile assembly systems cannot produce "sophisticated" shapes and patterns.

**Theorem 5.3.1.** Let $\mathcal{T} = (T, \sigma, \tau = 1)$ be a tile assembly system, and $\alpha \in \mathcal{A}_\Box[\mathcal{T}]$. If $\mathcal{T}$ is locally deterministic, then the binding graph $G_\alpha$ is a tree.

*Proof.* Suppose that the binding graph $G_\alpha$ is not a tree. Then there is a cycle $C$ in $G_\alpha$. Let $\vec{z} \in C$, $t = \alpha(\vec{z})$ and $\vec{\alpha}$ be an assembly sequence satisfying $\alpha = \text{dom res}(\vec{\alpha})$. There must be two simple paths $\pi_1$ and $\pi_2$ in $G_\alpha$ from $\vec{0}$ to $\vec{z}$ with $\pi_1 \neq \pi_2$. Since $\tau = 1$, there exists an assembly sequence $\vec{\alpha}'$ where $\vec{\alpha}'$ first assembles $\pi_1 \cap \pi_2$ (if such an intersection exists), then assembles $(\pi_1 - \pi_2) - \{\vec{z}\}$, and finally assembles $(\pi_2 - \pi_1) - \{\vec{z}\}$. We can use $\vec{\alpha}'$ to define the assembly sequence $\vec{\alpha}''$ that does what $\vec{\alpha}'$ does but places the tile type $t$ at position $\vec{z}$ and then does whatever $\vec{\alpha}$ does for all $\vec{m} \notin \text{dom res}(\vec{\alpha}')$. But since $\vec{z} \in C$, the tile that $\vec{\alpha}'$ places at position $\vec{z}$ has two input sides and binds with strength $2 > \tau = 1$. Thus, the assembly sequence $\vec{\alpha}'$ is not locally deterministic, and it follows by Lemma 2.1.2 that $\mathcal{T}$ is not locally deterministic. □

See Figure 5.3 for a visual depiction of the above proof.



(a) An example of a cycle existing in the binding graph of an assembly

(b) A possible partial assembly which forms both paths to $\vec{z}$ before $\vec{z}$ is tiled

Figure 5.3: Visual depiction of the proof of Lemma 5.3.1. If a cycle could exist (image a), then image b shows a possible partial assembly to which the tile attaching in location $\vec{z}$ would bind with strength 2. This is a contradiction to the claim that the tile assembly system is temperature one and locally deterministic.

Note that the converse of Theorem 5.3.1 is not necessarily true.

**Definition** (Lathrop, Lutz, and Summers [36]). Let $G = (V, E)$ be a graph, and let $D \subseteq V$. For each $r \in V$, the *$D$-$r$-rooted subgraph* of $G$ is the graph $G_{D,r} = (V_{D,r}, E_{D,r})$, where

$$V_{D,r} = \{v \in V \mid \text{every simple path from } v \text{ to } D \text{ in } G \text{ goes through } r\}$$

and $E_{D,r} = E \cap [V_{D,r}]^2$. $B$ is a *$D$-subgraph* of $G$ if it is a $D$-$r$-rooted subgraph of $G$ for some $r \in V$.

**Definition.** Let $A, B \subseteq \mathbb{Z}^2$. We say that $A$ and $B$ are isomorphic, and we write $A \sim B$ if there exists $\vec{v} \in \mathbb{Z}^2$ such that $A = B + \vec{v}$.

**Definition.** Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be grid graphs. We say that $G_1$ and $G_2$ are isomorphic and we write $G_1 \sim G_2$ if $V_1$ and $V_2$ are isomorphic.

See Figure 5.4 for an intuitive example of a $D$-subgraph.

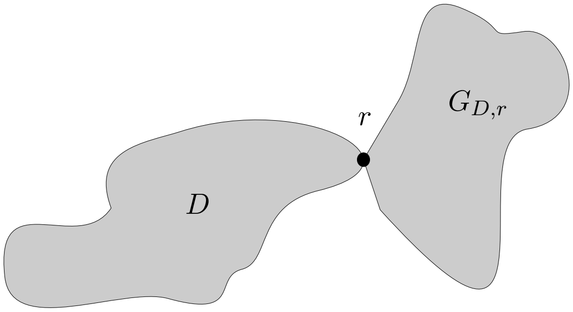


Figure 5.4: An example $D$-$r$-rooted subgraph of a graph $G$. Notice that the paths from all points in the region labeled $G_{D,r}$ to the region labeled $D$ must go through the point $r$.

**Lemma 5.3.2.** Let $\mathcal{T} = (T, \sigma, 1)$ be a locally deterministic tile assembly system, $\vec{\alpha}$ be an assembly sequence with $\text{res}(\vec{\alpha}) = \alpha \in \mathcal{A}_{\Box}[\mathcal{T}]$, and $\pi$ be a simple path in the binding graph $G_\alpha = (V, E)$ originating at the origin. If there exist points $\vec{p}, \vec{q} \in \pi$ with $\alpha(\vec{p}) = \alpha(\vec{q})$ ($\vec{p}$ precedes $\vec{q}$ on $\pi$), then, for all $\vec{a} \in V_{\{\vec{0}\},\vec{p}}$, $\alpha(\vec{a}) = \alpha(\vec{a} + \vec{v})$, where $\vec{v} = \vec{q} - \vec{p}$.

*Proof.* Let $n \in \mathbb{N}$, and $\pi_{\vec{p}}^{(n)} = (\vec{p}, \ldots)$ be a simple path in $G_{\{\vec{0}\},\vec{p}}$ with $\left|\pi_{\vec{p}}^{(n)}\right| = n$. Define, for all $i \in \{0, \ldots, n-2\}$, the unit vector $\vec{u}_i = \pi_{\vec{p}}^{(n)}[i+1] - \pi_{\vec{p}}^{(n)}[i]$. It suffices to show that there exists a simple path $\pi_{\vec{q}}^{(n)} = \left(\vec{q}, \vec{q} + \vec{u}_0, \vec{q} + \vec{u}_0 + \vec{u}_1, \ldots, \vec{q} + \sum_{i=0}^{n-1} \vec{u}_i\right)$ in $G_{\{\vec{0}\},\vec{p}}$, satisfying, for all $i \in \{0, \ldots, n-1\}$, $\vec{v} = \pi_{\vec{q}}^{(n)}[i] - \pi_{\vec{p}}^{(n)}[i]$, and $\alpha\left(\pi_{\vec{p}}^{(n)}[i]\right) = \alpha\left(\pi_{\vec{q}}^{(n)}[i]\right)$. We will proceed by induction on $n$.

For the base case (i.e., $n = 0$), we have $\pi_{\vec{p}}^{(0)} = (\vec{p})$. The hypothesis of the lemma tells us that $\vec{q} \in G_{\{\vec{0}\},\vec{p}}$, whence there exists a simple path $\pi_{\vec{q}}^{(0)} = (\vec{q})$ in $G_{\{\vec{0}\},\vec{p}}$ with $\left|\pi_{\vec{q}}^{(0)}\right| = 0$. Furthermore, we have

$$\begin{aligned} \pi_{\vec{q}}^{(0)}[0] - \pi_{\vec{p}}^{(0)}[0] &= \vec{q} - \vec{p} \\ &= \vec{v}, \end{aligned}$$

and

$$\begin{aligned}\alpha\left(\pi^{(0)}_{\vec{p}}[0]\right) &= \alpha\left(\vec{p}\right)\\ &= \alpha\left(\vec{q}\right)\\ &= \alpha\left(\pi^{(0)}_{\vec{q}}[0]\right).\end{aligned}$$

Assume that, if $\pi^{(n-1)}_{\vec{p}} = (\vec{p},\ldots)$ is a simple path in $G_{\{\vec{0}\},\vec{p}}$ with $\left|\pi^{(n-1)}_{\vec{p}}\right| = n-1$, then there exists a simple path $\pi^{(n-1)}_{\vec{q}} = \left(\vec{q},\vec{q}+\vec{u}_0,\vec{q}+\vec{u}_0+\vec{u}_1,\ldots,\vec{q}+\sum_{i=0}^{n-2}\vec{u}_i\right)$ in $G_{\{\vec{0}\},\vec{p}}$, satisfying, for all $i\in\{0,\ldots,n-2\}$, $\vec{v}=\pi^{(n-1)}_{\vec{q}}[i]-\pi^{(n-1)}_{\vec{p}}[i]$, and $\alpha\left(\pi^{(n-1)}_{\vec{p}}[i]\right)=\alpha\left(\pi^{(n-1)}_{\vec{q}}[i]\right)$.

Let $\pi^{(n)}_{\vec{p}} = (\vec{p},\ldots)$ be any simple in $G_{\{\vec{0}\},\vec{p}}$ with $\left|\pi^{(n)}_{\vec{p}}\right| = n$. Write

$$\pi^{(n)}_{\vec{p}} = \left(\vec{p},\vec{p}+\vec{u}_0,\vec{p}+\vec{u}_0+\vec{u}_1,\ldots,\vec{p}+\sum_{i=0}^{n-2}\vec{u}_i,\vec{p}+\sum_{i=0}^{n-1}\vec{u}_i\right),$$

where each $\vec{u}_i$ is defined as above. Then $\pi^{(n-1)}_{\vec{p}} = \left(\vec{p},\vec{p}+\vec{u}_0,\vec{p}+\vec{u}_0+\vec{u}_1,\ldots,\vec{p}+\sum_{i=0}^{n-2}\vec{u}_i\right)$ is a simple path in $G_{\{\vec{0}\},\vec{p}}$ with $\left|\pi^{(n-1)}_{\vec{p}}\right| = n-1$ By the induction hypothesis, there exists a simple path $\pi^{(n-1)}_{\vec{q}} = \left(\vec{q},\vec{q}+\vec{u}_0,\vec{q}+\vec{u}_0+\vec{u}_1,\ldots,\vec{q}+\sum_{i=0}^{n-2}\vec{u}_i\right)$ in $G_{\{\vec{0}\},\vec{p}}$ satisfying, for all $i\in\{0,\ldots,n-2\}$, $\vec{v}=\pi^{(n-1)}_{\vec{q}}[i]-\pi^{(n-1)}_{\vec{p}}[i]$, and $\alpha\left(\pi^{(n-1)}_{\vec{p}}[i]\right)=\alpha\left(\pi^{(n-1)}_{\vec{q}}[i]\right)$. It suffices to verify that

$$\pi^{(n)}_{\vec{q}} = \left(\vec{q},\vec{q}+\vec{u}_0,\vec{q}+\vec{u}_0+\vec{u}_1,\ldots,\vec{q}+\sum_{i=0}^{n-2}\vec{u}_i,\vec{q}+\sum_{i=0}^{n-1}\vec{u}_i\right)$$

is a simple path in $G_{\{\vec{0}\},\vec{p}}$ satisfying

1. $\vec{v}=\pi^{(n)}_{\vec{q}}[n-1]-\pi^{(n)}_{\vec{p}}[n-1]$, and
2. $\alpha\left(\pi^{(n)}_{\vec{p}}[n-1]\right)=\alpha\left(\pi^{(n)}_{\vec{q}}[n-1]\right)$.

To see that $\pi^{(n)}_{\vec{q}}$ is a simple path in $G_{\{\vec{0}\},\vec{p}}$, suppose otherwise. Then $\left\{\vec{q}+\sum_{i=0}^{n-2}\vec{u}_i,\vec{q}+\sum_{i=0}^{n-1}\vec{u}_i\right\}$ is not an edge in $G_{\{\vec{0}\},\vec{p}}$. This implies that $\alpha\left(\vec{q}+\sum_{i=0}^{n-1}\vec{u}_i\right)\downarrow$. Let $t=\alpha\left(\vec{q}+\sum_{i=0}^{n-1}\vec{u}_i\right)$, and $t^*=\alpha\left(\pi^{(n)}_{\vec{p}}[n-1]\right)$. Note that $t^*\neq t$. Then we have

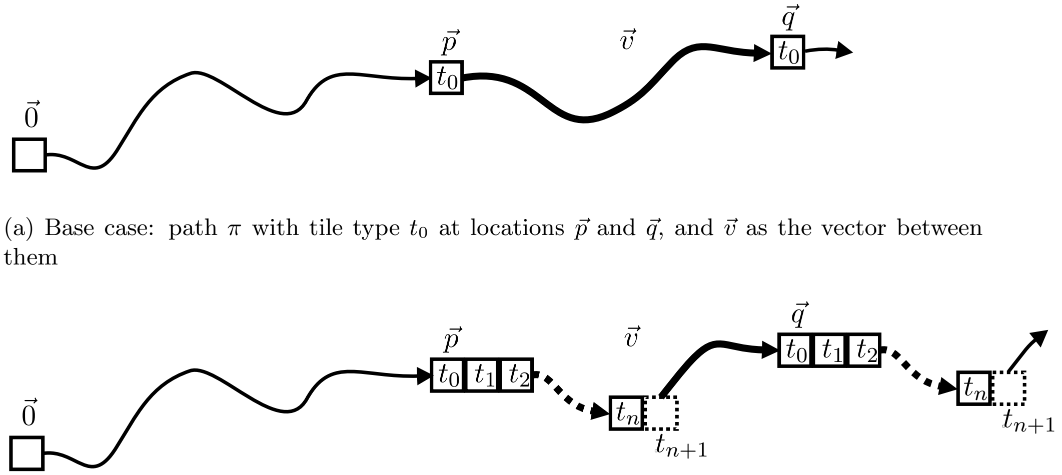
$$\vec{q}+\sum_{i=0}^{n-1}\vec{u}_i\in\partial^{\tau=1}_{t^*}\left(\vec{\alpha}\setminus\left\{\vec{q}+\sum_{i=0}^{n-1}\vec{u}_i\right\}\right)$$

because, by the induction hypothesis, $\alpha\left(\pi_{\vec{p}}^{(n)}[n-2]\right) = \alpha\left(\pi_{\vec{q}}^{(n)}[n-2]\right)$. However, this contradicts the fact that $\mathcal{T}$ is locally deterministic, whence $\pi_{\vec{q}}^{(n)}$ is a simple path in $G_{\{\vec{0}\},\vec{p}}$.

Note that the argument in the above paragraph also establishes that $\alpha\left(\pi_{\vec{p}}^{(n)}[n-1]\right) = \alpha\left(\pi_{\vec{q}}^{(n)}[n-1]\right)$. Finally, we have

$$\begin{aligned} \pi_{\vec{q}}^{(n)}[n-1] - \pi_{\vec{p}}^{(n)}[n-1] &= \vec{q} + \sum_{i=0}^{n-1} \vec{u}_i - \left(\vec{p} + \sum_{i=0}^{n-1} \vec{u}_i\right) \\ &= \vec{q} - \vec{p} \\ &= \vec{v}, \end{aligned}$$

and the induction hypothesis has been verified. □



(a) Base case: path $\pi$ with tile type $t_0$ at locations $\vec{p}$ and $\vec{q}$, and $\vec{v}$ as the vector between them

(b) Steps $n$ and $n+1$ of the induction, showing identical tile types at each location along $\pi$ from $\vec{p}$ offset by $\vec{v}$

Figure 5.5: Visual depiction of the inductive proof of Lemma 5.3.2

Figure 5.5 gives a visual outline of the proof of Lemma 5.3.2. Image 5.5a shows an example path $\pi$ beginning at the origin of an assembly. The points $\vec{p}$ and $\vec{q}$ can be seen, along with the darkened portion of the path, labeled $\vec{v}$, between them. Since the points $\vec{p}$ and $\vec{q}$ were defined as being positions in the path containing the same tile type, they've both been labeled with tile type $t_0$. This represents the base case for the proof by induction with $n = 0$ and $\pi_{\vec{p}}^{(0)} = (\vec{p})$.

Image 5.5b depicts the induction hypothesis with a path of length $n-1$, where the sequence of $n-1$ tiles immediately following $\vec{p}$ are of types identical to those following $\vec{q}$. Then, the final step, $n$, is shown by the tiles with dashed lines. The induction must hold because the tile set is locally deterministic and therefore tiles of type $t_n$ are the only type that can attach to that side of tiles of type $t_{n-1}$. Furthermore, position $\vec{q}$ could then be treated as the next $\vec{p}$, with a new $\vec{q}$ displaced by vector $\vec{v}$ further down the path, and thus the same pattern must repeat infinitely many times along the path $\pi$.

Theorem 5.3.1, in conjunction with Lemma 5.3.2, supports the claim that if a set $X \subseteq \mathbb{Z}^2$ weakly self-assembles in a locally deterministic temperature 1 tile assembly system, then $X$ is necessarily "simple" (of course, this is not surprising given the strength of local determinism). We formally define "simple" as follows.

**Definition.** Let $X \subseteq \mathbb{Z}^2$.

1. We say that $X$ is *periodic with respect to* $\vec{0} \neq \vec{v} \in \mathbb{Z}^2$ if, for all $\vec{x} \in X$, $\vec{x} + \vec{v} \in X$.

2. We say that $X$ is *periodic* if it is periodic with respect to $\vec{v}$ for some $\vec{0} \neq \vec{v} \in \mathbb{Z}^2$.

Note that the Definition 5.2.1 tells us that discrete self-similar fractals are not periodic sets. The following observation is also clear.

**Observation 5.3.3.** Let $X \subsetneq \mathbb{N}^2$, and suppose that $C \subset \mathbb{N}$ is finite. If $X$ is a discrete self-similar fractal then $X - C$ is not a finite union of periodic sets.

This observation simply states that, while we know from Definition 5.2.1 that discrete self-similar fractals are not periodic sets, there is also no finite portion of a discrete self-similar fractal which could be removed from it, leaving a set which is a finite union of periodic sets.

We will use the following technical lemma in the proof of our first main theorem.

**Lemma 5.3.4.** Let $\mathcal{T} = (T, \sigma, 1)$ be a locally deterministic tile assembly system, and $\alpha \in \mathcal{A}_{\Box}[\mathcal{T}]$. If $X \subseteq \mathbb{Z}^2$ weakly self-assembles in $\mathcal{T}$, then, there exists $m \in \mathbb{N}$, such that

$$X = \bigcup_{i=0}^{m-1} X_i,$$

where, for each $i \in \{0, \ldots, m-1\}$, $X_i$ is either finite or periodic.

*Proof.* Assume the hypothesis. If $X$ is finite, then we are done, so assume otherwise. Let

$$D = \left\{ (x,y) \in \mathbb{Z}^2 \cap \text{dom } \alpha \;\middle|\; |x| + |y| \leq |T| \right\},$$

and

$$B = \left\{ (x,y) \in \mathbb{Z}^2 \cap \text{dom } \alpha \;\middle|\; |x| + |y| = |T| + 1 \right\}.$$

For each $\vec{r} \in B$, let

$$G_{\left\{\vec{0}\right\},\vec{r}} = \left( V_{\left\{\vec{0}\right\},\vec{r}}, E_{\left\{\vec{0}\right\},\vec{r}} \right)$$

be the $\left\{\vec{0}\right\}$-subgraph of $G_\alpha$ rooted at $\vec{r}$. See Figure 5.6 for an intuitive and visual explanation of $D$, $B$, and $G_{\left\{\vec{0}\right\},\vec{r}}$. Note that if $V_{\left\{\vec{0}\right\},\vec{r}}$ is infinite, then $X \cap V_{\left\{\vec{0}\right\},\vec{r}}$ is periodic by Lemma 5.3.2. The lemma follows by taking

$$X = D \cup \bigcup_{\vec{r} \in B} V_{\left\{\vec{0}\right\},\vec{r}}.$$

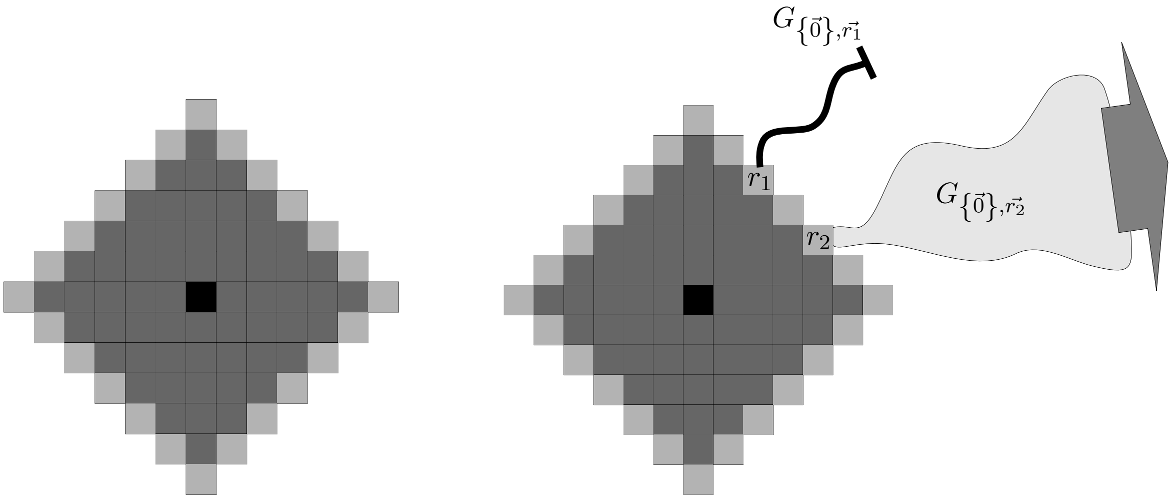
□

Our first main result is as follows.

