Fractals in Seeded Tile Automata

Asher Haun
University of Texas Rio Grande Valley, Edinburg, TX, USA

Ryan Knobel
University of Texas Rio Grande Valley, Edinburg, TX, USA

Adrian Salinas
University of Texas Rio Grande Valley, Edinburg, TX, USA

Ramiro Santos
University of Texas Rio Grande Valley, Edinburg, TX, USA

Robert Schweller
University of Texas Rio Grande Valley, Edinburg, TX, USA

Tim Wylie
University of Texas Rio Grande Valley, Edinburg, TX, USA

Abstract

This work fully characterizes fractal generation in the seeded Tile Automata model (seeded TA), a model similar to the abstract Tile Assembly model (aTAM) with the added ability for adjacent tiles to change states. Under these assumptions, we first show that all discrete self-similar fractals (DSSFs) with feasible generators are strictly buildable at scale 1 and temperature 1 in seeded TA. We then show that these results imply the existence of a *single* seeded TA system Γ that can strictly build any DSSF infinitely at scale 1 and temperature 1.

2012 ACM Subject Classification Theory of computation

Keywords and phrases self-assembly, tile automata, fractals

Digital Object Identifier 10.4230/LIPIcs.SAND.2025.14

Funding This research was supported in part by National Science Foundation Grant CCF-2329918.

1 Introduction

The past 20 years within the field of algorithmic self-assembly has led to an extensive analysis among numerous models that reflect the behavior of microscopic organisms and processes in nature. Some of these models include the abstract Tile Assembly Model (aTAM) [19] in which non-rotatable tiles attach to existing structures, the 2-Handed Assembly Model (2HAM) [13], where 2 existing structures can attach at a time, and the Signal-passing Tile Assembly model (STAM) [12] that allows for both tile detachment and dynamic behavior between tiles as the system grows. Although these models are theoretical, they are motivated by natural mechanisms. Approaching solutions from the implementation side, a lot of work has been done experimentally in not only building complex structures [16, 17, 19], but also performing non-trivial computation [7, 20]. However, from both a theoretical and experimental viewpoint, some of the most intriguing questions are those that study the geometric limitations of each model such as whether it is possible to build some specific shape in a given model.

For many of these models, general shape building has been resolved. For instance, in the aTAM, any finite shape may be constructed if finite scaling is allowed [18]. For specific types of shapes though, such as different classes of fractals, most models still have open questions. For infinite discrete self-similar fractals (DSSFs), some shapes are strictly buildable (without error) [3], while others exhibiting specific properties such as "pinch point" fractals are not [2, 8]. Recent work in SODA 2025 [4] has provided a complete, polynomial-time decidable categorization of which DSSFs are buildable within the aTAM.

While many DSSFs can be weakly assembled (with error) at temperature 1 in the aTAM, there does not exist a DSSF that can be strictly assembled at temperature 1 [14]. In contrast, the 2HAM and STAM models yield more positive results. The work of [6, 10] shows that in comparison to the aTAM, the 2HAM (and k-HAM) can build a larger class of discrete self-similar fractals. In [9], they show that any arbitrary DSSF can be strictly assembled in the STAM with detachments. Even without detachments, the limited number of times a STAM tile can change states makes some fractals impossible to build at any temperature.

Thus, there exists a middle ground between the growth only, no detachment aTAM, where a limited number of fractals are strictly buildable, in comparison to the state-changing STAM with detachment, where every arbitrary DSSF is strictly buildable. Our work focuses on bridging this gap through a study of building DSSFs in the seeded Tile Automata Model (seeded TA), a model that differs from the passive assembly aTAM by the ability for adjacent tiles to actively change states similar to cellular automata, but does not allow the powerful operation of the detachment of placed tiles. TA can also be simulated by the STAM [5].

Most related to our work is that of [11], in which it was shown that for a special class of DSSFs with "ham-path generators", there exists a seeded TA system Γ that can strictly build the DSSF. Our work focuses on extending these results to any general DSSF in two ways. We first show that for every arbitrary fractal X, there exists a seeded TA system that strictly builds X infinitely with a finite number of states, transitions, and affinities. We then show that under these constructions, there exists a *single* seeded TA system Γ that can strictly build any DSSF infinitely at temperature 1, thus providing a full characterization for fractal generation in seeded TA. An emulator of this seeded TA system can be found at `https://github.com/rknobel1/fractals`.

2 Preliminaries

This section defines the model, discrete self-similar fractals, and strictly building shapes similar to the previous definitions in [1, 15].