**Theorem 5.3.5** (first main theorem)**.** Let $\mathcal{T} = (T, \sigma, 1)$ be a locally deterministic tile assembly system. If $X \subsetneq \mathbb{N}^2$ is a discrete self-similar fractal, then $X$ does not weakly self-assemble in $\mathcal{T}$.

*Proof.* Assume that $X$ weakly self-assembles in $\mathcal{T}$. It suffices to prove that $X$ is not a discrete self-similar fractal. If $X$ is finite, then we are done, so assume otherwise. Lemma 5.3.4 tells us that there exists $m \in \mathbb{N}$, such that

$$X = \bigcup_{i=0}^{m-1} X_i,$$



(a) $D$ (dark grey) and $B$ (light grey) centered on $\vec{0}$ (black) for $|T| = 5$

(b) Example finite $G_{\{\vec{0}\},\vec{r_1}}$ and infinite $G_{\{\vec{0}\},\vec{r_2}}$ for $\vec{r_1}$, $\vec{r_2} \in B$

Figure 5.6: Example images of proof of Lemma 5.3.4, (a) shows sets $D$ and $B$, and (b) shows sample finite and infinite $\left\{\vec{0}\right\}$-subgraphs. Note that any infinite $\left\{\vec{0}\right\}$-subgraph must be periodic by Lemma 5.3.2

where, for each $i \in \{0, \ldots, m-1\}$, $X_i$ is either finite or periodic. If, for every $i \in \{0, \ldots, m-1\}$, $X_i$ is infinite, then we are done, because $X$ is a finite union of periodic sets, and the theorem follows from Observation 5.3.3 by taking $C = \varnothing$. However, if there exists $i \in \mathbb{N}$ such that $i \in \{0, \ldots, m-1\}$ and $X_i$ is finite, then we can assume, without loss of generality, that $X_0, \ldots, X_{j-1}$ are finite, and $X_j, \ldots, X_{m-1}$ are infinite. The theorem follows from Observation 5.3.3 by taking

$$C = \left( \bigcup_{i=0}^{j-1} X_i \right) - \left( \bigcup_{i=j}^{m-1} X_i \right).$$

□

We now shift our attention to the strict self-assembly of discrete self-similar fractals at temperature $\tau \geq 2$. Specifically, we exhibit a class $\mathcal{C}$ of (non-tree) "pinch-point" discrete self-similar fractals that do not strictly self-assemble. Note that, unlike for the case $\tau = 1$, our proofs exploit the *geometry* of the fractal as opposed to the nature of self-assembly. In proving our second main result, we assume that $\sigma$ is a stable assembly having a finite domain that contains the origin. We use the following lower bound in the proof of our second main theorem.

**Lemma 5.3.6.** If $X \subseteq \mathbb{Z}^2$ strictly self-assembles in the TAS $\mathcal{T} = (T, \sigma, \tau)$, then

$$|T| \geq \left| \left\{ [B] \;\middle|\; B \text{ is a dom } \sigma\text{-subgraph of } G_X^{\#} \right\} \right|,$$

where $B$ is a dom $\sigma$-subgraph of $G_X^{\#}$, and $[B]$ is the set of all dom $\sigma$-subgraphs of $G_X^{\#}$ that are isomorphic to $B$.

*Proof.* Assume the hypothesis, and let $\alpha$ be the unique terminal assembly satisfying $\alpha \in \mathcal{A}_{\Box}[\mathcal{T}]$. Note that

$$[B] = \left\{ B' \;\middle|\; B' \text{ is a dom } \sigma\text{-subgraph of } G_X^{\#} \text{ and } B' \sim B \right\}.$$

For the purpose of obtaining a contradiction, suppose that

$$|T| < \left| \left\{ [B] \;\middle|\; B \text{ is a dom } \sigma\text{-subgraph of } G_X^{\#} \right\} \right|.$$

By the Pigeonhole Principle, there exists points $\vec{r_1}, \vec{r_2} \in X$ satisfying (1) $\alpha\left(\vec{r_1}\right) = \alpha\left(\vec{r_2}\right)$, and (2) $G_{\text{dom } \sigma, \vec{r_1}} \not\sim G_{\text{dom } \sigma, \vec{r_2}}$. Let $\sigma_1$ be the assembly with dom $\sigma_1 = \{\vec{r_1}\}$, and for all $\vec{u} \in U_2$, define

$$\sigma_1(\vec{r_1})(\vec{u}) = \begin{cases} \left(\text{col}_{\alpha(\vec{r_1})}(\vec{u}), \text{str}_{\alpha(\vec{r_1})}(\vec{u})\right) & \text{if } \vec{r_1} + \vec{u} \in G_{\text{dom } \sigma, \vec{r_1}} \\ (\lambda, 0) & \text{otherwise.} \end{cases}$$

Intuitively, $\sigma_1(\vec{r_1})$ is a modified version of $\alpha\left(\vec{r_k}\right)$ such that the input sides of $\alpha\left(\vec{r_k}\right)$ are zeroed out and the output sides left unchanged. Let $\sigma_2$ be the assembly with dom $\sigma_2 = \{\vec{r_2}\}$, and for all $\vec{u} \in U_2$, define

$$\sigma_2(\vec{r_2})(\vec{u}) = \begin{cases} \left(\text{col}_{\alpha(\vec{r_2})}(\vec{u}), \text{str}_{\alpha(\vec{r_2})}(\vec{u})\right) & \text{if } \vec{r_2} + \vec{u} \in G_{\text{dom } \sigma, \vec{r_2}} \\ (\lambda, 0) & \text{otherwise.} \end{cases}$$

Then $\mathcal{T}_1 = (T, \sigma_1, \tau)$ is a TAS in which $G_{\text{dom } \sigma, \vec{r_1}}$ strictly self-assembles, and $\mathcal{T}_2 = (T, \sigma_2, \tau)$ is a TAS in which $G_{\text{dom } \sigma, \vec{r_2}}$ strictly self-assembles. But this is a contradiction, because $\alpha\left(\vec{r_1}\right) = \alpha\left(\vec{r_2}\right)$ implies that, for all $\vec{u} \in U_2$, $\sigma_1\left(\vec{r_1}\right)\left(\vec{u}\right) = \sigma_2\left(\vec{r_2}\right)\left(\vec{u}\right)$. □

Essentially, this proof sets a lower bound on the size of a tile set which strictly self assembles a shape by relating it to the number of equivalence classes of dom-$\sigma$ subgraphs that appear in that shape. Each such equivalence class is rooted by a single tile, but since it forms a unique shape there must be a unique tile type for each such root.

Note that Theorem 3.2 of [36] is "quantitatively stronger" than Lemma 5.3.6. However, Lemma 5.3.6 applies to a more general class of discrete self-similar (non-tree) fractals. Our second impossibility result is the following.

**Definition.** Let $X \subsetneq \mathbb{N}^2$ be a discrete self-similar fractal. We say that $X$ is a *pinch-point discrete self-similar fractal* if $X$ is a discrete self-similar fractal, generated by $V$, and $V$ satisfies the following three conditions.

1. $\{(0,0),(0,c-1),(c-1,0)\} \subseteq V$.
2. $V \cap (\{1,\dots c-1\} \times \{c-1\}) = \varnothing$.
3. $V \cap (\{c-1\} \times \{1,\dots,c-1\}) = \varnothing$.
4. $G^{\#}_{V}$ is connected.

The generator for a pinch-point fractal has exactly one point in each of its top-most and right-most rows, $(0,c)$ and $(c,0)$, respectively. The other constraint is that the points in the generator are connected. See Figure 5.7 for an example.

We now have the machinery to prove our second main theorem.

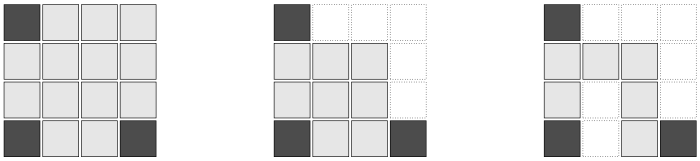
**Theorem 5.3.7** (second main theorem)**.** If $X \subsetneq \mathbb{N}^2$ is a pinch-point discrete self-similar fractal, then $X$ does not strictly self-assemble in the Tile Assembly Model.

*Proof.* By Lemma 5.3.6, it suffices to show that, for any $m \in \mathbb{N}$,

$$\left|\left\{[B] \;\middle|\; B \text{ is a dom } \sigma\text{-subgraph of } G^{\#}_{X}\right\}\right| \geq m,$$

where $B$ is a dom $\sigma$-subgraph of $G^{\#}_{X}$, and $[B]$ is the set of all dom $\sigma$-subgraphs of $G^{\#}_{X}$ that are isomorphic to $B$. Define the points, for all $k \in \mathbb{N}$, $\vec{r}_k = c^k(c(c-1),c-1)$, and let

$$B_k = X \cap \left(\left\{0,\dots c^k-1\right\}^2 + \vec{r}_k\right).$$



(a) Dark gray points must be in the generator

(b) Light gray points in top row and right column cannot be in the generator

(c) The generator must be connected

Figure 5.7: “Construction” of a pinch-point fractal generator

See Figure 5.8 for an example (with $c = 4$) of a $c$-discrete self-similar fractal with the points $\vec{r}_0$ $\vec{r}_1$, and $\vec{r}_2$ highlighted in black.

Conditions (1), (2), and (3) of Definition 5.3 tell us that, for each $k \in \mathbb{N}$, $G^{\#}_{B_k}$ is a dom $\sigma$-subgraph of $G^{\#}_X$ (rooted at $\vec{r}_k$). Furthermore, it is routine to verify that, for all $k_1, k_2 \in \mathbb{N}$, $G^{\#}_{B_{k_1}} \sim G^{\#}_{B_{k_2}} \Longrightarrow k_1 = k_2$. Thus, we have

$$\begin{aligned} m &= \left| \left\{ G^{\#}_{B_k} \;\middle|\; 0 \leq k < m \right\} \right| \\ &\leq \left| \left\{ [B] \;\middle|\; B \text{ is a dom } \sigma\text{-subgraph of } G^{\#}_X \right\} \right|. \end{aligned}$$

□

**Corollary 5.3.8** (Lathrop, Lutz, and Summers [36])**.** The standard discrete Sierpinski triangle **S** does not strictly self-assemble in the Tile Assembly Model.

## 5.4 Every Nice Self-Similar Fractal Has a Fibered Version

In this section, given a nice $c$-discrete self-similar fractal $X \subsetneq \mathbb{N}^2$ (generated by $V$), we define its fibered counterpart $\mathcal{F}(X)$. Intuitively, $\mathcal{F}(X)$ is nearly identical to $X$, but each successive stage of $\mathcal{F}(X)$ is slightly thicker than the equivalent stage of $X$ (see Figure 5.9 for

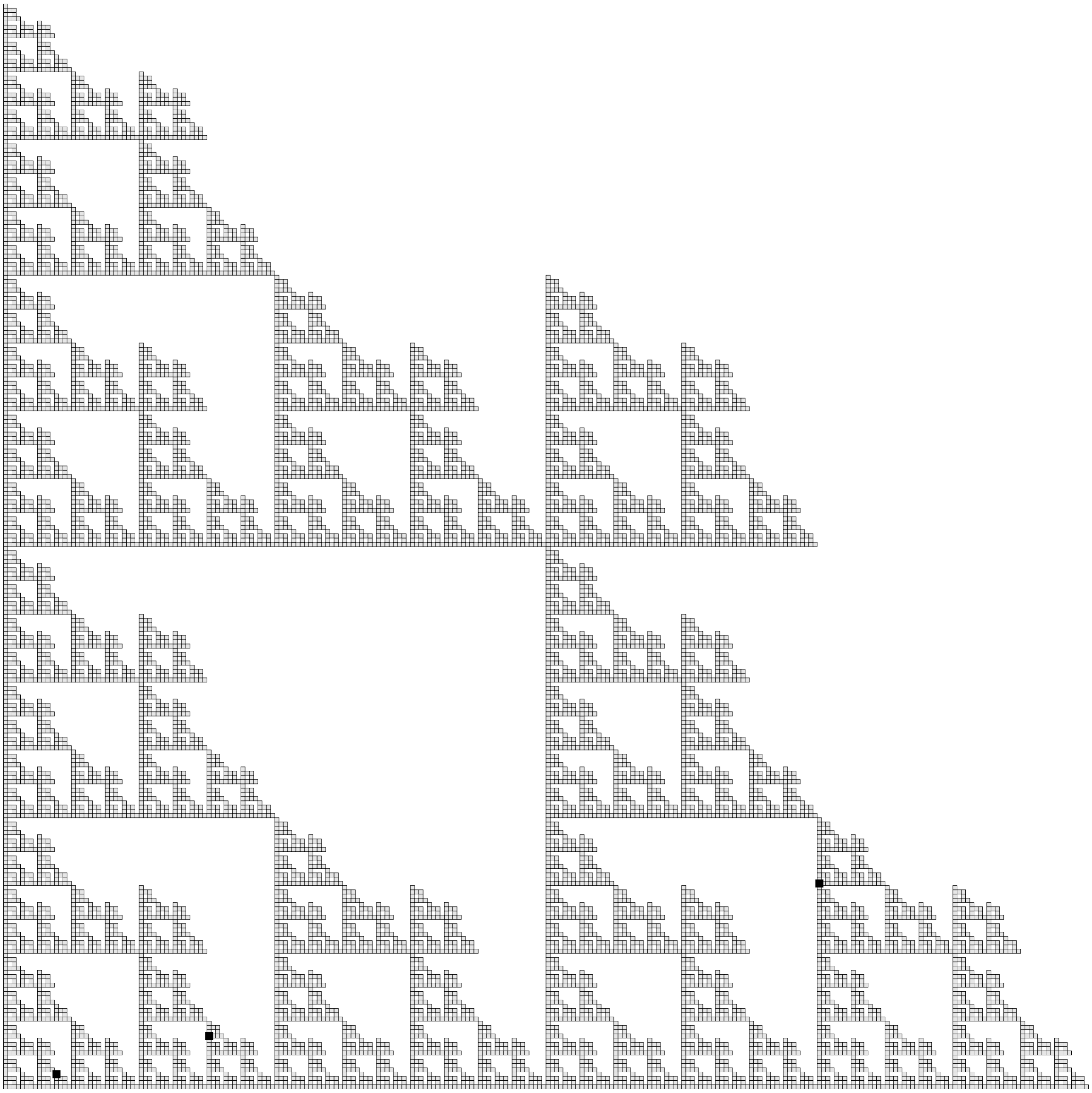


Figure 5.8: An example of a pinch-point discrete self-similar fractal. The black squares are the points $\vec{r}_0$, $\vec{r}_1$ and $\vec{r}_2$, respectively. Notice that the dom $\sigma$-subgraphs, rooted at each of these points, are growing larger. In general, for any pinch-point discrete self-similar fractal, there will exist an infinite sequence of points $\vec{r}_0, \vec{r}_1, \ldots$ at which successively larger dom $\sigma$-subgraphs are rooted.

an example). Our objective is to define sets $F_0, F_1, \ldots \subseteq \mathbb{Z}^2$, sets $T_0, T_1, \ldots \subseteq \mathbb{Z}^2$, and functions $l, f, t : \mathbb{N} \to \mathbb{N}$ with the following meanings.

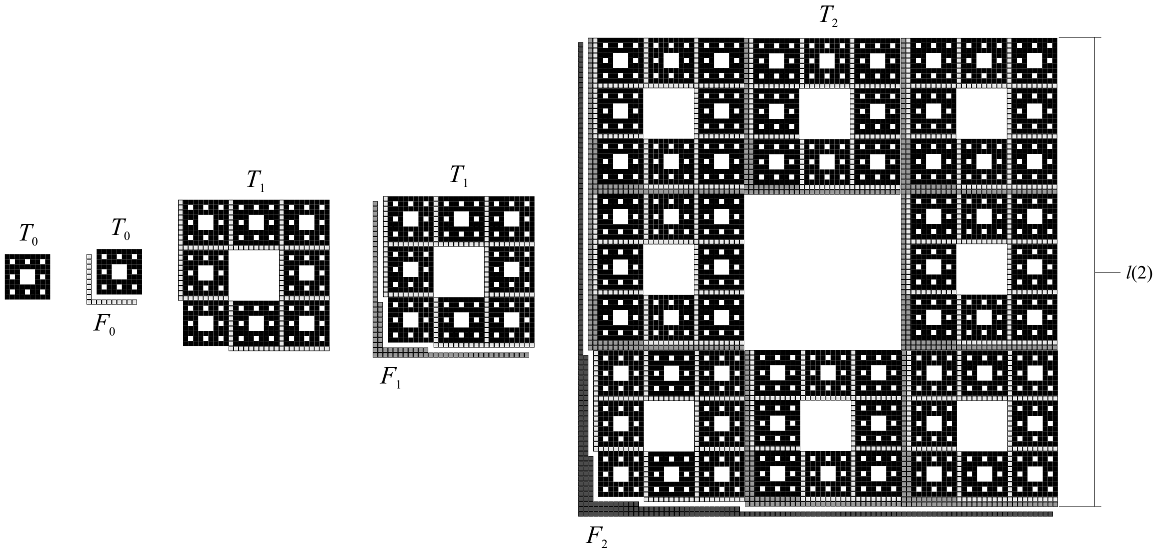


Figure 5.9: Construction of the fibered Sierpinski carpet. The black, dark gray, and light gray tiles represent (possibly translated copies of) $F_0$, $F_1$, and $F_2$, respectively.

1. $T_i$ is the $i^{\text{th}}$ stage of our construction of the fibered version of $X$, i.e., $\mathcal{F}(X)$.

2. $F_i$ is the *fiber* associated with $T_i$. It is the smallest set whose union with $T_i$ has a straight vertical left edge and a straight horizontal bottom edge, together with one additional layer added to these two now-straight edges (see Figure 5.9 for an example of the set $F_0$, $F_1$, and $F_2$ for the fibered version of the discrete Sierpinski carpet).

3. $l(i)$ is the length (number of tiles in) the left (or bottom) edge of $T_i \cup F_i$.

4. $f(i) = |F_i|$.

5. $t(i) = |T_i|$.

These five entities are defined recursively by the equations

$$
\begin{aligned}
&T_0 = X_2 \text{ (the third stage of } X\text{)},\\
&F_0 = \left(\{-1\} \times \left\{-1, \ldots, c^2\right\}\right) \cup \left(\left\{-1, \ldots, c^2\right\} \times \{-1\}\right),\\
&l(0) = c^2 + 1,\ f(0) = 2c^2 + 1,\ t(0) = (|V| + 1)^2,\\
&T_{i+1} = T_i \cup ((T_i \cup F_i) + l(i)V),\\
&F_{i+1} = F_i \cup (\{-i-2\} \times \{-i-2, -i-1, \cdots, l(i+1) - i - 3\})\\
&\cup (\{-i-2, -i-1, \cdots, l(i+1) - i - 3\} \times \{-i-2\}),\\
&l(i+1) = c \cdot l(i) + 1,\\
&f(i+1) = f(i) + c \cdot l(i+1) - 1,\\
&t(i+1) = |V|t(i) + f(i).
\end{aligned}
$$

Finally, we let

$$\mathcal{F}(X) = \bigcup_{i=0}^{\infty} T_i.$$

We have the following "similarity" between $X$ and $\mathcal{F}(X)$.

**Lemma 5.4.1.** If $X \subsetneq \mathbb{N}^2$ is a nice self-similar fractal, then $\mathrm{Dim}_\zeta(X) = \mathrm{Dim}_\zeta(\mathcal{F}(X))$.

*Proof.* Solving the recurrences for $l$, $f$, and $t$, in that order, gives the formulas

$$l(i) = \frac{c^{i+3} - c^{i+2} + c^{i+1} - 1}{c - 1},$$

$$f(i) = \frac{c^{i+5} - c^{i+4} + c^{i+3} - 2c^2 i + 3ci - i - c^5 + 3c^4 - 5c^3 + 3c^2 - 2c + 1}{(c-1)^2},$$

and

$$
\begin{aligned}
t(i) = K \big(&|V|^{i+4} + |V|^{i+3} - |V|^{i+2}c - c^{i+3}|V|^2 - 2c^3|V|i + 2c^2|V|^2 i + c^2|V|i - 3c|V|^2 i + 2c|V|i \\
&- 3|V|^{i+1} - 3|V|^2c^2 - 3|V|^{i+3}c + 5c^2 + 2|V| - 2c - 5c^3 - c^{i+3} + ci - 3c^2 i + 2c^3 i - 3c^5 + 5c^4 \\
&+ c^6 - |V|^2 - c^{i+5} + c^{i+4} - 2|V|c^4 + 3|V|c^2 - 4|V|c + 2|V|c^5 - 2|V|c^3 + 2|V|^2c + 5|V|^2c^3 \\
&- 3|V|^2c^4 - c^6|V| + |V|^2c^5 - |V|i + |V|^2 i - 7c^2|V|^i + 3c|V|^i + 7c^3|V|^i + 4c^5|V|^i - 6c^4|V|^i \\
&- c^6|V|^i - c^3|V|^{i+3} + 3c^2|V|^{i+3} + c^2|V|^{i+4} - 2c|V|^{i+4} + c^3|V|^{i+1} - 4c^5|V|^{i+1} + 7c|V|^{i+1} \\
&- 6c^2|V|^{i+1} + 4c^4|V|^{i+1} + |V|^2c^{i+4} - 2|V|c^{i+4} + 2|V|c^{i+5} + c^6|V|^{i+1} - |V|^2c^{i+5} + 2c^4|V|^{i+2} \\
&-5c^3|V|^{i+2} + 2|V|c^{i+3} + 4c^2|V|^{i+2}\big),
\end{aligned}
$$

where

$$K = \frac{1}{(c-1)^2\,(|V|-1)(|V|-c)}.$$

Note that for every nice discrete self-similar fractal, we have $|V| > c$, whence

$$
\begin{aligned}
\mathrm{Dim}_\zeta(\mathcal{F}(X)) &= \limsup_{n\to\infty} \frac{\log t(n)}{\log l(n)} \\
&= \frac{\log |V|}{\log c} \\
&= \mathrm{Dim}_\zeta(X).
\end{aligned}
$$

□

In the next section we outline a proof that the fibered version of every nice discrete self-similar fractal strictly self-assembles.

## 5.5 Main Construction

Our second main theorem says that the fibered version of every nice discrete self-similar fractal strictly self-assembles in the Tile Assembly Model. It is not known at the time of this writing whether or not there exists a nice discrete self-similar fractal that strictly self-assembles.

**Theorem 5.5.1.** For every nice discrete self-similar fractal $X \subsetneq \mathbb{N}^2$, there exists a directed TAS $\mathcal{T}_{\mathcal{F}(X)}$ in which $\mathcal{F}(X)$ strictly self-assembles.

To prove Theorem 5.5.1, it suffices to exhibit a locally deterministic TAS $\mathcal{T}_{\mathcal{F}(X)} = \left(T_{\mathcal{F}(X)}, \sigma_{\mathcal{F}(X)}, 2\right)$ in which $\mathcal{F}(X)$ strictly self-assembles. Throughout this section, assume that $c \in \mathbb{N}$, and let $X \subsetneq \mathbb{N}$ be a nice $c$-discrete self-similar fractal generated by $V$. We have implemented our construction of $\mathcal{T}_{\mathcal{F}(X)}$ in C++ which is available at the URL `http://www.cs.iastate.edu/~lnsa`. In the next subsection, we provide an intuitive example showing how our construction will ultimately work before we give the formal construction.

We must first define some notation that we will use throughout our construction. We construct a directed spanning tree $T = (V, E)$ of $G^{\#}_V$, rooted at the point $\vec{0} \in V$, satisfying the following properties.

1. For all $\vec{v} \in \{0\} \times \{0, \ldots c-2\}$, $(\vec{v}, \vec{v} + (0,1)) \in E$.

2. For all $\vec{v} \in \{0, \ldots c-2\} \times \{0\}$, $(\vec{v}, \vec{v} + (1,0)) \in E$.

An example of such a spanning tree $T$ is depicted in the middle image of Figure 5.10 for a particular nice discrete self-similar fractal.

**Notation 5.** We define, for each $\vec{v} \in V$, the set

$$\mathrm{OUT}(\vec{v}) = \left\{ \vec{w} \mid (\vec{v}, \vec{w}) \in E \text{ and } \vec{w} \notin (\{0\} \times \{0, \ldots, c-1\} \cup \{0, \ldots, c-1\} \times \{0\}) \right\}.$$

The set $\mathrm{OUT}(\vec{v})$ is the set of vertices, reachable from $\vec{v}$, whose $x$ or $y$-coordinate is not 0. Next, we compute from $T$, the graph $T^{\mathrm{R}} = \left(V, E^{\mathrm{R}}\right)$, where

$$E^{\mathrm{R}} = \left\{ (\vec{v}, \vec{u}) \neq \vec{0} \;\middle|\; (\vec{u}, \vec{v}) \in E \right\} \cup \{((0,1),(0,c-1)),((1,0),(c-1),0)\}.$$

Essentially, $E^{\mathrm{R}}$ is the "reverse" of $E$. Namely, $E^{\mathrm{R}}$ satisfies the following.

1. Every edge in $E$ is reversed.

2. All edges that point to $\vec{0}$ are removed.

3. One edge emanating at $(0,1)$ and one edge emanating at $(1,0)$ are added to create cycles by pointing to $(0, c-1)$ and $(c-1, 0)$, respectively

Figure 5.10 depicts phase 1 of our construction for a particular nice discrete self-similar fractal. Note that the graph $T^{\mathrm{R}}$ is *not* a spanning tree. In fact, $\vec{0} \in V$ is an isolated vertex in $T^{\mathrm{R}}$.

**Notation 6.** For all $\vec{0} \neq \vec{u} \in V$, $\vec{u}_{\mathrm{in}}$ is the unique location $\vec{v} \in V$ satisfying $(\vec{u}, \vec{v}) \in E^{\mathrm{R}}$.

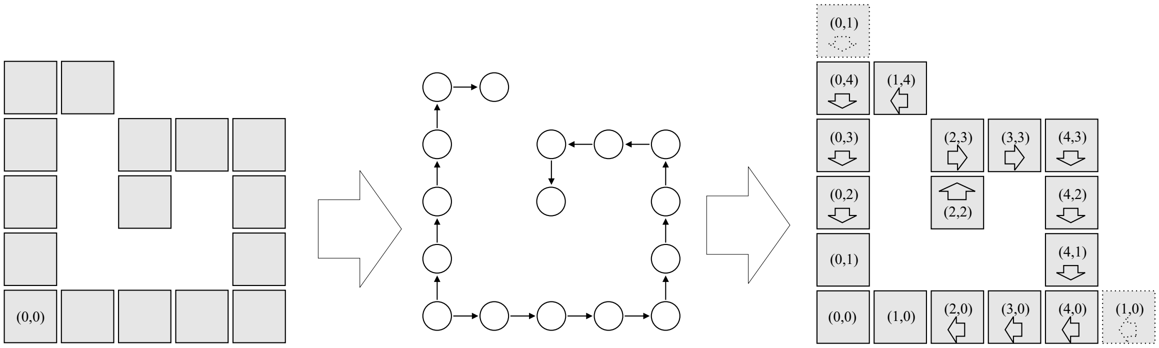


Figure 5.10: Phase 1 of our construction on an example nice discrete self-similar fractal. The left-most image represents the set $V$ - the generator of $X$. The middle image represents the spanning tree $T$. The right-most image represents $T^{\mathrm{R}}$. Notice the two special cases (right-most image) in which we define $(0,1)_{\mathrm{in}}$ and $(1,0)_{\mathrm{in}}$.

In the next sub-section, we will exhibit a characterization of $\mathcal{F}(X)$ in terms of squares and rectangles that will help guide its strict self-assembly.

### 5.5.1 Bar Characterization of $\mathcal{F}(X)$

We now formulate the characterization of $\mathcal{F}(X)$ that guides its strict self-assembly. At the outset, in the notation of section 4, we focus on the manner in which the sets $T_i \cup F_i$ can be constructed from horizontal and vertical bars. Recall that

$$l(i) = \frac{c^{i+3} - c^{i+2} + c^{i+1} - 1}{c - 1}$$

is the length of (number of tiles in) the left or bottom edge of the set $T_i \cup F_i$.

**Definition.** Let $-1 \leq i \in \mathbb{Z}$.

1. The $S_i$-*square* is the set

$$S_i = \{-i-1, \ldots, 0\} \times \{-i-1, \ldots, 0\}.$$

2. The $X_i$-*bar* is the set

$$X_i = \{1, \ldots, l(i) - i - 2\} \times \{-i-1, \ldots, 0\}.$$

3. The $Y_i$-*bar* is the set

$$Y_i = \{-i-1, \ldots, 0\} \times \{1, \ldots, l(i) - i - 2\}.$$

It is clear that the set

$$S_i \cup X_i \cup Y_i$$

is the “outer framework” of $T_i \cup F_i$. Define the *interior of* $T_0$ (i.e., the first stage of $\mathcal{F}(X)$) to be the set

$$I = T_0 - (\{0\} \times \{0, \ldots, c-1\} \cup \{0, \ldots, c-1\} \times \{0\}) .$$

Our attention thus turns to the manner in which smaller and smaller bars are recursively attached to this framework.

We use, for $c \in \mathbb{N}$, the *generalized ruler function*

$$\rho : \mathbb{Z}^+ \to \mathbb{N}$$

defined by the recurrence

$$\begin{aligned} \rho(ck+1) &= 0, \\ \rho(ck) &= \rho(k) + 1 \end{aligned}$$

for all $k \in \mathbb{N}$. It is easy to see that $\rho(n)$ is the (exponent of the) largest power of $c$ that divides $n$. Equivalently, $\rho(n)$ is the number of 0’s lying to the right of the rightmost 1 in the base-$c$ expansion of $n$ [25].

Using the ruler function, we define the function

$$\theta : \mathbb{Z}^+ \to \mathbb{Z}^+$$

by the recurrence

$$\begin{aligned}\theta(1) &= c^2+1,\\ \theta(j+1) &= \theta(j)+\rho(j)+\left(c^2+1\right)\end{aligned}$$

for all $j \in \mathbb{Z}^+$.

See Figure 5.11 for an illustration of the first ten "$\theta$ points" of the fibered Sierpinski triangle.

**Notation 7.** Let $j \in \mathbb{N}$.

1. $\vec{x}_j = (\rho(j) \cdot [\![c \mid j]\!] + (j \mod c) \cdot [\![c \nmid j]\!], 0)$

2. $\vec{y}_j = (0, \rho(j) \cdot [\![c \mid j]\!] + (j \mod c) \cdot [\![c \nmid j]\!])$

The following recursion attaches smaller bars to larger bars in a recursive fashion.

**Definition.** Let $i \in \mathbb{N}$. The *$\theta$-closures* of $X_i$, $Y_i$ and $S_i$ are the sets $\theta(X_i)$, $\theta(Y_i)$ and $\theta_{\vec{a}}(S_i)$ (for each $\vec{a} \in V$) defined, for all $i \in \mathbb{N}$, via the following mutual recursion.

$$\theta\left(X_i\right) = X_i \cup I \cup \bigcup_{j=1}^{c^i-1} \left( \left((\theta(j),1)+\left(\theta\left(Y_{\rho(j)}\right)\cup I\right)\right) \cup \bigcup_{\vec{a}\in\mathrm{OUT}(\vec{x}_j)} \left((\theta(j), l(\rho(j)+1)) + \theta_{\vec{a}}\left(S_{\rho(j)}\right)\right)\right)$$

$$\theta\left(X_0\right) = X_0 \cup I$$

$$\theta\left(Y_i\right) = Y_i \cup \bigcup_{j=1}^{c^i-1} \left( \left((1,\theta(j))+\theta\left(X_{\rho(j)}\right)\right) \cup \bigcup_{\vec{a}\in\mathrm{OUT}(\vec{y}_j)} \left((l(\rho(j)+1), \theta(j)) + \theta_{\vec{a}}\left(S_{\rho(j)}\right)\right)\right)$$

$$\theta\left(Y_0\right) = Y_0$$

$$\theta_{\vec{a}}\left(S_i\right) = S_i \cup \theta(X_i) \cup \theta(Y_i) \cup \bigcup_{\vec{b}\in\mathrm{OUT}(\vec{a})} \left(\theta_{\vec{b}}(S_i) + \left(\vec{b}-\vec{a}\right)\cdot l(i)\right)$$

We have the following characterization of the set $T_i \cup F_i$.

**Lemma 5.5.2.** For all $i \in \mathbb{N}$,

$$T_i \cup F_i = S_i \cup \theta(X_i) \cup \theta(Y_i).$$

We now shift our attention to the global structure of the set $\mathcal{F}(X)$.

**Definition.**

1. The *x-axis* of $\mathcal{F}(X)$ is the set

$$\widetilde{X} = \{\,(m,n) \in \mathcal{F}(X) \mid m > 0, \text{ and } n \leq 0\}.$$

2. The *y-axis* of $\mathcal{F}(X)$ is the set

$$\widetilde{Y} = \{\,(m,n) \in \mathcal{F}(X) \mid m \leq 0, \text{ and } n > 0\}.$$

Intuitively, the $x$-axis of $\mathcal{F}(X)$ is the part of $\mathcal{F}(X)$ that is a “gradually thickening bar” lying on and below the (actual) $x$-axis in $\mathbb{Z}^2$. For technical convenience, we have omitted the origin from this set. Similar remarks apply to the $y$-axis of $\mathcal{F}(X)$. Define the sets

$$\begin{aligned}
\widetilde{X}_{-1} &= \{(1,0),(2,0),(3,0),\ldots(c^2-1,0)\},\\
\widetilde{Y}_{-1} &= \{(0,1),(0,2),(0,3),\ldots(0,c^2-1)\}.
\end{aligned}$$

For each $i \in \mathbb{N}$, define the translations

$$\begin{aligned}
S_i^{\rightarrow} &= (l(i),0) + S_i,\\
S_i^{\uparrow} &= (0,l(i)) + S_i,\\
X_i^{\rightarrow} &= (l(i),0) + X_i,\\
Y_i^{\uparrow} &= (0,l(i)) + Y_i
\end{aligned}$$

of $S_i$, $X_i$, and $Y_i$. It is clear by inspection that $\widetilde{X}$ is the disjoint union of the sets

$$\widetilde{X}_{-1}, S_0^{\rightarrow}, X_0^{\rightarrow}, S_1^{\rightarrow}, X_1^{\rightarrow}, S_2^{\rightarrow}, X_2^{\rightarrow}, \ldots,$$

which are written in their left-to-right order of position in $\widetilde{X}$.

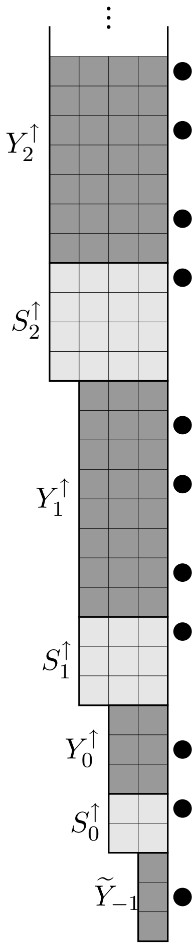


Figure 5.11: The structure of $\widetilde{Y}$ for the "fibered Sierpinski triangle" (see [36] for a more detailed discussion of the fibered Sierpinski triangle). The dots denote the points $(1, \theta(j))$ for $j = 1, \ldots, 10$.

**Definition.** The $\theta$-*closures* of the axes $\widetilde{X}$ and $\widetilde{Y}$ are the sets

$$\theta\left(\widetilde{X}\right) = \widetilde{X} \cup I \cup \bigcup_{j=1}^{\infty} \left( \left((\theta(j), 1) + \left(\theta\left(Y_{\rho(j)}\right) \cup I\right)\right) \cup \bigcup_{\vec{a} \in \mathrm{OUT}(\vec{x}_j)} \left((\theta(j), l(\rho(j)+1)) + \theta_{\vec{a}}\left(S_{\rho(j)}\right)\right) \right)$$

$$\theta\left(\widetilde{Y}\right) = \widetilde{Y} \cup \bigcup_{j=1}^{\infty} \left( \left((1, \theta(j)) + \theta\left(X_{\rho(j)}\right)\right) \cup \bigcup_{\vec{a} \in \mathrm{OUT}(\vec{y}_j)} \left((l(\rho(j)+1), \theta(j)) + \theta_{\vec{a}}\left(S_{\rho(j)}\right)\right) \right)$$

respectively.

Finally, we have the following theorem, which is the bar characterization of $\mathcal{F}(X)$.

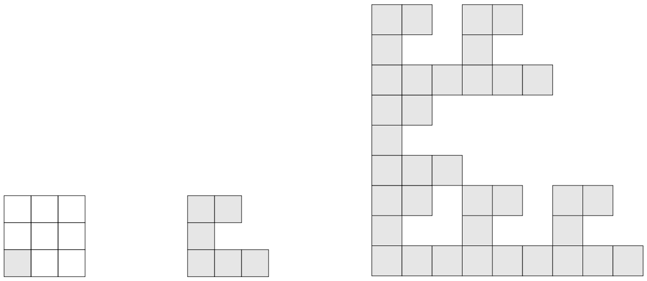
**Theorem 5.5.3.** $\mathcal{F}(X) = \{(0,0)\} \cup \theta\left(\widetilde{X}\right) \cup \theta\left(\widetilde{Y}\right)$.

It is easy to verify that the (recursive) union in Theorem 5.5.3 is disjoint. In subsection 5.5.3, we discuss the construction of a collection of tile sets that carry out the (recursive) strict self-assembly of $\mathcal{F}(X)$ as dictated by Definition 5.5.1

### 5.5.2 Intuitive Example

Before giving the formal construction of Theorem 5.5.1, we provide an intuitive example of how our construction will work.

In Figure 5.12, stages 0 through 2 of a particular nice discrete self-similar fractal are shown. Note that, for this example, $c$ (from Definition 5.2.1) is 3. It is clear that the second stage is constructed by treating stage 1 as a "magnified" version of the origin, then creating 5 additional copies of stage 1 (1 for each point in the generator, $V$, other than the origin) and treating them as magnified versions of those points and translating them accordingly. Each of the (infinitely many) subsequent stages of the discrete self-similar fractal are formed in this manner. Given our example fractal, Figure 5.14 shows the (intended) result of applying our construction. As noted above, we construct a directed spanning tree of $G_V^{\#}$ in order to assign a "type" to each point in the generator, except for $\vec{0}$. The "type" for each point is simply a name derived from the vector to that point from the origin, e.g. $(0, 1)$, and then assigns a single point in the



(a) Stage 0

(b) Stage 1, or $V$

(c) Stage 2

Figure 5.12: First 3 stages of the example nice discrete self-similar fractal

generator as that point's "input" point, as determined by the aforementioned directed spanning tree that we compute. Note that the origin is not used as an input point, but instead the points $(0,1)$ and $(1,0)$ use the points $(0,c-1)$ and $(c-1,0)$, respectively, as their input points. These points are guaranteed to be in $V$ by the definition of nice discrete self-similar fractals. For this example, there are no points with inputs coming from above or to the right, so the case of "reverse growth" (see cases 1(d) and 1(e) of Section 5.5.4) are not utilized. Reverse growth will therefore be ignored in the discussion of this example. Note, however, that handling reverse growth is a relatively straightforward modification of forward growth and does any change major details.