Seeded Tile Automata

Let Σ denote a set of *states* or symbols. A tile $t = (\sigma, p)$ is a non-rotatable unit square placed at point $p \in \mathbb{Z}^2$ and has a state of $\sigma \in \Sigma$. An *affinity function* Π over a set of states Σ takes an ordered pair of states $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ and an orientation $d \in D$, where $D = \{\perp, \vdash\}$, and outputs an element of $\mathbb{Z}^{0+}$. The orientation d is the relative position to each other with $\perp$ meaning vertical and $\vdash$ meaning horizontal, with the σ_1 being the west or north state respectively. A *transition rule* consists of two ordered pairs of states $(\sigma_1, \sigma_2), (\sigma_3, \sigma_4)$ and an orientation $d \in D$, where $D = \{\perp, \vdash\}$. This denotes that if the states (σ_1, σ_2) are next to each other in orientation d (σ_1 as the west/north state) they may be replaced by the states (σ_3, σ_4). An *assembly* A is a set of tiles with states in Σ such that for every pair of tiles $t_1 = (\sigma_1, p_1), t_2 = (\sigma_2, p_2), p_1 \neq p_2$. Informally, each position contains at most one tile.

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

A *Seeded Tile Automata* system is a $6-$tuple $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, s, \tau)$ where Σ is a set of states, $\Lambda \subseteq \Sigma$ a set of initial states, Π is an affinity function, Δ is a set of transition rules, s is a stable assembly called the seed assembly, and τ is the temperature (or threshold). A tile $t = (\sigma, p)$ may attach to an assembly A at temperature τ to build an assembly $A' = A \cup t$

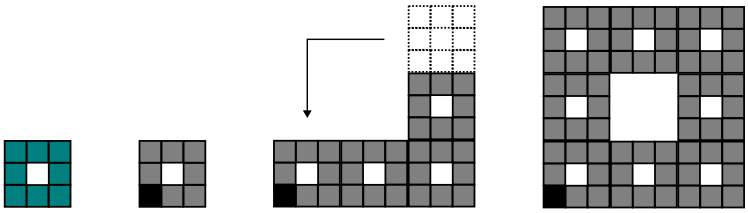

Figure 1 A generator for the Sierpinski square and how it's built. From left to right: a feasible generator, the fractal in stage 1, the fractal between stages 1 and 2, and the fractal in stage 2.

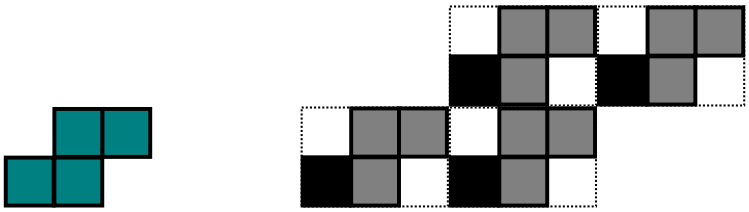

Figure 2 An example of a non-feasible generator (left) and the fractal in stage 2 (right). Note that the fractal is no longer connected in stage 2, as there does not exist a pair of points that lie on the left/right edges of the bounding box for the generator.

if A' is $\tau-$stable and $\sigma \in \Lambda$. We denote this as $A \to_{\Lambda,\tau} A'$. An assembly A can transition to an assembly A' if there exist two neighboring tiles $t_1 = (\sigma_1, p_1), t_2 = (\sigma_2, p_2) \in A$ (where t_1 is the west or north tile) such that there exists a transition rule in Δ with the first pair being (σ_1, σ_2), the second pair being some pair of states (σ_3, σ_4) such that $A' = (A \setminus \{t_1, t_2\}) \cup \{t_3 = (\sigma_3, p_1), t_4 = (\sigma_4, p_2)\}$. We denote this as $A \to_\Delta A'$. For this paper, we focus on systems of temperature $\tau = 1$, and all bond strengths are equal to 0 or 1.

An assembly sequence $\overrightarrow{\alpha} = \{\alpha_0, \alpha_1, \ldots\}$ in Γ is a (finite or infinite) sequence of assemblies such that each $\alpha_i \to_{\Lambda,\tau} \alpha_{i+1}$ or $\alpha_i \to_\Delta \alpha_{i+1}$. An assembly sub-sequence $\beta = \{\alpha'_0, \alpha'_1, \ldots\}$ in Γ is a (finite or infinite) sequence of assemblies such that for each α'_i, α'_{i+1} there exists an assembly sequence $\overrightarrow{\alpha} = \{\alpha'_i, \ldots, \alpha'_{i+1}\}$. We define the *shape* of an assembly A, denoted $(A)_\Lambda$, as the set of points $(A)_\Lambda = \{p | (\sigma, p) \in A\}$.