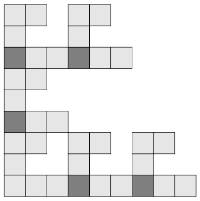
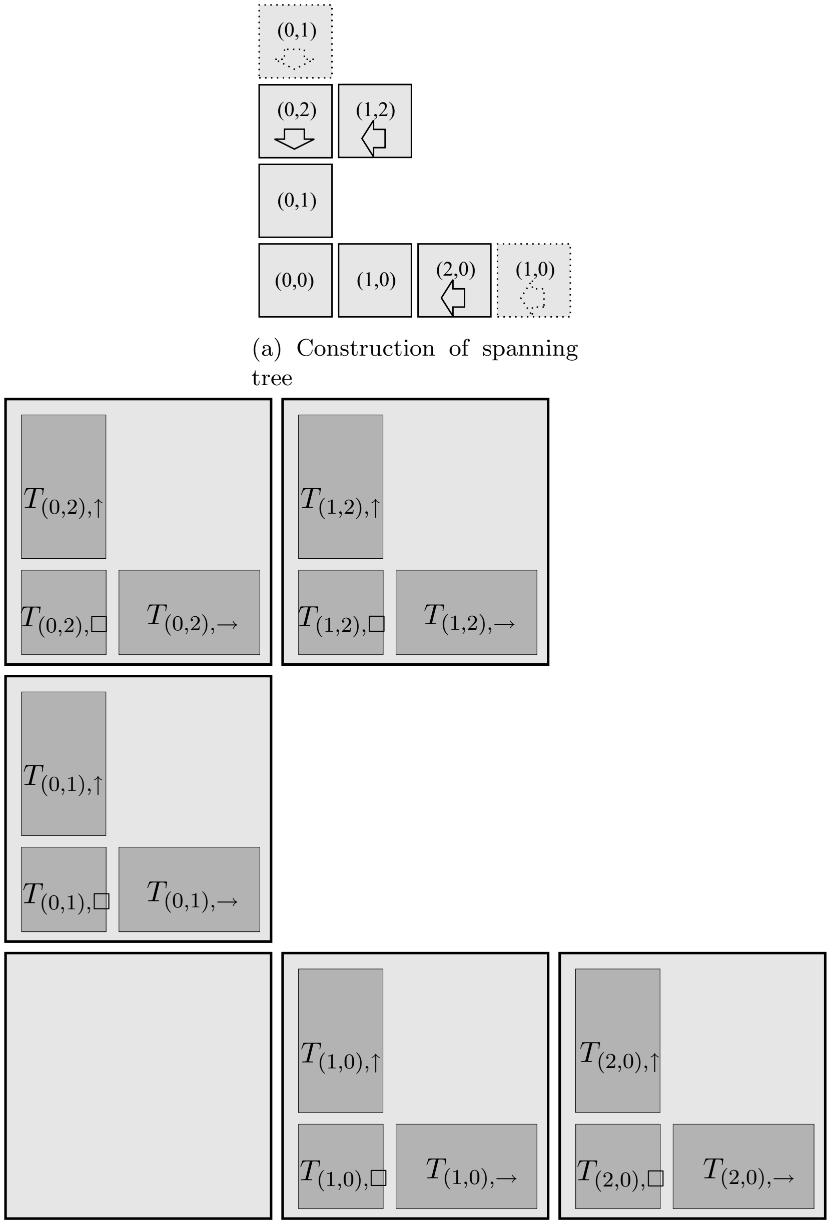


Figure 5.13: Fibered equivalent of stage 2

In the construction of tile types, for each point (excluding the origin) in the generator, several sub-tile sets are generated (see Section 5.5.4 for more details). However, in this example, we will concentrate on the three tile sets in which $S_i$, $Y_i$, and $X_i$ strictly self-assemble (for $i \geq 0$). The first is the set $T_{\vec{v},\square}$ that forms the square at the bottom left of each such location in Figure 5.14b. This is known as the "transition" block because, in the case of forward growth, it is the first portion of each type to assemble for any given stage of the fibered fractal, and its growth is initiated from the assembly of the portion corresponding to the type of its input point. Thus, it *transitions* from one type to another.

The remaining two sub-tile sets $(T_{\vec{v},\uparrow}, T_{\vec{v},\rightarrow})$ that are constructed for each point in the generator self-assemble into vertical and horizontal modified counters. We will only discuss the vertical counter here, but the construction of the horizontal counter is conceptually identical in the sense that it is simply a reflection of its vertical counterpart across the diagonal line $y = x$. The vertical counter tile set for each point type is a modified fixed-width base-$c$, or, for this specific example, base-3 counter. The width is specified by, and identical to, the transition block that initiates its growth. The modification from a standard base-3 counter will be discussed next. Each modified fixed-width counter performs the dual jobs of counting and initiating the growth of the necessary horizontal modified counters to the right. The trivial example of the fibered version of stage 2 of the example fractal is shown in Figure 5.13. (It is trivial because no row of fiber is added until the next stage.) The darker locations represent the transition blocks (in this case, single tiles), which transition from counters of one point type to another. These transition block tiles are vertical width-1 base-3 counters. Despite being only one tile, or digit, wide, they demonstrate two important properties of the counters used in this construction. First, whenever the most significant bit of a counter (the left-most) would have to flip back to 0 (because the counter has reached its maximum value based on its width), instead it stops assembling and initiates the growth of the transition block for the point type, if any, which uses this point type as input from below.

The second property of the counters that can be seen in Figure 5.13 is that, for each value of the least significant (or right-most) bit $b$, if point $(1, b)$ exists in $V$ and $(0, b)$ is its assigned input point, then a hard coded tile type will attach to the right of the counter (see case 1(a)



(a) Construction of spanning tree

(b) Construction of tile types

Figure 5.14: Our construction applied to a particular nice discrete self-similar fractal.

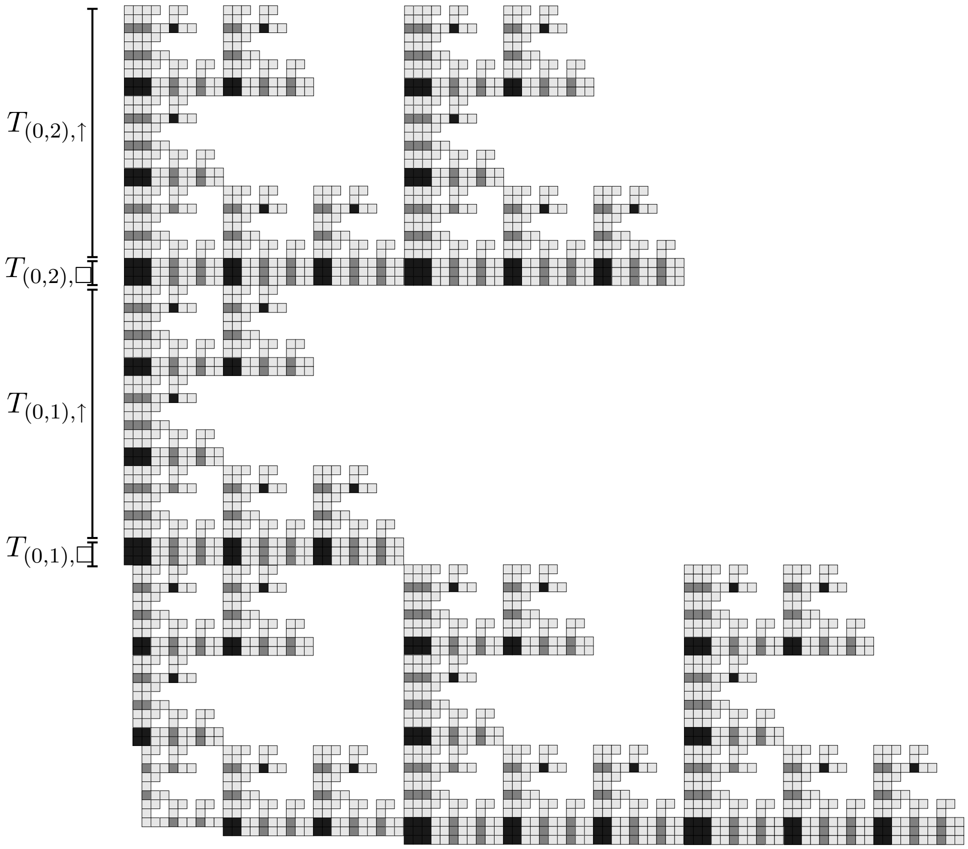


Figure 5.15: Fibered equivalent of the third stage.

of Section 5.5.4). Similarly, hard coded tile types for $(2, b)$ will attach to $(1, b)$, etc. so that each point in $V$ that is not on the $x$ and $y$-axes is tiled. Keep in mind that the corresponding behavior also occurs for the horizontal counters.

Figure 5.15 depicts the completed assembly of the third stage of the fibered fractal. Certain portions of the assembly produced by $T_{(0,1),\square}$, $T_{(0,1),\uparrow}$, $T_{(0,2),\square}$, and $T_{(0,2),\uparrow}$ have been labeled appropriately. Notice that for the first fibered stage, a single additional row of tiles has been added to the right (and bottom for transition blocks), and likewise two rows for the second fibered stage. Each additional row provides the counters with enough bits to count through another factor of $c$, or 3 in this example, over the previous stage.

Another important feature of the modified counters can be seen in Figure 5.15, namely the initiation of horizontal counters and the creation of *spacing rows*. Every time a bit value changes in the counter other than the least significant bit, rows are added which, on their right edges, mimic the behavior of a transition block. In general, if the new value of the flipped bit is $j$, then a transition block assembled via $T_{(0,j),\square}$ is mimicked. This will cause the assembly of the horizontal counter via the set $T_{(0,j),\rightarrow}$ to the right. If the flipped bit is in the $i^{\text{th}}$ row from the right (with the right-most row being the $0^{\text{th}}$ row), then exactly $i - 1$ spacing rows self-assemble before counting resumes, so that the height of the simulated transition block will be exactly $i$ rows. In this manner, by including the same, reflected behavior for the horizontal counters and applying the technique recursively for successively smaller stages (or widths of the transition blocks), the entire internal structure of each fibered stage assembles.

The final point to note is that, whenever a transition block assembles for point types $(0, 1)$ or $(1, 0)$, these transition blocks spontaneously add a row of tiles to their widths and heights (see the tile types defined in Section 5.5.3.3). This is because they represent the beginning of the next stage of the fibered fractal, and thereby they increase the width of the fiber, and subsequently the number of bits in the proceeding counters, by 1.

### 5.5.3 Tile Types for Main Construction

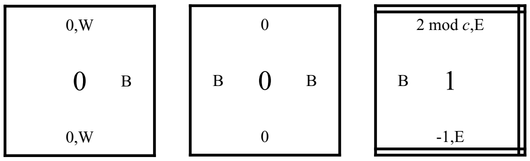
In this section, we give detailed constructions of three fundamental tile sets that we will use in the proof of Theorem 5.5.1.

#### 5.5.3.1 Modified Counter Tile Types

The following tile types implement a modified base-$c$ counter.

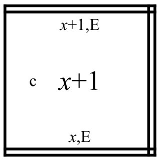
1. The following tile types perform base-$b$ counting.

   (a) The following tile types are used in, and only in, the first row of the counter.

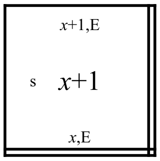


   (b) The following tile types perform addition on the least significant bit.

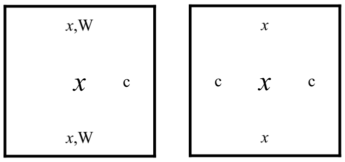
      i. For all $x \in \{0, \ldots, c-3\}$, construct the following tile types:



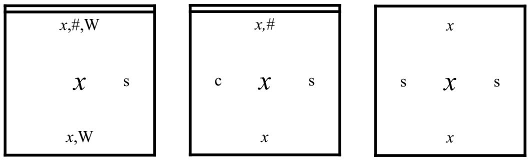
      ii. For $x = c - 2$, add the following tile type:



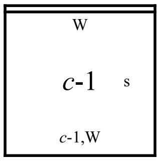
   (c) The following tile types perform copy operations between rows. For all $x \in \{0, \ldots, c-1\}$, construct the following tile types:



   (d) The following tile types search for the right most digit that is not $b - 1$. For all $x \in \{0, \ldots, c-2\}$, construct the following tile types:

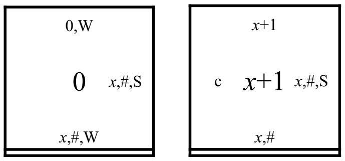


(e) The following tile type is always the left most tile in the last row of the counter.

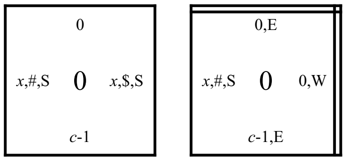


2. The following tile types implement the spacing rows.

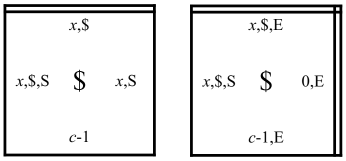
(a) The following tile types initiate the spacing rows. For all $x \in \{0, \ldots, c-2\}$, construct the following tile types:



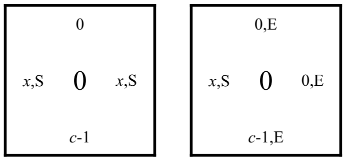
(b) The following tile types are always part of the first spacing row and they initiate the shifting of '$' up and to the right. For all $x \in \{0, \ldots, c-2\}$, construct the following tile types:



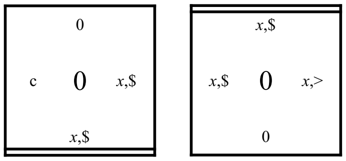
(c) The following tile types are always part of the first spacing row, and shift the '$' over to the right and then up once. For all $x \in \{0, \ldots, c-2\}$, construct the following tile types:



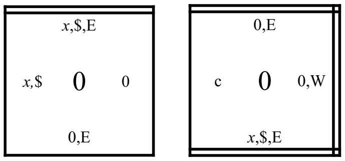
(d) The following tile types construct the remainder of the first spacing row. Note that the 'S' (SOUTH) signal is propagated to the right. For all $x \in \{0, \ldots, c-2\}$, construct the following tile types:



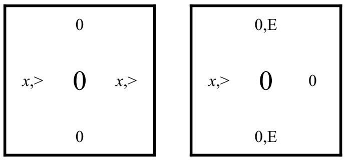
(e) The following tile types shift the '\$' up and to the right. For all $x \in \{0, \ldots, c-2\}$, construct the following tile types:



(f) The following tile types terminate the shifting of '\$'. For all $x \in \{0, \ldots, c-2\}$, construct the following tile types:



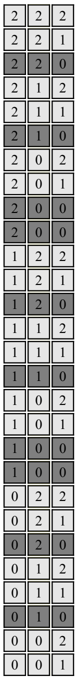
(g) The following tile types copy values up to the next row that are below and to the right of the main diagonal. For all $x \in \{0, \ldots, c-2\}$, construct the following tile types:



The set of tiles given above implements a fixed-width base-$c$ counter that, assuming an initial (appropriately constructed) seed row of width $i \in \mathbb{N}$, performs the following counting scheme: Count each positive integer $j$, satisfying $1 \leq j \leq c^i - 1$, in order but count each integer exactly $J$ times, where

$$J = [\![c \text{ divides } j]\!] \cdot \rho(j) + [\![c \text{ does not divide } j]\!] \cdot 1.$$

The *value* of a row is the number that it represents. We refer to any row whose value is a multiple of $c$ as a *spacing row.* All other rows are *count* rows.



| | | |
|---|---|---|
| 2 | 2 | 2 |
| 2 | 2 | 1 |
| 2 | 2 | 0 |
| 2 | 1 | 2 |
| 2 | 1 | 1 |
| 2 | 1 | 0 |
| 2 | 0 | 2 |
| 2 | 0 | 1 |
| 2 | 0 | 0 |
| 2 | 0 | 0 |
| 1 | 2 | 2 |
| 1 | 2 | 1 |
| 1 | 2 | 0 |
| 1 | 1 | 2 |
| 1 | 1 | 1 |
| 1 | 1 | 0 |
| 1 | 0 | 2 |
| 1 | 0 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 2 | 2 |
| 0 | 2 | 1 |
| 0 | 2 | 0 |
| 0 | 1 | 2 |
| 0 | 1 | 1 |
| 0 | 1 | 0 |
| 0 | 0 | 2 |
| 0 | 0 | 1 |

Figure 5.16: Example of a base-3 modified counting scheme. The darker shaded rows are the spacing rows. Note that details have been purposefully left out.

In our construction, each counter self-assembles on top (or to the right) of a square with the width of the counter being determined by that of the square. It is easy to verify that if the width of the square is $i + 2$, then the modified counter self-assembles a rectangle having a width of $i + 2$ and a height of

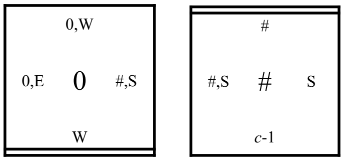
$$\left(c^2 + 1\right) c^i + \frac{c^i - 1}{c - 1} = l(i) - (i + 2),$$

which is exactly $Y_i$. Figure 5.16 shows the counting scheme of a modified base-3 counter of width 3.

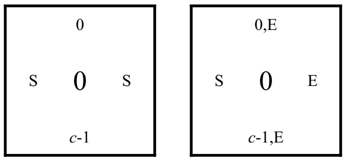
#### 5.5.3.2 Standard Transition Block Tile Types

The following tile types implement a standard transition block. A transition block will essentially act as the seed of a modified counter.

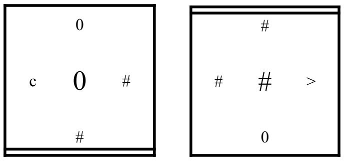
1. The following tile types initiate the growth of the transition block. Construct the following tile types:



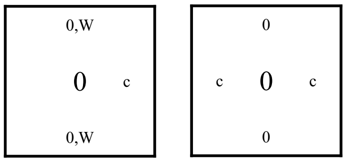
2. The following tile types build the remainder of the first row of the transition block. Construct the following tile types:



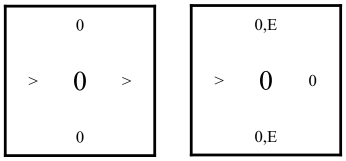
3. The following tile types shift the '#' up and to the right. They essentially fill in the main diagonal of the square. Construct the following tile types:



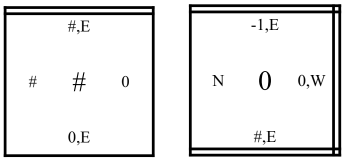
4. The following tile types fill in the upper left half of the square (i.e., tiles above the main diagonal). Construct the following tile types:



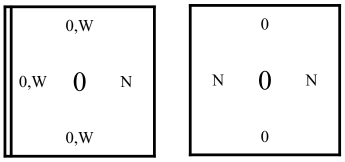
5. The following tile types fill in the lower right half of the square (i.e., tiles below the main diagonal). Construct the following tile types:



6. The following tile types terminate the shifting of the '#'. Construct the following tile types:

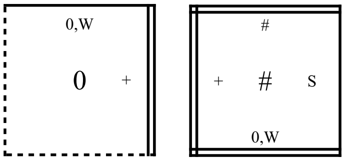


7. The following tile types fill in the last row of the transition block. Construct the following tile types:



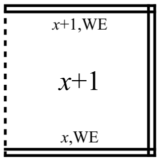
#### 5.5.3.3 Growing Transition Block Tile Types

The tile types that make up a growing transition block are all of the tile types in Section 5.5.3.2, in conjunction with the following tile types, which actually initiate the growth:



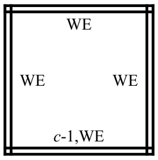
#### 5.5.3.4 Tile Types for Single-bit Modified Counters

For all $x \in \{0, \ldots c-2\}$, define the following (single-row modified counter) counting tile types.



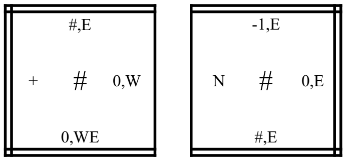
#### 5.5.3.5 Tile Types for Single-bit Transition Blocks

The following tile type is the last tile to attach in a single-row modified counter.



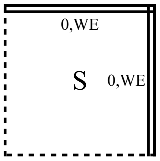
#### 5.5.3.6 The First Growing Transition Block

The following tile types initiate the growth of the first transition block.



#### 5.5.3.7 The Seed Tile Type

The following tile type is the seed tile type for our construction.



### 5.5.4 Construction of $T_{\mathcal{F}(X)}$

In this subsection, we construct the tile set $T_{\mathcal{F}(X)}$. Recall that $(0,0) \in V \subsetneq \{0\ldots,c-1\} \times \{0,\ldots,c-1\}$ is the generator of $X$. For each $\vec{v} \in V$, we construct the tile set

$$T_{\vec{v}} = T_{\vec{v},\Box} \cup T_{\vec{v},\uparrow} \cup T_{\vec{v},\rightarrow} \cup T_{\vec{v},\Box,\text{init}} \cup T_{\vec{v},\uparrow,\text{init}} \cup T_{\vec{v},\rightarrow,\text{init}} \cup T_{\vec{v},*}$$

Note that the tile sets $T_{\vec{v},\Box}$, $T_{\vec{v},\uparrow}$, and $T_{\vec{v},\rightarrow}$ make up the bulk of the tile set $T_{\vec{v}}$. The tile sets $T_{\vec{v},\Box,\text{init}}$, $T_{\vec{v},\uparrow,\text{init}}$, $T_{\vec{v},\rightarrow,\text{init}}$, and $T_{\vec{v},*}$ are used exclusively to self-assemble the initial stage and internal structure of $\mathcal{F}(X)$.

We first define some standard operations on (sets of) tile types that we will ultimately apply to the tile types that we defined in Sections 5.5.3.1, 5.5.3.2, 5.5.3.3, 5.5.3.4, 5.5.3.5, and 5.5.3.7.

**Definition.** Let $t$ be a tile type, and $T$ be a set of tile types.

1. The *reflection of $t$ about the line $y = x$*, written as $R_{y=x}(t)$, is the tile type $t'$ satisfying, for all $(x, y) \in U_2$, $t'(y, x) = t(x, y)$; $R_{y=x}(T) = \{\, R_{y=x}(t) \mid t \in T\}$.

2. The *reflection of $t$ about the $y$-axis*, written as $R_{y\text{-axis}}(t)$, is the tile type $t'$ satisfying, for all $\vec{u} \in \{(-1, 0), (1, 0)\}$, $t'(\vec{u}) = t(-\vec{u})$; $R_{y\text{-axis}}(T) = \{\, R_{y\text{-axis}}(t) \mid t \in T\}$.

3. The *reflection of $t$ about the $x$-axis*, written as $R_{x\text{-axis}}(t)$, is the tile type $t'$ satisfying, for all $\vec{u} \in \{(0, -1), (0, 1)\}$, $t'(\vec{u}) = t(-\vec{u})$; $R_{x\text{-axis}}(T) = \{\, R_{x\text{-axis}}(t) \mid t \in T\}$.

**Definition.** Let $t$ be a tile type, $U \subseteq U_2$ be a set of unit vectors, and $s \in \Sigma^*$. The *concatenation of $s$ to the glue color on each side $\vec{u} \in U$ of $t$*, written as $C(t, U, s)$, is the tile type $t'$ satisfying, for all $\vec{u} \in U$, $t'(\vec{u}) = (\text{col}_t(\vec{u}) \circ s, \text{str}_t(\vec{u}))$.

We now have the tools that we need to construct the set $T_{\vec{v}}$. When building the set $T_{\vec{v}}$, we consider the following cases.

1. $\vec{v} \in V - \{(0, 0), (0, 1), (1, 0)\}$. Let $\vec{u} = \vec{v}_{\text{in}} - \vec{v}$.

   (a) (hard-coded internal structure) If $\vec{v} \in (\{0\} \times \{0, \ldots, c-1\} \cup \{0, \ldots, c-1\} \times \{0\})$, then let $T_{\vec{v},*} = \{t_{\vec{v},*}\}$, where, $t_{\vec{v},*}(\vec{u}) = (\vec{v}, 2)$, for all $\vec{u}' \in U_2 - \{\vec{u}\}$ such that $(\vec{v}, \vec{v} + \vec{u}') \in E$, $t_{\vec{v},*}(\vec{u}') = (\vec{v} + \vec{u}', 2)$, and, for all $\vec{u}' \in U_2 - \{\vec{u}\}$ such that $(\vec{v}, \vec{v} + \vec{u}') \notin E$, $t_{\vec{v},*}(\vec{u}') = (\lambda, 0)$.

   (b) (forward growth) We label this case as "forward growth" because the transition block of type "$\vec{v}$ " will self-assemble before both counters of the same type. Furthermore, both counters will not grow *toward* from the $x$ or $y$-axis.

   If $\vec{u} = (-1, 0)$, then we construct $T_{\vec{v},\Box}$, $T_{\vec{v},\uparrow}$, and $T_{\vec{v},\rightarrow}$ as follows.

   i. Define the following sets of tile types.

$$
\begin{aligned}
T_{\S 5.5.3.2} &= \{t \mid t \text{ is a tile type defined in Section 5.5.3.2}\} \\
T_{\S 5.5.3.2,\text{S}} &= \left\{t \,\middle|\, t \in T_{\S 5.5.3.2} \text{ and the symbol `S' is in some glue color of } t\right\} \\
T_{\S 5.5.3.5} &= \{t \mid t \text{ is a tile type defined in Section 5.5.3.5}\}
\end{aligned}
$$

Let

$$\begin{aligned}
T_{\vec{v},\Box} &= R_{y=x}(C\left(T_{\S5.5.3.2} - T_{\S5.5.3.2,\mathrm{S}}, U_2, \vec{v}\right) \cup \\
&\qquad C\left(C\left(T_{\S5.5.3.2,\mathrm{S}}, \{(0,-1)\}, \vec{v}_{\mathrm{in}}\right), U_2 - \{(0,-1)\}, \vec{v}\right)). \\
T_{\vec{v},\Box,\mathrm{init}} &= R_{y=x}(C(C(T_{\S5.5.3.5}, U_2 - \{(0,-1)\}, \vec{v}_{\mathrm{in}}), U_2 - \{(0,-1)\}, \vec{v}))
\end{aligned}$$

This ensures that the transition block of "type $\vec{v}$ " binds to the horizontal counter of type "$\vec{v}_{\mathrm{in}}$."

ii. Define the following sets of tile types.

$$\begin{aligned}
T_{\S5.5.3.1} &= \{t \mid t \text{ is a tile type defined in Section 5.5.3.1}\} \\
T_{\S5.5.3.4} &= \{t \mid t \text{ is a tile type defined in Section 5.5.3.4}\} \\
T_{\S5.5.3.1,\mathrm{Counting}} &= \{t \mid t \in T_{\S5.5.3.1} \text{ and } t \text{ is defined in Group 1}\} \\
T_{\S5.5.3.1,\mathrm{Spacing}} &= T_{\S5.5.3.1} - T_{\S5.5.3.1,\mathrm{Counting}} \\
T_{\S5.5.3.1,\mathrm{E}} &= \left\{t \;\middle|\; t \in T_{\S5.5.3.1} \text{ and the symbol `E' is in a glue color of } t\right\}
\end{aligned}$$

Let

$$\begin{aligned}
T_{\vec{v},\uparrow} &= C\left(T_{\S5.5.3.1} - T_{\S5.5.3.1,\mathrm{E}}, U_2, \vec{v}\right) \cup \\
&\quad C(C(T_{\S5.5.3.1,\mathrm{Spacing}} \cap T_{\S5.5.3.1,\mathrm{E}}, \{(1,0)\}, (0,x)), U_2 - \\
&\qquad \{(1,0)\}, \vec{v}) \cup \\
&\quad C(C(T_{\S5.5.3.1,\mathrm{Counting}} \cap T_{\S5.5.3.1,\mathrm{E}}, \{(1,0)\}, (1,x+1)), U_2 - \\
&\qquad \{(1,0)\}, \vec{v}) \\
T_{\vec{v},\uparrow,\mathrm{init}} &= C(C(T_{\S5.5.3.4}, \{(1,0)\}, (1,x)), U_2 - \{(1,0)\}, \vec{v})
\end{aligned}$$

iii. Let

$$\begin{aligned}
T_{\vec{v},\rightarrow} &= R_{y=x}(C\left(T_{\S 5.5.3.1} - T_{\S 5.5.3.1,\mathrm{E}}, U_2, \vec{v}\right) \cup \\
&\quad C(C(T_{\S 5.5.3.1,\mathrm{Spacing}} \cap T_{\S 5.5.3.1,\mathrm{E}}, \{(1,0)\}, (x,0)), U_2 - \\
&\qquad \{(1,0)\}, \vec{v}) \cup \\
&\quad C(C(T_{\S 5.5.3.1,\mathrm{Counting}} \cap T_{\S 5.5.3.1,\mathrm{E}}, \{(1,0)\}, (x+1,1)), U_2 - \\
&\qquad \{(1,0)\}, \vec{v})) \\
T_{\vec{v},\rightarrow,\mathrm{init}} &= R_{y=x}(C(C(T_{\S 5.5.3.4}, \{(0,1)\}, (x,1)), U_2 - \{(0,1)\}, \vec{v}))
\end{aligned}$$

(c) (forward growth) If $\vec{u} = (0,-1)$, then we construct $T_{\vec{v},\Box}$, $T_{\vec{v},\uparrow}$, and $T_{\vec{v},\rightarrow}$ as follows.

i. Let

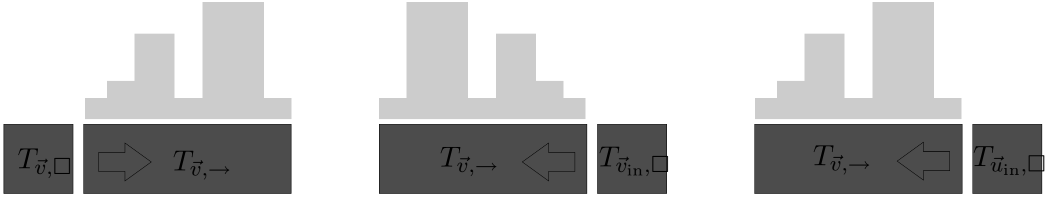
$$\begin{aligned}
T_{\vec{v},\Box} &= C\left(T_{\S 5.5.3.2} - T_{\S 5.5.3.2,\mathrm{S}}, U_2, \vec{v}\right) \cup \\
&\quad C\left(C\left(T_{\S 5.5.3.2,\mathrm{S}}, \{(0,-1)\}, \vec{v}_{\mathrm{in}}\right), U_2 - \{(0,-1)\}, \vec{v}\right) \\
T_{\vec{v},\Box,\mathrm{init}} &= C(C(T_{\S 5.5.3.5}, U_2 - \{(0,-1)\}, \vec{v}_{\mathrm{in}}), U_2 - \{(0,-1)\}, \vec{v})
\end{aligned}$$

This ensures that the transition block of "type $\vec{v}$ " binds to the vertical counter of type "$\vec{v}_{\mathrm{in}}$."

ii. Let $T_{\vec{v},\uparrow}$, $T_{\vec{v},\rightarrow}$, $T_{\vec{v},\uparrow,\mathrm{init}}$, and $T_{\vec{v},\rightarrow,\mathrm{init}}$ be defined as they were in case 2(a).

(d) (reverse growth) We label this case as "reverse growth" because one of the counters that we construct will grow *toward* the $x$ or $y$-axis. Furthermore, the transition block of type "$\vec{v}$ " will not self-assemble before both counters of the same type. We must take special care when constructing the tile types for a reverse growing counter (and transition block) because nice discrete self-similar fractals need not be symmetric. See Figure 5.17 for an illustration of reverse growth.

If $\vec{u} = (1,0)$, then we construct $T_{\vec{v},\Box}$, $T_{\vec{v},\uparrow}$, and $T_{\vec{v},\rightarrow}$ as follows.


$T_{\vec{v},\square}$
$T_{\vec{v},\rightarrow}$
$T_{\vec{v},\rightarrow}$
$T_{\vec{v}_{\text{in}},\square}$
$T_{\vec{v},\rightarrow}$
$T_{\vec{u}_{\text{in}},\square}$

(a) (b) (c)

Figure 5.17: (a) depicts forward growth, (b) shows what happens if the tile set $T_{\vec{v},\rightarrow}$ were to simply "count in reverse," and (c) is the desired result (that we achieve in our construction).

i. Define the following set of tile types.

$$\begin{aligned}
T_{\S 5.5.3.1,\text{B}} &= \left\{t \;\middle|\; t \in T_{\S 5.5.3.1} \text{ and the symbol `B' is in some glue color of } t\right\} \\
T_{\S 5.5.3.4,x=0} &= \left\{t \;\middle|\; t \in T_{\S 5.5.3.4} \text{ and } x = 0\right\}
\end{aligned}$$

Let

$$\begin{aligned}
T_{\vec{v},\rightarrow} &= R_{y\text{-axis}}(R_{y=x}(C\left(T_{\S 5.5.3.1} - \left(T_{\S 5.5.3.1,\text{E}} \cup T_{\S 5.5.3.1,\text{B}}\right), U_2, \vec{v}\right) \cup \\
&\quad C(C(T_{\S 5.5.3.1,\text{Spacing}} \cap T_{\S 5.5.3.1,\text{E}}, \{(1,0)\}, (c-x,0)), U_2 - \\
&\qquad \{(1,0)\}, \vec{v}) \cup \\
&\quad C(C(T_{\S 5.5.3.1,\text{Counting}} \cap T_{\S 5.5.3.1,\text{E}}, \{(1,0)\}, (c-x+1,1)), U_2 - \\
&\qquad \{(1,0)\}, \vec{v}) \cup \\
&\quad C\left(C\left(T_{\S 5.5.3.1,\text{B}}, \{(0,-1)\}, \vec{v}_{\text{in}}\right), U_2 - \{(0,-1)\}, \vec{v}\right))) \\
T_{\vec{v},\rightarrow,\text{init}} &= R_{y\text{-axis}}(R_{y=x}(C(C(T_{\S 5.5.3.4} - T_{\S 5.5.3.4,x=0}, \{(1,0)\}, (x+1,0)), U_2 - \\
&\qquad \{(1,0)\}, \vec{v}) \cup \\
&\quad C(C(C(T_{\S 5.5.3.4,x=0}, \{(0,-1)\}, \vec{v}_{\text{in}}), \{(0,1)\}, (c-x+1,0)), \\
&\qquad U_2 - \{(0,-1),(1,0)\}, \vec{v})))
\end{aligned}$$

ii. Let

$$\begin{aligned} T_{\vec{v},\square} &= R_{y\text{-axis}}(R_{y=x}(C\left(T_{\S 5.5.3.2}, U_2, \vec{v}\right))) \\ T_{\vec{v},\square,\text{init}} &= R_{y\text{-axis}}(R_{y=x}(C\left(T_{\S 5.5.3.5}, U_2, \vec{v}\right))) \end{aligned}$$

This ensures that the (reverse growing) transition block of "type $\vec{v}$ " binds to the (reverse growing) horizontal counter of type "$\vec{v}$."

iii. Let $T_{\vec{v},\uparrow}$ and $T_{\vec{v},\uparrow,\text{init}}$ be defined as they were in case 2(a).

(e) (reverse growth) If $\vec{u} = (0,1)$, then we construct $T_{\vec{v},\square}$, $T_{\vec{v},\uparrow}$, and $T_{\vec{v},\rightarrow}$ as follows.

i. Let

$$\begin{aligned} T_{\vec{v},\uparrow} &= R_{x\text{-axis}}(C\left(T_{\S 5.5.3.1} - T_{\S 5.5.3.1,\text{E}}, U_2, \vec{v}\right) \cup \\ &\quad C(C(T_{\S 5.5.3.1,\text{Spacing}} \cap T_{\S 5.5.3.1,\text{E}}, \{(1,0)\}, (0, c-x)), U_2 - \\ &\qquad \{(1,0)\}, \vec{v}) \cup \\ &\quad C(C(T_{\S 5.5.3.1,\text{Counting}} \cap T_{\S 5.5.3.1,\text{E}}, \{(1,0)\}, (1, c-x+1)), U_2 - \\ &\qquad \{(1,0)\}, \vec{v})) \\ T_{\vec{v},\uparrow,\text{init}} &= R_{x\text{-axis}}(R_{y=x}(C(C(T_{\S 5.5.3.4} - T_{\S 5.5.3.4,x=0}, \{(1,0)\}, (0, x+1)), U_2 - \\ &\qquad \{(1,0)\}, \vec{v}) \cup \\ &\quad C(C(C(T_{\S 5.5.3.4,x=0}, \{(0,-1)\}, \vec{v}_{\text{in}}), \\ &\qquad \{(0,1)\}, (0, c-x+1)), U_2 - \{(0,-1),(1,0)\}, \vec{v}))) \end{aligned}$$

This ensures that the (reverse growing) vertical counter of "type $\vec{v}$ " binds to the transition block of type "$\vec{v}$."

ii. Let

$$\begin{aligned} T_{\vec{v},\square} &= R_{x\text{-axis}}(C\left(T_{\S 5.5.3.2}, U_2, \vec{v}\right) \\ T_{\vec{v},\square,\text{init}} &= R_{x\text{-axis}}(C\left(T_{\S 5.5.3.5}, U_2, \vec{v}\right)) \end{aligned}$$

This ensures that the (reverse growing) transition block of type "$\vec{v}$ " binds to the (reverse growing) vertical counter of type "$\vec{v}$."

iii. Let $T_{\vec{v},\rightarrow}$ and $T_{\vec{v},\rightarrow,\text{init}}$ be defined as they were in case 2(a).

2. $\vec{v} \in \{(0,1),(1,0)\}$. Let $\vec{u} = \vec{v}_{\text{in}} - \vec{v}$.

(a) (forward growth) If $\vec{u} = (0,-1)$, then we construct $T_{\vec{v},\Box}$, $T_{\vec{v},\uparrow}$, and $T_{\vec{v},\rightarrow}$ as follows.

i. Define the following sets of tile types.

$$\begin{aligned} T_{\S 5.5.3.3} &= \{t \mid t \text{ is a tile type defined in Section 5.5.3.3}\} \\ T_{\S 5.5.3.6} &= \{t \mid t \text{ is a tile type defined in Section 5.5.3.6}\} \end{aligned}$$

Let

$$\begin{aligned} T_{\vec{v},\Box} &= C\left(T_{\S 5.5.3.2} - (T_{\S 5.5.3.2,\text{B}}), U_2, \vec{v}\right) \cup \\ &\quad C\left(C\left(T_{\S 5.5.3.2,\text{B}} \cup T_{\S 5.5.3.3}, \{(0,-1)\}, \vec{v}_{\text{in}}\right), U_2 - \{(0,-1)\}, \vec{v}\right) \\ T_{\vec{v},\Box,\text{init}} &= C(T_{\S 5.5.3.2} \cup T_{\S 5.5.3.3}, U_2, \vec{v}) \cup \\ &\quad C\left(C\left(T_{\S 5.5.3.6}, \{(0,-1)\}, (0,0)\right), U_2 - \{(0,-1)\}, \vec{v}\right) \end{aligned}$$

This ensures that the growing transition block of "type $\vec{v}$ " binds to the horizontal counter of type "$\vec{v}_{\text{in}}$."

ii. Let $T_{\vec{v},\uparrow}$, $T_{\vec{v},\rightarrow}$, $T_{\vec{v},\uparrow,\text{init}}$, and $T_{\vec{v},\rightarrow,\text{init}}$ be defined as they were in case 2(a).

(b) (forward growth) If $\vec{u} = (-1,0)$, then we construct $T_{\vec{v},\Box}$, $T_{\vec{v},\uparrow}$, and $T_{\vec{v},\rightarrow}$ as follows.

i. Let

$$\begin{aligned} T_{\vec{v},\square} &= R_{y=x}(C\left(T_{\S5.5.3.2} - (T_{\S5.5.3.2,\mathrm{B}} \cup T_{\S5.5.3.3}), U_2, \vec{v}\right) \cup \\ &\quad C\left(C\left(T_{\S5.5.3.2,\mathrm{B}} \cup T_{\S5.5.3.3}, \{(0,-1)\}, \vec{v}_{\mathrm{in}}\right), U_2 - \{(0,-1)\}, \vec{v}\right)) \\ T_{\vec{v},\square,\mathrm{init}} &= R_{y=x}(C(T_{\S5.5.3.2} \cup T_{\S5.5.3.3}, U_2, \vec{v}) \cup \\ &\quad C(C(T_{\S5.5.3.6}, \{(0,-1)\}, (0,0)), U_2 - \{(0,-1)\}, \vec{v})) \end{aligned}$$

This ensures that the growing transition block of "type $\vec{v}$ " binds to the horizontal counter of type "$\vec{v}_{\mathrm{in}}$."

ii. Let $T_{\vec{v},\uparrow}$, $T_{\vec{v},\rightarrow}$, $T_{\vec{v},\uparrow,\mathrm{init}}$, and $T_{\vec{v},\rightarrow,\mathrm{init}}$ be defined as they were in case 2(a).