Discrete Self-Similar Fractals

Let $1 < c, d \in \mathbb{N}$ and $X \subseteq \mathbb{N}^2$. We say that X is a $(c \times d)$-discrete self-similar fractal if there is a set $G \subseteq \{0, \ldots, c-1\} \times \{0, \ldots, d-1\}$ with $(0,0) \in G$, such that $X = \bigcup_{i=1}^{\infty} G_i$, where G_i is the i^{th} *stage* of G satisfying $G_0 = \{(0,0)\}$, $G_1 = G$, and $G_{i+1} = \{(a,b) + (c^i v, d^i u) | (a,b) \in G_i, (v,u) \in G\}$. In this case, we say that G *generates* X (see Figure 1 for an example). We say that X is a discrete self-similar fractal if it is a $(c \times d)$-discrete self-similar fractal for some $c, d \in \mathbb{N}$. A generator G is termed *feasible* if it is a connected set, and there exist (not necessarily distinct) points $(0, y), (c-1, y), (x, 0), (x, d-1) \in G$, i.e., a pair of points on each opposing edge of the generator bounding box that share the same row or column. Note that the fractal generated by a generator is connected if and only if the generator is feasible. For the remainder of this paper we only consider feasible generators. An example of a non-feasible generator is shown in Figure 2.

Strict Self-Assembly. Let X be a discrete self-similar fractal with a feasible generator G. Consider a seeded TA system $\Gamma = (\Sigma, \Lambda, \Pi, \Delta, s, \tau)$ with $(s)_\Lambda = G$, and let S denote the set of all valid assembly sequences for Γ. Γ *strictly* builds X if $\forall \overrightarrow{\alpha} = \{s, \alpha_1, \ldots, \alpha_i, \ldots\} \in S$, $\overrightarrow{\alpha}$ is infinite and $\lim_{i \to \infty} (\alpha_i)_\Lambda = X$.

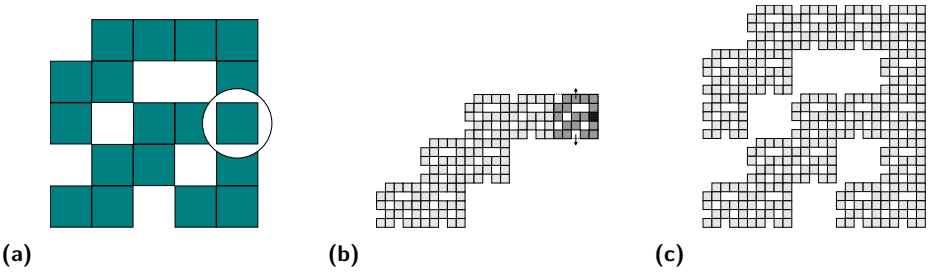

Figure 3 (a) A feasible generator. (b) The fractal between stages 1 and 2. The region shaded dark corresponds to the circled tile from (a), with the black tile denoting the pseudo-seed for this region. (c) The fractal in stage 2. This fractal becomes the new generator, and the process is repeated.

Other Notation. Let A be an assembly. We denote A as A_i if $(A)_\Lambda = G_i$. We refer to a particular sub-assembly of A as $A_j^{(w,z)}$, where $(A_j^{(0,0)})_\Lambda = G_j$, if $(A_j^{(w,z)})_\Lambda = \{(a,b) + (c^j w, d^j z) \mid (a,b) \in (A_j^{(0,0)})_\Lambda\}$ for $0 \leq w < c$ and $0 \leq z < d$. For convenience, we also refer to a sub-assembly with shape $(A_j^{(w,z)})_\Lambda$ as $(A_j^{(a,b)})_\Lambda + (l \cdot c^j, m \cdot d^j)$ for $l = w - a$ and $m = z - b$, or in other words, a translation of $A_j^{(a,b)}$ $l \cdot c^j$ units horizontally and $m \cdot d^j$ units vertically.