Let $T_{\mathrm{seed}}$ be the singleton set containing the seed tile defined in Section 5.5.3.7, and let

$$T_{\vec{0}} = C\left(T_{\mathrm{seed}}, \{(1,0),(0,1)\}, (0,0)\right).$$

Finally, we have

$$T_{\mathcal{F}(X)} = \bigcup_{\vec{v}\in V} T_{\vec{v}},$$

Figure 5.18 gives a visual interpretation of our construction. Our final TAS is $\mathcal{T}_{\mathcal{F}(X)} = \left(T_{\mathcal{F}(X)}, \sigma, 2\right)$, where $\sigma$ consists of a single "seed" tile type placed at the origin. In general, if $X \subsetneq \mathbb{N}^2$ is a nice discrete self-similar fractal generated by $V$, then $\left|T_{\mathcal{F}(X)}\right| = O(|V|)$. Unfortunately, the hidden constant is rather large. For instance, our construction yields a tile set of 5983 tile types for the discrete self-similar fractal generated by the points in the left-most image in Figure 5.18.

### 5.5.5 Correctness of Construction

It is routine to verify that our TAS $\mathcal{T}_{\mathcal{F}(X)} = (T_{\mathcal{F}(X)}, \sigma_{\mathcal{F}(X)}, 2)$ is locally deterministic *and* self-assembles $\mathcal{F}(X)$ according to the recursion given in Definition 5.5.1.

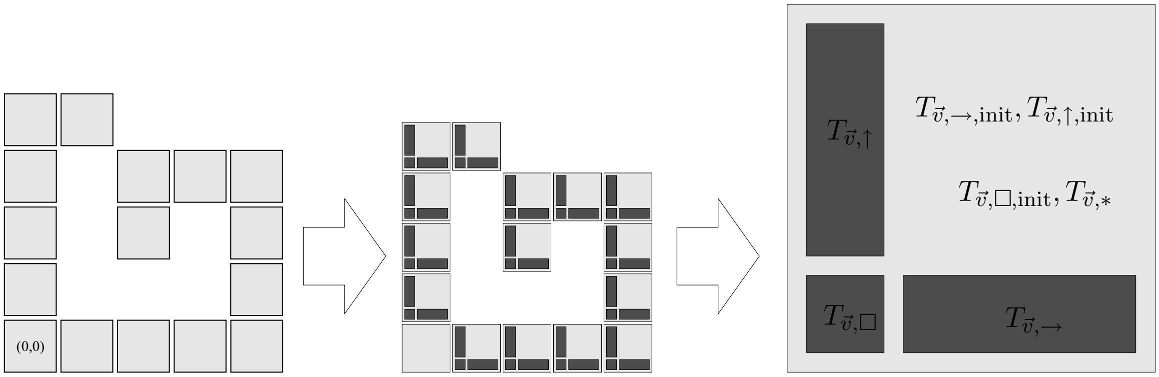

(0,0)
$T_{\vec{v},\uparrow}$
$T_{\vec{v},\rightarrow,\text{init}}, T_{\vec{v},\uparrow,\text{init}}$
$T_{\vec{v},\Box,\text{init}}, T_{\vec{v},*}$
$T_{\vec{v},\Box}$
$T_{\vec{v},\rightarrow}$

Figure 5.18: Let $V$ be the set of points in the left-most image. The first arrow represents our construction. The second arrow shows a magnified view of a particular point in $V$. Each point $\vec{0} \neq \vec{v} \in V$ can be viewed (in principle) as several sub-tile sets: $T_{\vec{v},\Box}$, $T_{\vec{v},\rightarrow}$, $T_{\vec{v},\uparrow}$, $T_{\vec{v},\Box,\text{init}}$, $T_{\vec{v},\rightarrow,\text{init}}$, $T_{\vec{v},\uparrow,\text{init}}$, and $T_{\vec{v},*}$.

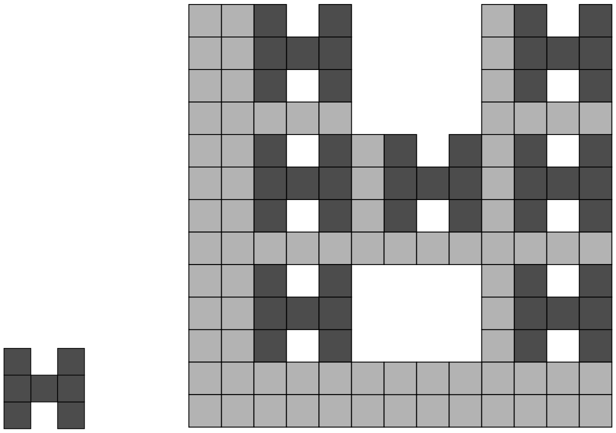
## 5.6 Conclusion

The Tile Assembly Model is a powerful and robust model that abstracts laboratory-based DNA tile self-assembly. The model exhibits rich theoretical behavior in that it allows for the self-assembly of a computationally and geometrically diverse set of assemblies that form intricate shapes or perform (potentially Turing universal) computation. However, as our first two main results show, there are limitations to the kinds of shapes and patterns that can, even theoretically, self-assemble in the TAM.

In our first main (impossibility) result, we proved that non-trivial discrete self-similar fractals do not self-assemble (in either the strict or the weak sense) in any temperature 1 tile assembly system that is locally deterministic. The assumption of local determinism allows one to reason about the nature of self-assembly at temperature 1. From this, we derived that *any* shape or pattern that self-assembles in a locally deterministic temperature 1 tile assembly system is necessarily too simple to be a fractal. At the time of this writing, the question of whether or not the assumption of local determinism can be removed remains open.

In our second main theorem, also an impossibility result, we proved that certain kinds of "pinch-point" discrete self-similar fractals do not strictly self-assemble at any temperature. Unlike our first main theorem, the proof of our second main theorem exploits the underlying geometry of certain kinds of shapes in order to prove that they do not strictly self-assemble. This is a necessary feature of the proof because self-assembly at temperature 2 (or greater) can perform Turing universal comptuation. It remains to be seen whether or not our second main theorem can be extended to *any* non-trivial discrete self-similar fractal.

In our third (and final) main result, we exhibited a construction based on simple modified base-$c$ counters that overcomes fundamental limitations of the TAM with respect to the self-assembly of discrete self-similar fractal structures. Our construction takes as input a simple description of a discrete self-similar fractal and produces a (possibly very large) tile assembly system in which an "approximation" (i.e., fibered version) of the input fractal strictly self-assembles. Other "shape approximation" techniques have been explored in the TAM by Kao and Schweller in [29], and Soloveichik and Winfree in [48]. Note that both of the previous results are of the flavor, "if a lot of (possibly infinitely many) tile types are necessary for the self-assembly of some particular shape, then very few tiles types suffice for the self-assembly of another shape that closely resembles the original shape." Our fiber construction applies to



(a) 'H' fractal generator

(b) Fibered 'H' fractal

a class of (nice) discrete self-similar fractals. It not only preserves discrete fractal dimension but also produces fractal-like shapes that are visually similar to the input fractals. A natural extension of our third main theorem would be to show that *every* (connected) discrete self-similar fractal has a fibered version that strictly self-assembles. In other words, if $X \subsetneq \mathbb{N}^2$ is a connected discrete self-similar fractal, then is it always the case that $X$ has a "fibered" version $\mathcal{F}(X)$ that strictly self-assembles? Of course, $\mathcal{F}(X)$ must also be similar to $X$ in some reasonable sense.

For example, consider the 'H' fractal, whose generator is shown in Figure 5.19a. In Figure 5.19b, we sketch a naive application of our fiber construction to the H fractal. The result is some shape that is clearly not similar to the actual H fractal in any reasonable sense.

The results in this chapter continue to expand our knowledge about the boundaries of self-assembly in the Tile Assembly Model. We also present several open problems which, if answered, will significantly extend them. Such continued research should both benefit theoretical understanding of the model, as well as guiding the direction of laboratory based work.

# CHAPTER 6. Simulation of Self-Assembly in the Abstract Tile Assembly Model with ISU TAS

The work in this chapter has been published as [38].

## 6.1 Introduction

Assembled structures, or assemblies, in the aTAM start from predefined 'seed' structures and can grow to produce finite or infinite structures which represent predefined shapes or represent the outputs of computations. This model has proven to be remarkably powerful and expressive in terms of the types of assemblies which can be produced and the computations which can be performed, and much research has been done to explore this potential.

As this research into the aTAM has progressed, the tile sets being considered and the assemblies studied have steadily increased in size and complexity. Additionally, various alternatives to the original aTAM have been proposed and investigated. This growth has resulted in the need for powerful software tools that can aid in the development and visualization of such sophisticated systems, as well as standardizing results for valid discussion and comparison across diverse approaches. For just such reasons, the Iowa State University Tile Assembly Simulator, or ISU TAS, was developed.

ISU TAS is a freely available, open source, cross-platform software package which provides a full development and simulation environment for the abstract Tile Assembly Model. It provides the ability to create and edit both 2-D and 3-D tile assembly systems in a graphical framework, and then simulate the growth of the ensuing assemblies. Various parameters of the particular version of the aTAM being used can be specified, and several features are provided to help in debugging tile assembly systems.

In this chapter, we first briefly discuss a prior simulator, Xgrow, and its abilities and limitations. We then give a more detailed breakdown of ISU TAS and its current features and functionality, with sections focusing on both the tile set editor and the simulator. Next, we briefly mention some tools we also make available for algorithmically generating tile assembly systems which can be simulated by ISU TAS. Finally, we mention future direction and additional features and functionality that we hope to implement in ISU TAS.

## 6.2 Previous Work

The Xgrow Simulator [15] is a graphical simulator for both the aTAM and the kinetic Tile Assembly Model (kTAM). It was written by the DNA and Natural Algorithms Group, headed by Erik Winfree, at the California Institute of Technology. It is available for download, along with source code, at the following URL: `http://www.dna.caltech.edu/Xgrow`.

Xgrow is written in C for X Windows environment and supports a wide range of options for controlling various parameters of the tile assembly systems it simulates. It also allows for modification of the environment and the assembly dynamically, while an assembly is growing. This functionality allows researchers to better understand the interplay of the many factors that influence assembly in the aTAM and kTAM.

However, while Xgrow is very useful for gaining high-level insights into how tile-based self-assembly works, it doesn't provide the ability to inspect assemblies at the level of individual tiles, or provide tools for editing or debugging tile assembly systems. This makes it difficult to design complex new tile assembly systems, and was the main motivation for the creation of ISU TAS.

## 6.3 ISU TAS Overview

The Iowa State University Tile Assembly Simulator is an integrated platform for designing, simulating, testing, and debugging tile assembly systems in the abstract Tile Assembly Model.

The software and source code are available for download from `http://www.cs.iastate.edu/~lnsa`.

ISU TAS is broken into two main components, the simulator and the tile set editor, which will each be covered in detail in proceeding sections. In this section we provide an overview of the underlying architecture as well as describing a subset of its features.

### 6.3.1 Code Base

ISU TAS is written in C++, and the source code is open source and freely available. It is built upon the wxWidgets toolkit [2], which is a set of cross-platform C++ libraries, and can be built and run on both Windows and Linux operating systems. Build scripts are included for both platforms, and a compiled Windows executable is available for download.

Additional third party libraries are also utilized to provide several enhanced features. To provide 3-D rendering, ISU TAS makes use of the OpenGL [1] and Texfont [33] libraries. RandomLib [30] is used for random number generation in order to provide a uniform distribution.

### 6.3.2 Architecture

In the aTAM, tile assembly systems are defined by three parameters: a tile set, a seed assembly, and a temperature value. To facilitate the building of tile assembly systems, ISU TAS provides tools to graphically build and edit tile sets, design seed assemblies, and set the temperature parameter. This functionality is split between two largely disjoint components, the simulator and the tile set editor. The simulator maintains the definition of a full tile assembly system, while the editor maintains the definition of a separate copy of the tile set, allowing it to be edited without invalidating any assembly currently contained within the simulator. Therefore, whenever an edited version of the tile set is to be used within a simulation, the existing assembly in the simulator is reset to the seed configuration and the tile set from the editor is copied over the tile set maintained by the simulator.

The simulator and tile set editor each have their own top level window. Each of those have dockable sub-windows which can be toggled on and off to provide additional information.

Tile assemblies and tile sets can be saved to and loaded from files. The format is a very simple text file format which also allows for modification within standard text editors or easy programmatic generation.

### 6.3.3 Supported Variations of the aTAM

ISU TAS supports several variations of the aTAM (including a few which differ from the definition in Chapter 2). Some of the configurable parameters for tile assembly systems include the following:

1. The temperature value can set to any desired positive integer.
2. At each time step of the simulation, either one frontier location can be selected at random into which a fitting tile type is placed, or every frontier which exists at the beginning of that time step can be filled in a single time step.
3. Tiles and assemblies can be either 2-dimensional or 3-dimensional.
4. Values can be specified for the relative concentrations of tile types so that, given frontier locations where multiple tile types can fit, a particular tile type is selected with probability proportional to its concentration value relative to all other fitting tile types.

## 6.4 The Tile Set Editor

In the tile set editor window, a new tile set can be created or an existing one loaded. Figure 6.1 shows a screenshot of the tile set editor window. Each tile type is graphically depicted in the ‘Tiletype editor’ window, where they can be cut, copied, pasted, and dragged into different orderings, singly or in selected groups. If a particular tile type is selected by left-clicking on it, it is loaded into the ‘Tile Type Definition’ window, where every attribute of it can be edited.

Additional features of the editor include the ability to:

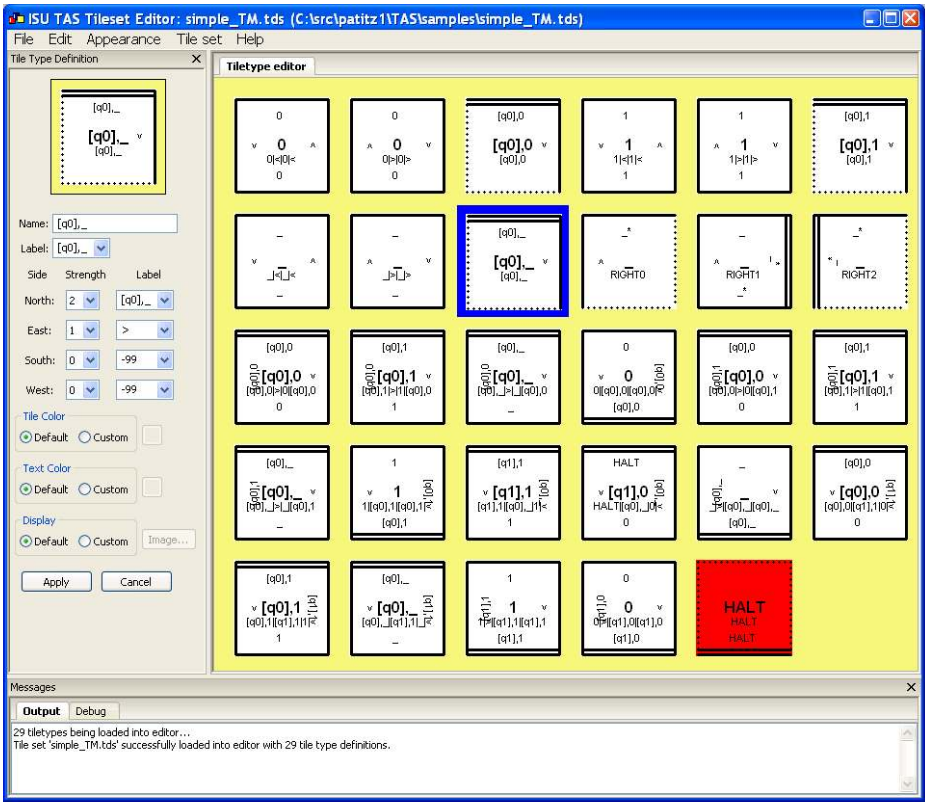
1. Rotate tile types

2. Search for tile types with attributes matching user-specified strings

3. Search for tile types which can bind to a particular side of a selected tile type

4. Highlight all tiles which are being used (or unused) in the current assembly contained in simulator

Additionally the editor automatically highlights any tile types which are functionally equivalent (they have all of the same glues).



Name: [q0],_
Label: [q0],_

| Side | Strength | Label |
|---|---|---|
| North: | 2 | [q0],_ |
| East: | 1 | > |
| South: | 0 | -99 |
| West: | 0 | -99 |

Tile Color: ● Default ○ Custom
Text Color: ● Default ○ Custom
Display: ● Default ○ Custom

```
29 tiletypes being loaded into editor...
Tile set 'simple_TM.tds' successfully loaded into editor with 29 tile type definitions.
```

Figure 6.1: Tile set editor window

All modifications to the tile set in the editor are independent of the tile set loaded into the simulator until the editor's tile set is manually copied into the simulator, which can be done via the ‘Tile set’ menu or a toolbar button in the simulator window.

## 6.5 The Simulator

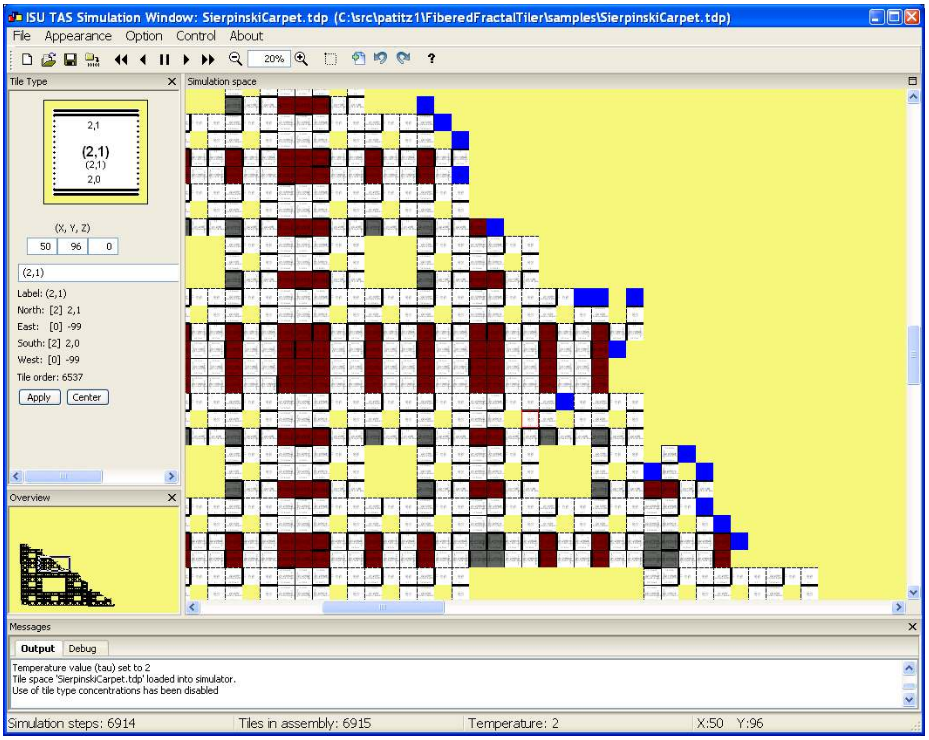
The simulation window of ISU TAS allows a user to create, load, and save tile assembly systems, either as a unit or as separate components. Simulation can be done one step at a time or in a fast-forward mode. Simulation steps are cached, so they can also be run in reverse. The simulation engine is optimized to maximize the speed of assembly while handling very large tile sets (testing has been done with tile sets containing over $10,000$ unique tile types). To provide for maximum simulation speed, the simulator can be configured to redraw the display of the assembly only at user-specified intervals.

Seed assemblies can be created by moving the mouse cursor over the desired coordinates for a seed assembly tile, then right-clicking and selecting the appropriate tile type from the menu which appears.

### 6.5.1 Viewing the Assembly

Figure 6.2 shows the simulation window during the simulation of a two-dimensional tile assembly. The ‘Simulation space’ window shows a portion of the current assembly and allows for arbitrary zoom factors and panning across the entire assembly for high level viewing of the assembly or for tile by tile inspection. It shows the tiles in the assembly as well as highlighting frontier locations with blue squares. The ‘Overview’ window (seen near the bottom left of Figure 6.2) shows a small version of the entire assembly with a box drawn around the area that is currently viewable in the ‘Simulation space’ window. Clicking within the ‘Overview’ window causes the ‘Simulation space’ window to automatically scroll so that the clicked location of the assembly is centered.