Let G be a feasible generator with points $(0,y), (c-1,y), (x,0), (x,d-1) \in G$ and X be the discrete self-similar fractal corresponding to G. We denote *key positions* for some A_i as four points $p_N, p_E, p_W, p_S \in (A_i)_\Lambda$ satisfying $p_N = (x + c^{i-1} \cdot x, d^i - 1)$, $p_E = (c^i - 1, y + d^{i-1} \cdot y)$, $p_W = (0, y + y \cdot d^{i-1})$ and $p_S = (x + c^{i-1} \cdot x, 0)$. The four tiles $t_N, t_E, t_W, t_S \in A_i$ with positions p_N, p_E, p_W, p_S, respectively, are called *key tiles*. We denote $t_0 \in G$ as the *origin tile* if t_0 has position $(0,0)$.[1]

3 The Complete Algorithm

A formal description of the complete algorithm is shown in Algorithm 1 and is based on the sub procedures detailed throughout Sections 5, 6, and 7, which include Algorithms 5 through 14. Let G be a generator for DSSF X, with S denoting the seed assembly generated from Section 5. The basic idea behind this algorithm is shown in Figure 3.

4 Construction Preliminaries

We start by detailing the notation used in Sections 6 and 7. With the ability for tiles to transition between multiple states, it is possible to encode information as the state of the tile and update this state as the assembly grows. Thus, for ease of understanding, we consider tiles as a *class*, with each tile t an *object* of this class with accessible *fields*. We denote accessing a specific field as *t.field_name*. We also make use of *functions* in standard form *function_name(parameters)*.

[1] While not all generators contain the origin (e.g., the 5-point cross), we consider generators that do have it for simplicity and note that our constructions work for any arbitrarily chosen tile as the origin tile.

Algorithm 1 The full algorithm for building fractal X starting from generator G. At the end of the algorithm, the newly created assembly is treated as the new seed assembly S, and the process restarts.

Input: A generator G, seed assembly S, origin tile t_o

```
1:  procedure BuildFractal
2:      stack ← [t_o]
3:      while len(stack) > 0 do
4:          t_p ← stack.pop()
5:          ChooseCopyDirection(t_p)
6:          t ← LocateFirstTile(t_p)                    ▷ Locates first tile to be copied
7:          while True do                               ▷ Copy all tiles in sub-assembly
8:              CreateCopy(t)
9:              TransferSignal(t)
10:             t' ← RetraceSignal(t)
11:             t ← LocateNextTile(t)
12:             if num_completed_sub-assemblies(t) = len(t.neighbors) then
13:                 stack.append(LocatePseudoSeed(t'))
14:                 ResetCopiedSubAssembly(t))
15:                 if t_p.num_times_copied = len(t_p.neighbors) then
16:                     MarkingSubAssembly(t_p)         ▷ No pending directions to copy
17:                 else
18:                     stack.append(t_p)               ▷ Still pending direction to copy
19:                 break
20:     GlobalResetting                                 ▷ No sub-assemblies have pending directions to copy
```

4.1 Stored Tile Information

The following are accessible fields for a given tile t, i.e., these are the attributes of the object. Let $d \in \{N, E, W, S\}$.

- ***state.*** Takes a value of *None*, *P*, *W*, *F* with a default of *None*. These values denote whether a tile is not ready to produce (None), has not placed itself yet (Producing), is currently placing itself (Waiting), or has placed itself (Finished).
- ***copy_direction.*** Takes a value of *N*, *E*, *W*, *S*, *None* or *r* with a default of *None*. These values denote if t should be copied in the sub-assembly to the north (N), east (E), west (W), south (S), or not (None), or if the sub-assembly is being reset (r).
- ***origin_direction.*** Takes a value of *N*, *E*, *W*, *S*, ***. These values denote the direction to the original seed. A value of * denotes that t is the original seed.
- ***key_d.*** Takes a value of *N*, *E*, *W*, *S*, ***. These values denote the direction to key tile t_d. A value of * denotes that t is the key tile for direction d.
- ***new_key_tile.*** Takes a value of *True* or *False* with a default of *False*. These values denote if t will become a key tile for the growing assembly (True) or not (False).
- ***neighbors.*** Takes a value of the power-set $P(\{N, E, W, S\}) \setminus \emptyset$. These values denote the direction to some (but not always *all*) adjacent tiles to t. Informally, we say $t_a \in t.neighbors$ if t_a is in direction d from t, where $d \in t.neighbors$.
- ***new_neighbors.*** Takes a value of the power-set $P(\{N, E, W, S\})$ with a default of *None*. These values denote the direction to any new neighbors of t as the assembly grows.
- ***state_d.*** Takes a value of *N*, *W*, *M*, *Y*, *None*. These values denote whether the sub-assembly stemming from direction d is not complete (N), placing a tile (W), placed a tile and might now be complete (M), complete (Y), or does not exist (None).
- ***transfer.*** Points to a created tile t_c or takes a value of ?, *r*, or *None* with a default of *None*. These values denote whether the tile is passing a signal or not, or if the sub-assembly is ready to globally reset (r).

Algorithm 2 The "next" direction at a given tile t. N, E, W, S represent the 4 cardinal directions.

Input: A tile t
Output: A direction $d \in \{N, E, W, S\}$
1: **procedure** NEXT
2: **if** $N \notin t.caps \wedge N \in t.neighbors$ **then return** $\boldsymbol{N}$
3: **else if** $E \notin t.caps \wedge E \in t.neighbors$ **then return** $\boldsymbol{E}$
4: **else if** $W \notin t.caps \wedge W \in t.neighbors$ **then return** $\boldsymbol{W}$
5: **else return** $\boldsymbol{S}$

- ***terminal.*** Takes a value of *True* or *False*. These values denote if a tile is terminal (True) or not (False).
- ***caps.*** Takes a value of the power-set $P(\{N, E, W, S\})$ with a default of *None*. These values denote that the sub-assembly stemming from each direction in $t.caps$ is complete.
- ***pseudo_seed.*** Takes a value of *True* or *False* with a default of *False*. These values denote if the current tile is a pseudo-seed (True) or not (False).
- ***will_be_pseudo_seed.*** Takes a value of *True* or *False* with a default of *False*. These values denote if the current tile will be a pseudo-seed (True) or not (False) for the created assembly.
- ***original_seed.*** Takes a value of *True* or *False* with a default of *False*. These values denote if the current tile is the origin tile (True) or not (False).
- ***copied.*** Takes a value of *True* or *False* with a default of *False*. These values denote if t is a copied tile (True) or not (False).
- ***sub_assembly_copied.*** Takes a value of *True* or *False* with a default of *False*. These values denote if the sub-assembly corresponding to t is complete (True) or not (False).
- ***first_tile.*** Takes a value of *True* or *False* with a default of *False*. These values denote if t is the first tile copied for a sub-assembly (True) or not (False).
- ***can_place.*** Takes a value of *True* or *False* with a default of *False*. These values denote if t can copy itself (True) or not (False).
- ***num_times_copied.*** Takes a value $\in \{0, 1, 2, 3, 4\}$ with a default of *0*. These values denote the number of times the sub-assembly corresponding to t has been copied.
- ***counter.*** Takes a value $\in \{0, 1, 2, 3, 4\}$ or *None* with a default of *None*. This counter field is used primarily for synchronization.
- ***temp.*** Default of *None*. Used to store temporary information.

4.2 Functions

The following details certain functions that will be used.

- ***opp(d).*** Takes a direction d and outputs the "opposite" direction $N \to S$, $E \to W$, $W \to E$, $S \to N$.
- ***retrieve_tile(t, d).*** Takes a tile t and direction d and outputs the neighboring tile in direction d.
- ***num_comp_sub-assemblies(t).*** Takes a tile t and outputs the number of $state\_d = Y$, or in other words, the number of completed sub-assemblies stemming from t.
- ***next(t).*** Takes a tile t and outputs the "next" direction for a signal from tile t as shown in Algorithm 2.
- ***next_tile_placed(t).*** Takes a tile t with $t.state = F$ and outputs the next tile to get placed as shown in Algorithm 3.
- ***breadcrumb(t).*** Takes a tile t and outputs the direction of the "breadcrumb" left by a signal as shown in Algorithm 4.

Algorithm 3 The "next" tile to be placed from tile t. P denotes a "producing" state. N, E, W, S represent the 4 cardinal directions.

Input: A tile t
Output: A tile t_a
procedure NextTilePlaced
  if $retrieve\_tile(t, N).state = P \wedge N \in t.neighbors$ **then**
    return retrieve_tile(t, N)
  else if $retrieve\_tile(t, E).state = P \wedge E \in t.neighbors$ **then**
    return retrieve_tile(t, E)
  else if $retrieve\_tile(t, W).state = P \wedge W \in t.neighbors$ **then**
    return retrieve_tile(t, W)
  else return retrieve_tile(t, S)

Algorithm 4 The direction of the "breadcrumb" trail at a tile t. "W" and "M" denote "waiting" and "maybe" states, respectively. N, E, W, S represent the 4 cardinal directions.