When the mouse cursor is moved over the assembly, the ‘Tile type’ window shows the contents of the location currently under the cursor, allowing tile types to be clearly seen without requiring large zoom factors. Besides the attributes of the tile type, the time step in



(X, Y, Z)
50 96 0

(2,1)

Label: (2,1)
North: [2] 2,1
East: [0] -99
South: [2] 2,0
West: [0] -99
Tile order: 6537

```
Temperature value (tau) set to 2
Tile space 'SierpinskiCarpet.tdp' loaded into simulator.
Use of tile type concentrations has been disabled
```

Simulation steps: 6914 | Tiles in assembly: 6915 | Temperature: 2 | X:50 Y:96

Figure 6.2: Simulation window

which a tile was added to that particular location is also displayed, along with its coordinates.

### 6.5.2 Debugging Features

Due to the immense complexity of many tile assembly systems being researched, there is a great need for extensive debugging features that can be used during their development. Some of those provided by ISU TAS are listed below.

1. Breakpoints can be set to stop fast-forward simulation based on any of the three user-specified criteria:
   (a) A specified number of simulation steps have occurred
   (b) A tile is placed in a specified location
   (c) A tile of a specified type is placed in any location
2. The seed used for random number generation can be retrieved, manually set, or automatically generated, and every time an assembly is reset it is possible to specify what seed is used in order to provide reproducible results when debugging issues that arise due to the nondeterminism inherent in the TAM.
3. Locations which have incident glue strengths equal to or greater than $\tau$ are drawn as blue squares since they are eligible frontier locations, and any such locations at which a tile addition has been attempted but no possible fitting tile type was found are drawn as red squares and referred to as ‘dead’ frontier locations.
4. A box-drawing tool can be used which allows frontier locations to be selected and toggled ‘on’ or ‘off’ in order to restrict assembly growth to particular locations.
5. An option can be turned on which causes the simulator to report every instance in which a tile was added to a location in which more than one tile type could have validly been placed (a type of non-determinism).

6. An option can be turned on which causes the simulator to report every instance in which a tile was added to a location in which it bound with strength greater than $\tau$, which is a violation of the first condition of local determinism (Soloveichik and Winfree [48]).

### 6.5.3 3-D Simulation

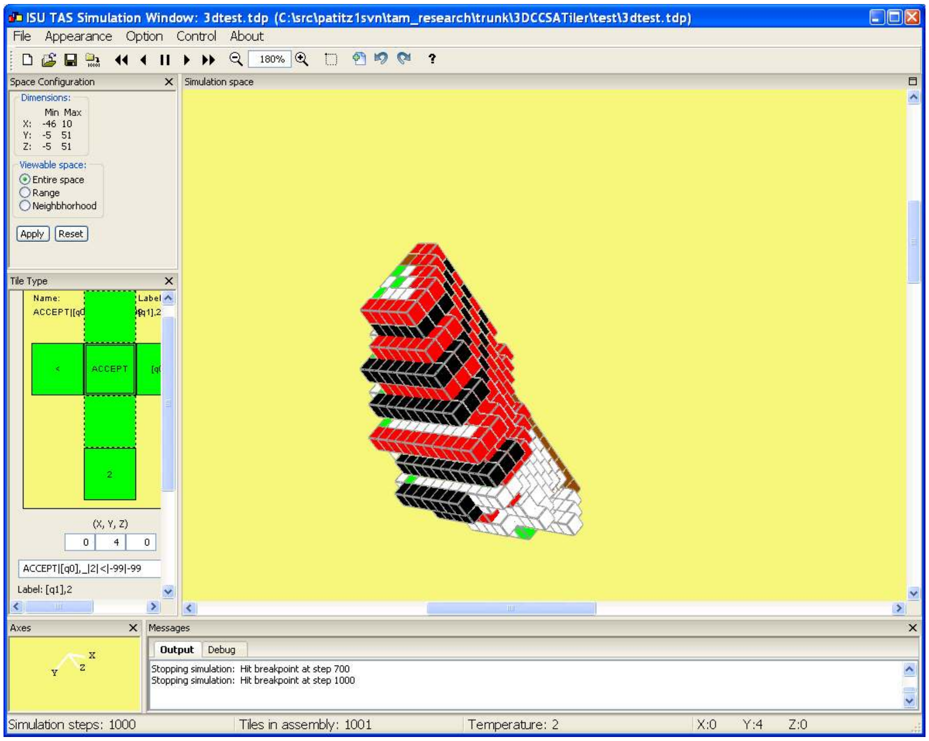
Figure 6.3 shows the simulation window when the simulator is in 3-D mode. The differences in 3-D simulation from 2-D mostly concern the visualization. In 3-D mode, the mouse is used to rotate the assembly and view it from different angles. One additional window, the 'Axes' window, displays the current orientation of the three positive axes, while another, the 'Space Configuration' window, allows user-defined regions (or 'slices') of the assembly to be the only visible portions. This allows for inspection of arbitrary pieces of the assembly, even if they are interior and would otherwise be blocked from view by other portions of the assembly. Frontier locations are displayed as semi-transparent cubes. Finally, the 'Tile Type' window now shows an 'unwrapped' three-dimensional tile so that all sides can be simultaneously viewed.

## 6.6 Additional Tools

In addition to ISU TAS, we also make several programs for algorithmically generating tile assembly systems used by ISU TAS, and their source code, available. They are written in standard C++ with no requirements on third party libraries and include a basic library, TileLib, which can be used to easily create new applications. Tools for generating tile sets which act as counters, Turing machine simulations, and other constructions related to several of our results can all be downloaded.

## 6.7 Future Work

While ISU TAS has come a long way toward becoming a solid and robust environment for designing and testing tile assembly systems, there remain many features on our list to implement in the future.



ISU TAS Simulation Window: 3dtest.tdp (C:\src\patitz1svn\tam_research\trunk\3DCCSATiler\test\3dtest.tdp)

180%

Dimensions:

|    | Min | Max |
|----|-----|-----|
| X: | -46 | 10 |
| Y: | -5 | 51 |
| Z: | -5 | 51 |

Viewable space:
● Entire space
○ Range
○ Neighbhorhood

(X, Y, Z)
0 4 0

ACCEPT|[q0],_|2|<|-99|-99

Label: [q1],2

```
Stopping simulation:  Hit breakpoint at step 700
Stopping simulation:  Hit breakpoint at step 1000
```

Simulation steps: 1000 | Tiles in assembly: 1001 | Temperature: 2 | X:0 Y:4 Z:0

Figure 6.3: Simulation window (3-D mode)

Many more optimizations for 3-D simulation are required to support large assemblies. Partial support for temperature programming has been implemented, but an efficient way to calculate the fragments of assemblies which should 'melt' off at temperature increases is needed. Building and testing of ISU TAS needs to be performed on Mac OSX in order to add that as a supported platform. We also wish to add support for versions of the kTAM.

With increased interest in, and usage of, ISU TAS we hope to receive useful feedback and testing that can enable us continue to provide tools that help further research in tile-based self-assembly and aid in moving this theoretical research closer to a physical reality.

# CHAPTER 7. A Domain-Specific Language for Programming in the Tile Assembly Model

The work in this chapter was performed with co-author David Doty. It has been published as [19].

## 7.1 Introduction

### 7.1.1 Background

The current practice of the design of tile types for the abstract Tile Assembly Model and related models is not unlike early machine-level programming. Numerous theoretical papers [28, 48, 45, 47] focus on the *tile complexity* of systems, the minimum number of tile types required to assemble certain structures such as squares. A major motivation of such complexity measures is the immense level of laboratory effort presently required to create physical implementations of tiles.

Early electronic computers occupied entire rooms to obtain a fraction of the memory and processing power of today's cheapest mobile telephones. This limitation did not stop algorithm developers from creating algorithms, such as divide-and-conquer sorting and the simplex method, that find their most useful niche when executed on data sets that would not have fit on all the memory existing in the world in 1950. Similarly, we hope and expect that the basic components of self-assembly will one day, through the ingenuity of physical scientists, become cheap and easy to produce, not only in quantity but in variety. The *computational* scientists will then be charged with developing disciplined methods of organizing such components so that systems of high complexity can be designed without overwhelming the engineers producing the design. In this chapter, we introduce a preliminary attempt at such a disciplined method of controlling the complexity of designing tile assembly systems in the aTAM.

Simulated tile assembly systems of moderate complexity cannot be produced by hand; one must write a computer program that handles the drudgery of looping over related groups of individual tile types. However, even writing a program to produce tile types directly is error-prone, and more low-level than the ways that we tend to think about tile assembly systems.

Fowler [23] suggests that a *domain-specific language (DSL)* is an appropriate tool to introduce into a programming project when the syntax or expressive capabilities of a general-purpose programming language are awkward or inadequate for certain portions of the project. He distinguishes between an *external DSL*, which is a completely new language designed especially for some task (such as SQL for querying or updating databases), and an *internal DSL*, which is a way of "hijacking" the syntax of an existing general-purpose language to express concepts specific to the domain (e.g. Ruby on Rails for writing web applications). An external DSL may be as simple as an XML configuration file, and an internal DSL may be as simple as a class library. In either case the major objective is to express "commands" in the DSL that more closely model the way one thinks about the domain than the "host" language in which the project is written.

If the syntax and semantics of the DSL are precisely defined, this facilitates the creation of a *semantic editor* (text-based or visual), in which programs in the DSL may be produced using a tool that can visually show semantically meaningful information such as compilation or even logical errors, and can help the programmer directly edit the abstract components of the DSL, instead of editing the source code directly. For example, Intellij IDEA and Eclipse are two programs that help Java programmers to do refactorings such as variable renaming or method inlining, which are at their core operations that act directly on the abstract syntax tree of the Java program rather than on the text constituting the source code of the program. We have kept such a tool in mind as a guide for how to appropriately structure the DSL for designing tile systems.

We structure this chapter in such a way as to de-emphasize any dependence of the DSL on Python in particular or even on object-oriented class libraries in general. The DSL provides a high-level way of *thinking* about tile assembly programming, which, like any high-level language or other advance in software engineering, not only automates mundane tasks better left to computers, but also *restricts* the programmer from certain error-prone tasks, in order to better guide the design, following the dictum of Antoine de Saint-Exupery that a design is perfected "not when there is nothing left to add, but nothing left to take away."

### 7.1.2 Brief Outline of the DSL

We now briefly outline the design of the DSL. Section 7.2 provides more detail.

The most fundamental component of designing a tile assembly system manually is the tile type. In our DSL, the most fundamental component is an object known as a *tile template.* A tile template represents a group of tile types (each tile type being an *instance* of the tile template), sharing the same input sides, output sides, and function that transforms input signals into output signals. The two fundamental operations of the DSL are *join* and *add transition.* Both make the assumption that each tile template has well-defined input and output sides. In a join, an input tile template $A$ is connected to an output tile template $B$ in a certain direction $d \in \{N, S, E, W\}$, expressing that an instance $t_A$ of $A$ may have on its output side in direction $d$ an instance $t_B$ of $B$. This expresses an intention that in the growth of the assembly, $t_A$ will be placed first, then $t_B$, and they will bind with positive strength, with $t_A$ passing information to $t_B$. Whereas a join is an operation connecting the output side of a tile template to the input side of another, a transition "connects" input sides to output sides within a single tile template, by specifying how to compute information on the output side as a function of information on the input sides. This is called *adding* a transition rather than *setting*, since the information on the output sides may contain multiple independent output signals, and their computations may be specified independently of one another if convenient.

This notion of independent signals is modeled already in other DSLs for the aTAM [11] and for similar systems such as cellular automata [14]. The join operation, however, makes sense in the aTAM but not in a system such as a cellular automaton, where each cell contains the same transition function(s). The notion of tile templates allows one to break an “algorithm” for assembly into separate “stages”, each stage corresponding to a different tile template, with each stage being modularized and decoupled from other stages, except through well-defined and restricted signals passed through a join. There is a rough analogy with lines of code in a program: a single line of code may execute more than once, each time executed with the state of memory being different than the previous. Similarly, many different tile types, with different actual signal values, generated from the same tile template, may be placed during the growth of an assembly. Another difference between our language and that of [11] is that our language appears to be more general; rather than being geared specifically toward the creation of geometric shapes, our language is more low-level, but also more general.[1] Additionally, [49] presents a DSL which provides a higher level framework for the modeling of various self-assembling systems, while the focus of our DSL on the aTAM creates a more powerful platform for the development of complex tile assembly systems.

This chapter is organized as follows. Section 7.2 describes the DSL for tile assembly programming in more detail and gives examples of design and use. Section 7.3 concludes the chapter and discusses future work and open theoretical questions. A preliminary implementation of the DSL and the visual editor can be found at `http://www.cs.iastate.edu/~lnsa`.

## 7.2 Description of Language

The DSL is written as a class library in the Python programming language. It is designed as a set of classes which encapsulate the logical components needed to design a tile assembly system in the aTAM where the temperature value $\tau = 2$.

The DSL is designed around the principal notion that data moves through an assembly as ‘signals’ which pass through connected tiles as the information encoded in the input glues,

[1] An extremely crude analogy would be that the progression *manual tile programming* → *Doty-Patitz* → *Becker* is roughly analogous to *machine code* → *C* → *Logo*.

allowing a particular tile type to bind in a location, and then, based on the 'computation' performed by that tile type, as the resultant information encoded in the glues of its output edges. (Of course, tiles in the aTAM are static objects so the computation performed by a given tile type is a simple mapping of one set of input glues to one set of output glues which is hardcoded at the time the tile type is created.) Viewed in this way, signals can be seen to propagate as tiles attach and an assembly forms, and it is these signals which dictate the final shape of the assembly. Using this understanding of tile-based self-assembly, we designed the DSL from the standpoint of building tile assembly systems around the transmission and processing of such signals.

We present a detailed overview of the objects and operations provided by the DSL, then demonstrate an example exhibiting the way in which the DSL is used to design a tile set.

### 7.2.1 Client-side description

The DSL provides a collection of objects which represent the basic design units and a set of operations which can be performed on them. We describe these objects and operations in this section, without describing the underlying data structures and algorithms implementing them.

The DSL strictly enforces the notion of input and output sides for tile types, meaning that any given side can be designated as receiving signals (an input side), sending signals (an output side), or neither (a blank side).

#### 7.2.1.1 DSL objects

**Tile system** The highest level object is the *tile system* object, which represents the full specification of a temperature 2 tile assembly system. It contains child objects representing the tile set and seed specification, and provides the functionality to write them out to files in a format readable by the ISU TAS simulator [38] (`http://www.cs.iastate.edu/~lnsa/software.html`).

**Tile** In some cases, especially for tiles contained within seed assemblies, there is no computation being performed. In such situations, it may be easier for the programmer to fully specify the glues and properties of a tile type. The *tile* object can be used to easily create such hardcoded tile types.

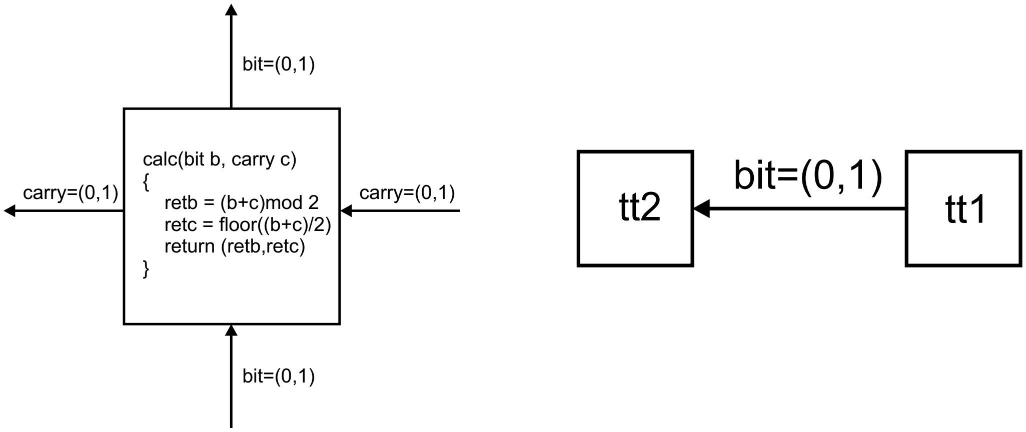
**Tile template** The principle design unit in the DSL is the tile template. This object represents a collection of tile types which share the following properties:

- They have exactly the same input, output, and blank sides.
- The types of signals received/transmitted on every corresponding input/output side are identical.
- The computation performed to transform the input signals to output signals can be performed by the same function, using the values specific to each instantiated tile type.

Logically, a tile template represents the set of tile types which perform the same computation on different values of the same input signal types.

**Tile set template** A *tile set template* contains all of the information needed to generate a tile set. It contains the sets of tile and tile template objects included in the tile set, as well as the logic for performing joins, doing error checking, etc. The tile set template object encapsulates information and operations that require communication between more than one tile template object, such as the *join* operation, whereas a tile template is responsible for operations that require information entirely local to the tile template, such as *add transition*.

**Signal** A *signal* is simply the name of a variable and the set of allowable values for it. For example, a signal used to represent a binary digit could be defined by giving it the name *bit* and the allowable values 0 and 1.



(a) Transition: The function *calc* takes bit/carry values as input from the bottom/right, and computes new bit/carry values which are then output on the top/left.

(b) Join: The tile template on the right (tt1) is sending the signal *bit* with allowable values $(0, 1)$ to the tile template on the left (tt2).

Figure 7.1: Logical representations of some DSL objects

**Transition** *Transitions* are the objects which provide the computational abilities of tile templates. A transition is defined as a set of input signal names, output signal names, and a function which operates on the input signals to yield output signals. The logic of a function can be specified as a table enumerating all input signals and their corresponding outputs, a Python expression, or a Python function, yielding the full power of a general purpose programming language for performing the computations. An example is shown in Figure 7.1a.

#### 7.2.1.2 DSL operations

**Join** The primary operation between tile templates, which defines the signals that are passed and the paths that they take through an assembly, is called a *join.* Joins are performed between the complementary sides (north and south, or east and west) of two tile templates (which need not be unequal), or a tile and a tile template. If one parameter is a tile, then all signals must be given just one value; otherwise a set of possible values that could be passed between the tile templates is specified (which can still be just one). A join is directional, specifying the direction in which a signal (or set of signals) moves. This direction defines the input and output sides of the tile templates which a join connects. An example is shown in

Figure 7.1b.

**Add transition** Transition objects can be added to tile template objects, and indicate how to compute the output signals of a tile template from the input signals. If convenient, a single transition can compute more than one output signal by returning a tuple. Each output signal of a tile template with more than one possible value must have a transition computing it.

**Add chooser** There may be multiple tile templates that share the same input signal values on all of their input sides. Depending on the join structure, the library may be able to use annotations (see below) to avoid “collisions” resulting from this, but if the tile templates share joins, then it may not be possible (or desirable) for the library to automatically calculate which tile template should be used to create a tile matching the inputs. In this case, a user-defined function called a *chooser* must be added so that the DSL can ensure that only a single tile type is generated for each combination of input values. This helps to avoid accidental nondeterminism.

**Set property** Additional properties such as the display string, tile color, and tile type concentrations can be set using user-defined functions for each property.

### 7.2.2 Additional features

The DSL provides a number of additional, useful features to programmers, a few of which will be described in this section. First, the DSL performs an analysis of the connection structure formed between tile templates by joins and creates *annotations*, or additional information in the form of strings, which are appended to glue labels. These annotations ensure that only tile types created from tile templates which are connected by joins can bind with each other in the directions specified by those joins, preventing the common error of accidentally allowing two tiles to bind that should not because they happen to communicate the same information in the same direction as two unrelated tiles.