Input: A tile t
Output: A direction $d \in \{N, E, W, S\}$
procedure Breadcrumb
  if $state\_N =$"W" $\vee$ $state\_N =$"M" **then return N**
  else if $state\_E =$"W" $\vee$ $state\_E =$"M" **then return E**
  else if $state\_W =$"W" $\vee$ $state\_W =$"M" **then return W**
  else return S

5 Seed Initialization

This section details how the initial seed for the seeded TA system is constructed given a feasible generator G with key positions p_N, p_E, p_W, p_S. The seed is a blank assembly A in which each point $p \in G$ contains a tile. Each tile begins with default values, and with the tile t_o located at the origin having $t_o.origin\_seed = True$. An example of an initial seed assembly is shown in Figure 4a.

Neighbor Initialization. The first step is to initialize the *origin_direction*, *neighbors*, *terminal*, and *state_d* fields of each tile, which is achieved by deriving a spanning tree (ST) for the seed assembly from the generator. Start by running a breadth-first search (BFS) from the original seed in which each tile is a vertex and edges exist between cardinally adjacent vertices, leaving a breadcrumb at each tile to denote the previous tile. With this information, we can now initialize the *neighbors* field for each tile t. First, for each adjacent tile t_a in direction d, if t_a is marked by t, append d to $t.neighbors$. Then, for the tile that marked t in direction d, append d to $t.neighbors$ and $t.origin\_direction$. The completion of these steps initialize the *origin_direction* and *neighbors* fields for each tile. It is then left to update the *terminal* and *state_d* fields. For each tile t and for each $d \in t.neighbors$, set $state\_d = N$. Furthermore, if $len(t.neighbors) = 1$, set $t.terminal = True$. An example of the result of this neighbor initialization is shown in Figure 4b.

Key Tile Initialization. The next step is to initialize each *key_d* field. This is achieved by running a BFS from the tile located at the key position for direction d on the ST derived from earlier. More formally, for each key tile t_d, start by running a BFS from t_d with a breadcrumb left at each tile denoting the direction d_p to the previous tile. Once all tiles have been marked, set each $t_d.key\_d = *$ to denote that t_d is the key tile for direction d. Then, for every other tile, set $t.key\_d = d_p$. An example of the result of the key tile initialization is shown in Figure 4c for the north direction.

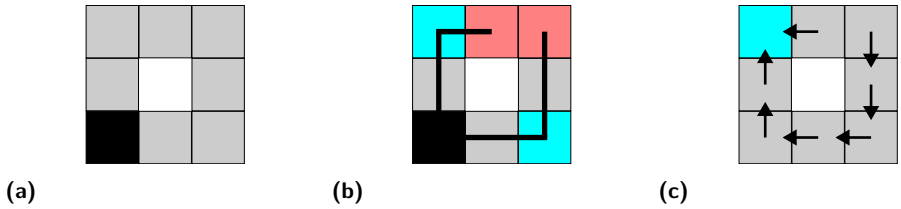

Figure 4 (a) An initial seed assembly. (b) An ST of the seed assembly from (a), with the origin tile colored black, terminal tiles colored red, and key tiles colored aqua (including the origin tile). (c) The direction from each tile to the north key tile. Note that these directions are dependent on the ST derived from (b).

6 Copying Procedure

This section focuses on how an assembly A_i is taken to assembly A_{2i}. Specifically, we consider the copying procedure for a sub-assembly $A_i^{(w_1,z_1)}$ that creates a new sub-assembly $A_i^{(w_2,z_2)}$. This procedure can be broken down into 4 steps: choosing a copying direction for $A_i^{(w_1,z_1)}$, copying each tile in $A_i^{(w_1,z_1)}$ to create $A_i^{(w_2,z_2)}$, readying the newly created $A_i^{(w_2,z_2)}$, and lastly resetting $A_i^{(w_1,z_1)}$. Each step heavily relies on synchronicity to ensure each step does not interfere with the others, which will be discussed within each sub-section.

6.1 Choosing a Copying Direction

Choosing a copying direction starts from the origin seed or the pseudo-seed. Let t_o denote this tile. The idea is that t_o looks at adjacent neighbors in each direction $d \in t.neighbors$. Let t_a be an adjacent tile to t_o in direction d. If $t_a.sub\_assembly\_copied = False$, then d becomes the chosen copying direction, as $A_i^{(w_1,z_1)}$ still needs to be copied in direction d. This choice is propagated to all tiles in $A_i^{(w_1,z_1)}$, using the *counter* field of each tile to ensure that all tiles receive this signal. Once this is done, then the next step begins. Algorithm 5 explicitly details how this process is undertaken.