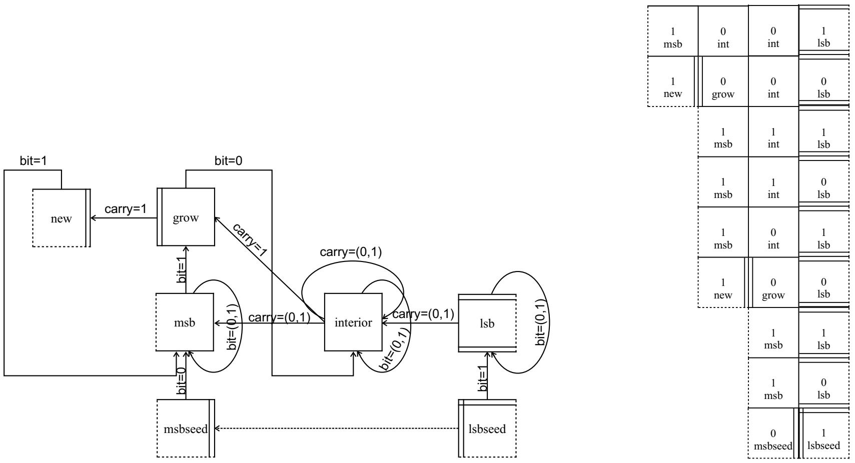
Although some forms of ‘accidental’ nondeterminism are prevented by the DSL, it does provide methods by which a programmer can intentionally create nondeterminism. Specifically, a programmer can either design a chooser function which returns more than one output tile type for a given set of inputs, or one can add *auxiliary inputs* to a tile template, which are input signals that do not come from any direction.

The DSL provides additional error-checking functionality. For example, each tile template must have either one strength-2 input or two strength-1 inputs. As another example, the programmer is forced to specify a chooser function if the DSL cannot automatically determine a unique output tile template, which is a common source of accidental nondeterminism in tile set design.

### 7.2.3 Example construction

In this section, we present an example of how to use the DSL to produce a tile assembly system which assembles a log-width binary counter. In order to slightly simplify this example, the seed row of the assembly, representing the value 1, will be two tiles wide instead of one. All other rows will be of the correct width, i.e. a row representing the value $n$ will be $\lceil \log_2(n+1) \rceil$ tiles wide.

Figure 7.2a shows a schematic diagram representing a set of DSL objects, namely tiles, tile templates and joins, which can generate the necessary tile types. It is similar in appearance to how such a diagram appears in the visual editor. The squares labeled *lsbseed* and *msbseed* represent hard-coded tiles for the seed row. The other squares represent tile templates, with the names shown in the middle. The connecting lines represent joins, which each have a direction specified by a terminal arrow and a signal which is named (either *bit* or *carry*) and has a range of allowable values (either 0, or 1, or both). The dashed line between *lsbseed* and *msbseed* represents an implicit join between these two hard-coded tile types because they were manually assigned the same glues.



(a) Schematic diagram depicting the tile templates and joins which are used to generate a tile set that self-assembles a log-width binary counter. All east-west joins pass the *carry* signal and south-north joins pass the *bit* signal (although some of them are restricted to subsets of the allowable values $(0,1)$). Transitions are not shown explicitly, but, for example, the tile template *interior* would have a transition as shown in Figure 7.1a.

(b) First 9 rows of the assembly of a log-width binary counter. Each tile is labeled with its bit value on top and the name of the tile template from which it was generated on the bottom.

Figure 7.2: Schematic diagram and partial assembly of generated tile types for log-width counter.

In addition to the joins, transition functions must be added to the *lsb*, *interior*, and *msb* tile templates to compute their respective output signals, since each has multiple possible values for those signals (whereas *grow*, for instance, which always outputs carry=1 to its west and bit=0 to its north, so needs no transitions). Finally, since the tile templates *msb* and *grow* have overlapping input signal values of bit=1 from *msb* on the south and carry=1 from *interior* on the east, a chooser function must be supplied to indicate that an instance of *grow* should be created for the signals bit=1 and carry=1, and an instance of *msb* should be created in the other three circumstances. Figure 7.2b shows a partial assembly of tiles generated from the tile template diagram of Figure 7.2a, without glue labels shown explicitly.

## 7.3 Conclusion and Future Work

We have described a domain-specific language (DSL) for creating tile sets in the abstract Tile Assembly Model. This language is currently implemented as a Python class library but is framed as a DSL to emphasize its role as a high-level, disciplined way of thinking about the creation of tile systems.

### 7.3.1 Semantic Visual Editor

The DSL is implemented as a Python class library, but one thinks of the "real" programming as the creation of visual tile templates and the joins between them, as well as the addition of signal transitions. We have implemented a visual editor that removes the Python programming and allows the direct creation of the DSL objects and execution of its operations. It detects errors, such as a tile template having only one input side of strength 1, while temporarily allowing the editing to continue. Other types of errors, that do not aid the user in being allowed to persist, such as the usage of a single side as both an input and output, are prohibited outright. Many improvements in error detection and handling remain to be implemented and support remains to be added for several features of the DSL.

One can therefore create a tile set by directly drawing the tile templates as shown in Figure 7.2a, while receiving helpful tips from a semantically-aware editor about errors.

### 7.3.2 Avoidance of Accidental Nondeterminism

Initially, our hope had been to design a DSL with the property that it could be used in a straightforward way to design a wide range of existing tile assembly systems, but was sufficiently restricted that it could be guaranteed to produce a deterministic TAS, so long as nondeterminism was not directly and intentionally introduced through chooser functions or auxiliary inputs. Unfortunately, it is straightforward to use the DSL to design a TAS that begins the parallel simulation of two copies of a Turing machine so that if (and only if) the Turing machine halts, two "lines" of tiles are sent growing towards each other to compete nondeterministically, whence the detection of such nondeterminism is undecidable. A common cause of “accidental nondeterminism” in the design of tile assembly systems is the use of a single side of a tile type as both input and output; this sort of error is prevented by the nature of the DSL in assigning each tile template unique, unchanging sets of input sides and output sides. However, we cannot see an elegant way to restrict the DSL further to automatically prevent or statically detect the presence of the sort of “geometric nondeterminism” described in the Turing machine example. The development of such a technique would prove a boon to the designers of tile assembly systems. Conversely, consider Blum’s result [12] that any programming language in which all programs are guaranteed to halt requires uncomputably large programs to compute some functions that are trivially computable in a general-purpose language. Perhaps an analogous theorem implies that any restriction statically enforcing determinism also cripples the Tile Assembly Model, so that accidental nondeterminism in tile assembly, like accidental infinite loops in software engineering, would be an unavoidable fact of life, and time is better spent developing formal methods for proving determinism rather than hoping that it could be automatically guaranteed.

### 7.3.3 Further Abstractions

To better handle very large and complex constructions, it would be useful to support further abstractions, such as grouping sets of tile templates and related joins into “modules” which could then be treated as atomic units. These modules should have well-defined interfaces which allow them to be configured and combined to create more complicated, composite tile sets.

### 7.3.4 Other Self-Assembly Models

The abstract Tile Assembly Model is a simple but powerful tool for exploring the theoretical capabilities and limitations of molecular self-assembly. However, it is an (intentionally) overly-simplified model. Generalizations of the aTAM, such as the kinetic Tile Assembly Model [53, 54], and alternative models, such as graph self-assembly [43], have been studied theoretically and implemented practically. We hope to leverage the lessons learned from designing the aTAM DSL to guide the design of more advanced DSLs for high-level programming in these alternative models.

# CHAPTER 8. Conclusion

Researchers in the area of self-assembly began by designing very simplified models, such as the aTAM, which could be easily understood and studied. This made it possible to build a firm foundation for understanding the basic principles of self-assembling systems. As we have gained proficiency and increased our knowledge about such systems, the general direction has been towards developing increasingly complex models which come closer and closer to mimicking the amazing instances of self-assembly exhibited by the natural world.

While the aTAM has proven to be a powerful model in which an immense array of structures can self-assemble and powerful computations can be performed, many research results have shown fundamental limitations to what is achievable within it. Additionally, laboratory experiments have proven that substantial error rates greatly restrict what can self-assemble using current techniques. Because of this, research is now turning to variations of the TAM which are both more powerful as well as more robust with respect to errors, with the eventual goal of creating systems which can be physically implemented and which will be able to self-assemble vastly more complex structures than are possible today. These alternative models are based on new designs of the molecular building blocks which form tiles (such as RNA tiles [3], for example), more physically realistic behaviors allowed by the models (such as the two-handed aTAM [20]), a greater understanding of the most prevalent types of errors that occur today, and a blending of tools from other branches of research into molecular computing which provide the framework for more adaptive components.

In order to allow this field of research to mature, it will also be necessary to greatly extend the direction of work in this thesis, expanding the software tools available for design and simulation of new models and constructions, and furthering the design of a programming language for self-assembly. Such software will serve two main goals. First, it will speed up the rate at which researchers can test the viability of new ideas. Second, it will shrink the amount of time that is needed to teach researchers new to the field, and thus inspire new researchers to enter it. For example, abstracting the particular details of the aTAM from the ISU TAS would allow it to become a more generic self-assembly simulator into which researchers can easily plug their own modules which contain their own model definitions (for instance, adding support for aspects of the kinetic Tile Assembly Model [52] such as errors by insufficient attachment, support for two-handed assembly [20], and handling of behaviors such as those caused by RNAse enzymes [3]). Such work will require a significant amount of work in computational theory and software engineering, as well as collaborations with molecular biologists who can provide guidance into how to more realistically simulate biomolecular systems.

In the longer term, once we have achieved mastery of increasingly powerful self-assembling systems, I expect such research to have provided us with a much greater understanding of basic principles that govern living systems at the lowest levels. This will, in turn, inspire the incorporation of a wider set of nature's tools into our designs. For instance, methods for incorporating evolutionary paradigms into self-assembling systems will advance them to the next generation, allowing researchers to build more and more complex assemblies while further deciphering the origins and intricacies of living systems. Algorithmic self-assembly is a rich area of research which will provide a wealth of fundamentally interesting and practically useful results, transforming diverse areas of science and technology.

# BIBLIOGRAPHY

[1] OpenGL.
http://www.opengl.org (accessed January 2009).

[2] wxWidgets Cross-Platform GUI Library.
http://www.wxwidgets.org (accessed January 2009).

[3] Z. Abel, N. Benbernou, M. Damian, E. Demaine, M. Demaine, R. Flatland, S. Kominers, and R. Schweller. Shape replication through self-assembly and rnase enzymes. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2010)*, 2010. to appear.

[4] L. M. Adleman, Q. Cheng, A. Goel, M.-D. A. Huang, D. Kempe, P. M. de Espanés, and P. W. K. Rothemund. Combinatorial optimization problems in self-assembly. In *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing*, pages 23–32, 2002.

[5] L. M. Adleman, Q. Cheng, A. Goel, M.-D. A. Huang, D. Kempe, P. M. de Espanés, and P. W. K. Rothemund. Combinatorial optimization problems in self-assembly. In *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing*, pages 23–32, 2002.

[6] L. M. Adleman, J. Kari, L. Kari, and D. Reishus. On the decidability of self-assembly of infinite ribbons. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 530–537, 2002.

[7] G. Aggarwal, M. H. Goldwasser, M.-Y. Kao, and R. T. Schweller. Complexities for generalized models of self-assembly. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, 2004.

[8] E. S. Andersen, M. Dong, M. M. Nielsen, K. Jahn, R. Subramani, W. Mamdouh, M. M. Golas, B. Sander, H. Stark, C. L. P. Oliveira, J. S. Pedersen, V. Birkedal, F. Besenbacher, K. V. Gothelf, and J. Kjems. Self-assembly of a nanoscale dna box with a controllable lid. *Nature*, 459(7243):73–76, May 2009.

[9] J. Bachrach and J. Beal. Building spatial computers. Technical report, MIT CSAIL, 2007.

[10] J. Beal and G. Sussman. Biologically-inspired robust spatial programming. Technical report, MIT, 2005.

[11] F. Becker. Pictures worth a thousand tiles, a geometrical programming language for self-assembly. *Theoretical Computer Science*, 410(16):1495–1515, 2009.

[12] M. Blum. On the size of machines. *Information and Control*, 11(3):257–265, 1967.

[13] Q. Cheng, A. Goel, and P. M. de Espanés. Optimal self-assembly of counters at temperature two. In *Proceedings of the First Conference on Foundations of Nanoscience: Self-assembled Architectures and Devices*, 2004.

[14] H.-H. Chou, W. Huang, and J. A. Reggia. The Trend cellular automata programming environment. *SIMULATION: Transactions of The Society for Modeling and Simulation International*, 78:59–75, 2002.

[15] DNA and Natural Algorithms Group. Xgrow simulator.
http://www.dna.caltech.edu/Xgrow (accessed January 2009).

[16] D. Doty, X. Gu, J. H. Lutz, E. Mayordomo, and P. Moser. Zeta-Dimension. In *Proceedings of the Thirtieth International Symposium on Mathematical Foundations of Computer Science*, pages 283–294. Springer-Verlag, 2005.

[17] D. Doty, J. H. Lutz, M. J. Patitz, S. M. Summers, and D. Woods. Intrinsic universality in self-assembly. In *Proceedings of the 27th International Symposium on Theoretical Aspects of Computer Science*, 2009. to appear.

[18] D. Doty, J. H. Lutz, M. J. Patitz, S. M. Summers, and D. Woods. Random number selection in self-assembly. In *Proceedings of The Eighth International Conference on Unconventional Computation (Porta Delgada (Azores), Portugal, September 7-11, 2009)*, 2009.

[19] D. Doty and M. J. Patitz. A domain specific language for programming in the tile assembly model. In *Proceedings of The Fifteenth International Meeting on DNA Computing and Molecular Programming (Fayetteville, Arkansas, USA, June 8-11, 2009)*, volume 5877 of *Lecture Notes in Computer Science*, pages 25–34, 2009.

[20] D. Doty, M. J. Patitz, D. Reishus, R. T. Schweller, and S. M. Summers. Strong fault-tolerance for self-assembly with fuzzy temperature. Technical report, 2009.

[21] D. Doty, M. J. Patitz, and S. M. Summers. Limitations of self-assembly at temperature 1. *Theoretical Computer Science*. to appear.

[22] D. Doty, M. J. Patitz, and S. M. Summers. Limitations of self-assembly at temperature 1. In *Proceedings of The Fifteenth International Meeting on DNA Computing and Molecular Programming (Fayetteville, Arkansas, USA, June 8-11, 2009)*, volume 5877 of *Lecture Notes in Computer Science*, pages 35–44, 2009.

[23] M. Fowler. Language workbenches: The killer-app for domain specific languages?, June 2005. `http://martinfowler.com/articles/languageWorkbench.html`.

[24] Y. Fu and R. Schweller. Temperature 1 self-assembly: Deterministic assembly in 3d and probabilistic assembly in 2d. Technical Report 0912.0027, Computing Research Repository, 2009.

[25] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 1994.

[26] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[27] S. Irani, M. Naor, and R. Rubinfeld. On the time and space complexity of computation using write-once memories, or Is pen really much worse than pencil? *Theory of Computing Systems*, 25:141–159, 1992.

[28] M.-Y. Kao and R. T. Schweller. Reducing tile complexity for self-assembly through temperature programming. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006), Miami, Florida, Jan. 2006, pp. 571-580*, 2007.

[29] M.-Y. Kao and R. T. Schweller. Randomized self-assembly for approximate shapes. In L. Aceto, I. Damgrd, L. A. Goldberg, M. M. Halldrsson, A. Inglfsdttir, and I. Walukiewicz, editors, *International Colloqium on Automata, Languages, and Programming (ICALP)*, volume 5125 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2008.

[30] C. Karney. RandomLib.
http://charles.karney.info/random (accessed January 2009).

[31] S. M. Kautz and J. I. Lathrop. Self-assembly of the Sierpinski carpet and related fractals. In *Proceedings of The Fifteenth International Meeting on DNA Computing and Molecular Programming (Fayetteville, Arkansas, USA, June 8-11, 2009)*, volume 5877 of *Lecture Notes in Computer Science*, pages 78–87, 2009.

[32] R. J. Kershner, L. D. Bozano, C. M. Micheel, A. M. Hung, A. R. Fornof, J. N. Cha, C. T. Rettner, M. Bersani, J. Frommer, P. W. K. Rothemund, and G. M. Wallraff. Placement and orientation of individual dna shapes on lithographically patterned surfaces. *Nature Nanotechnology*, 4(9):557–561, August 2009.

[33] M. Kilgard. Texfont. http://www.opengl.org/resources/code/samples/glut_examples/texfont/texfont.html (accessed January 2009).

[34] J. I. Lathrop, J. H. Lutz, M. J. Patitz, and S. M. Summers. Computability and complexity in self-assembly. *Theory of Computing Systems*. to appear.

[35] J. I. Lathrop, J. H. Lutz, M. J. Patitz, and S. M. Summers. Computability and complexity in self-assembly. In *Proceedings of The Fourth Conference on Computability in Europe (Athens, Greece, June 15-20, 2008)*, pages 349–358, 2008.

[36] J. I. Lathrop, J. H. Lutz, and S. M. Summers. Strict self-assembly of discrete Sierpinski triangles. *Theoretical Computer Science*, 410:384–405, 2009.

[37] U. Majumder, T. H. LaBean, and J. H. Reif. Activatable tiles for compact error-resilient directional assembly. In *13th International Meeting on DNA Computing (DNA 13), Memphis, Tennessee, June 4-8, 2007.*, 2007.

[38] M. J. Patitz. Simulation of self-assembly in the abstract tile assembly model with ISU TAS. In *6th Annual Conference on Foundations of Nanoscience: Self-Assembled Architectures and Devices (Snowbird, Utah, USA, April 20-24 2009).*, 2009.

[39] M. J. Patitz and S. M. Summers. Self-assembly of decidable sets. *Natural Computing*. to appear.

[40] M. J. Patitz and S. M. Summers. Self-assembly of discrete self-similar fractals. *Natural Computing*. to appear.

[41] M. J. Patitz and S. M. Summers. Self-assembly of decidable sets. In *Proceedings of The Seventh International Conference on Unconventional Computation (Vienna, Austria, August 25-28, 2008)*, volume 5204 of *Lecture Notes in Computer Science*, pages 206–219. Springer-Verlag, 2008.

[42] M. J. Patitz and S. M. Summers. Self-assembly of discrete self-similar fractals (extended abstract). In *Proceedings of The Fourteenth International Meeting on DNA Computing (Prague, Czech Republic, June 2-6, 2008)*, volume 5347 of *Lecture Notes in Computer Science*, pages 156–167. Springer-Verlag, 2009.

[43] J. H. Reif, S. Sahu, and P. Yin. Complexity of graph self-assembly in accretive systems and self-destructible systems. In *DNA*, pages 257–274, 2005.

[44] P. W. K. Rothemund. *Theory and Experiments in Algorithmic Self-Assembly.* PhD dissertation, University of Southern California, December 2001.

[45] P. W. K. Rothemund and E. Winfree. The program-size complexity of self-assembled squares (extended abstract). In *STOC '00: Proceedings of the thirty-second annual ACM Symposium on Theory of Computing*, pages 459–468, New York, NY, USA, 2000. ACM.

[46] N. C. Seeman. Nucleic-acid junctions and lattices. *Journal of Theoretical Biology*, 99:237–247, 1982.

[47] D. Soloveichik and E. Winfree. Complexity of compact proofreading for self-assembled patterns. In *The eleventh International Meeting on DNA Computing*, 2005.

[48] D. Soloveichik and E. Winfree. Complexity of self-assembled shapes. *SIAM Journal on Computing*, 36(6):1544–1569, 2007.

[49] A. Spicher, O. Michel, and J.-L. Giavitto. Algorithmic self-assembly by accretion and by carving in MGS. In *Artificial Evolution*, volume 3871 of *Lecture Notes in Computer Science*, pages 189–200. Springer, 2005.

[50] H. Wang. Proving theorems by pattern recognition – II. *The Bell System Technical Journal*, XL(1):1–41, 1961.

[51] H. Wang. Dominoes and the AEA case of the decision problem. In *Proceedings of the Symposium on Mathematical Theory of Automata (New York, 1962)*, pages 23–55. Polytechnic Press of Polytechnic Inst. of Brooklyn, Brooklyn, N.Y., 1963.

[52] E. Winfree. *Algorithmic Self-Assembly of DNA*. PhD dissertation, California Institute of Technology, June 1998.

[53] E. Winfree. Simulations of computing by self-assembly. Technical Report CaltechCSTR:1998.22, California Institute of Technology, 1998.

[54] E. Winfree and R. Bekbolatov. Proofreading tile sets: Error correction for algorithmic self-assembly. In J. Chen and J. H. Reif, editors, *DNA*, volume 2943 of *Lecture Notes in Computer Science*, pages 126–144. Springer, 2003.