6.2 Copying Tiles

With $A_i^{(w_1,z_1)}$ having chosen a direction d_c to copy, the next step is to copy each of the tiles in $A_i^{(w_1,z_1)}$ to create $A_i^{(w_2,z_2)}$. Let t_{d_c} denote the key tile for $A_i^{(w_1,z_1)}$ in direction d_c. Note that all signals between $A_i^{(w_1,z_1)}$ and $A_i^{(w_2,z_2)}$ pass through t_{d_c}. Thus, the first tile that must be placed is the key tile for direction $OPP(d_c)$, which we denote as $t_{OPP(d_c)}$. We start by describing how this tile is located, then detail how and in what order each tile is copied.

6.2.1 Locating the First Tile to Place

From Section 6.1, the last tile to get updated is t_o. Thus, locating $t_{OPP(d_c)}$ is a matter of following the direction from each $t.key\_OPP(d_c)$ until $t_{OPP(d_c)}$ is reached. Algorithm 6 explicitly details how this process is undertaken.

Once $t_{OPP(d_c)}$ is found, $t_{OPP(d_c)}.first\_tile$ and $t_{OPP(d_c)}.can\_place$ are set to true. This distinction from other tiles is important for determining when $A_i^{(w_1,z_1)}$ has been completely copied and to start the sub-assembly copying procedure.

Algorithm 5 Choosing a copying direction and propagating choice. "?" denotes the pseudo-seed/origin tile can choose a copying direction.

Input: A tile t_o

```
 1: procedure ChooseCopyDirection
 2:     cd ← None
 3:     if t_o.original_seed or t_o.copy_direction ="?" then        ▷ Look for a pending direction to copy
 4:         for d in t_o.neighbors do
 5:             adj_tile ← retrieve_tile(t_o, d)
 6:             if adj_tile.sub_assembly_copied = False then
 7:                 cd ← d
 8:                 adj_tile.sub_assembly_copied ← True
 9:                 adj_tile.will_be_pseudo_seed ← True
10:                 break
11:     if cd ≠ None then                    ▷ If a pending direction to copy exists, propagate choice
12:         stack ← [t_o]
13:         while len(stack) > 0 do
14:             t ← stack.pop()
15:             t.counter ← 0
16:             if t.terminal = True then t.counter++
17:             for d in t.neighbors do
18:                 adj_tile ← retrieve_tile(t, d)
19:                 if adj_tile.copy_direction = None then
20:                     adj_tile.copy_direction ← cd
21:                     t.counter++
22:                     stack.append(adj_tile)
```

6.2.2 Copying a Tile

Copying a given tile t happens in two steps. In the first step, a copy of t is created. This copy is transferred to t_{d_c}, where it can then be transferred to $A_i^{(w_2,z_2)}$. Once the signal has reached $A_i^{(w_2,z_2)}$, the copy is transferred using the "next" direction at each tile until a tile no longer exists where the signal needs to go. The copy is then placed as a physical tile in this location using an affinity rule. Algorithm 7 details what information is copied and stored in $t.transfer$. Algorithm 8 details how a signal is passed starting from $A_i^{(w_1,z_1)}$ and ending in $A_i^{(w_2,z_2)}$. A crucial part of this signal passing is the use of "caps" to act as funnels in $A_i^{(w_2,z_2)}$. As parts of $A_i^{(w_2,z_2)}$ are finished, these caps can continue to be shifted until the entire sub-assembly is completely built.

In the second step, a signal is sent back to mark the tile being copied as complete, following the breadcrumb trail that is left behind. As stated, *caps* are created once terminal tiles get placed to prevent signals from entering already completed parts of the sub-assembly. Algorithm 9 explicitly details how the breadcrumb trail is retraced, as well as how caps are used and shifted.

6.2.3 Choosing the Next Tile to Place

Once the recently copied tile t has $t.state = F$, the following step is to select the next tile to get placed. The logic behind choosing the next tile varies depending on if t is a terminal tile or not. If $t.terminal = False$, then the tile selection is a matter of picking the adjacent tile t_a in direction $next\_tile\_placed(t)$. However, if $t.terminal = True$, then it is a matter of retracing steps until a tile is reached where some adjacent tiles have yet to be placed and selecting from these tiles. Algorithm 10 explicitly details how this process is undertaken.

Algorithm 6 Locating the first tile to be placed. "?" denotes a "searching" signal. "$*$" denotes the respective tile is the key tile for some direction d_c.

Input: A tile t_o
Output: A tile t
procedure FIRSTTILE
  $t \leftarrow t_o$
  $d_c \leftarrow t$.copy_direction
  t.transfer $\leftarrow$"?"
  while t.key_OPP(d_c) $\neq$ "$*$" **do** ▷ Propagating signal to find first tile
    adj_tile = retrieve_tile(t, t.key_OPP(d_c))
    t.transfer $\leftarrow$ None
    adj_tile.transfer $\leftarrow$ "?"
    $t \leftarrow adj\_tile$
  t.first_tile $\leftarrow$ True ▷ Update first tile fields to start the copying procedure
  t.can_place $\leftarrow$ True
  t.transfer $\leftarrow$ None
  return t

Algorithm 7 Creating a copy of a tile t to be transferred to the created sub-assembly. "$*$" denotes the respective tile is the key tile for some direction d.

Input: A tile t
Output: None
procedure CREATECOPY
  nt $\leftarrow t$ ▷ Create a copy of t
  nt.copied $\leftarrow$ True
  if t.pseudo_seed **then**
    if t.key_d = "$*$" **then**
      nt.new_key_tile $\leftarrow$ True
      nt.sub_assembly_copied $\leftarrow$ True
  else if t.origin_seed **then**
    if t.key_d = "$*$" **then**
      nt.new_key_tile $\leftarrow$ True
      nt.sub_assembly_copied $\leftarrow$ True
  else
    nt.new_key_tile $\leftarrow$ False
  if nt.will_be_pseudo_seed **then**
    nt.pseudo_seed $\leftarrow$ True
    nt.will_be_pseudo_seed $\leftarrow$ False
    nt.num_times_copied + = 1
    t.will_be_pseudo_seed $\leftarrow$ False
  t.transfer $\leftarrow$ nt

6.3 Readying the Created Sub-assembly

As the last tile in $A_i^{(w_2,z_2)}$ is placed and the breadcrumb trail is retraced, $t_{OPP(d_c)}$ from $A_i^{(w_2,z_2)}$ will contain $len(t_{OPP(d_c)}.neighbors) - 1$ caps, signifying $A_i^{(w_1,z_1)}$ has been fully copied into $A_i^{(w_2,z_2)}$. It is then left to update the state of the pseudo-seed in $A_i^{(w_2,z_2)}$ so that this sub-assembly can repeat the process. Algorithm 11 explicitly details how this process is undertaken.

6.4 Resetting the Copied Sub-assembly

The final step is to reset the copied sub-assembly. As the last tile is marked as complete and each $t_{OPP(d_c)}.state\_d$ from $A_i^{(w_1,z_1)}$ has a value of Y or $*$, this signifies the completion of the copying process. It is then left to first reset all tiles in $A_i^{(w_1,z_1)}$, then choose a new copying direction. Algorithm 12 explicitly details how this process is undertaken.

Algorithm 8 Transferring signal to place a tile in direction d. "W" denotes a "waiting" state. "$*$" denotes the respective tile is the key tile for some direction d.

Input: A tile t
Output: None

```
1:  procedure TransferSignal
2:      while t.copied=False ∧ t.key_d ≠ "*" do        ▷ Propagate signal to the key tile for direction d
3:          adj_tile ← retrieve_tile(t, t.key_d)
4:          if adj_tile.counter = len(adj_tile.neighbors) then
5:              adj_tile.transfer ← t.transfer
6:              t.transfer ← None
7:              adj_tile.temp ← adj_tile.OPP(t.key_d)
8:              adj_tile.state_OPP(t.key_d) ← "W"
9:              t ← adj_tile
10:     if retrieve_tile(t, d) = None then          ▷ Place tile if created sub-assembly does not exist yet
11:         Place t.transfer in direction d
12:         t.new_neighbors.append(d)
13:         adj_tile ← retrieve_tile(t, d)
14:         adj_tile.origin_direction ← OPP(d)
15:         adj_tile.new_neighbors.append(OPP(d))
16:     else                                        ▷ Transfer signal to sub-assembly being created
17:         t ← retrieve_tile(t, d)
18:         loop                                    ▷ Transfer signal to correct location
19:             adj_tile ← retrieve_tile(t, next(t))
20:             if adj_tile = None then             ▷ Place tile in correct location
21:                 Place t.transfer in direction d
22:                 break
23:             else                                ▷ If correct location not found yet, keep looping
24:                 adj_tile.transfer ← t.transfer
25:                 t.transfer ← None
26:                 adj_tile.temp ← adj_tile.state_OPP(next(t))
27:                 adj_tile.state_OPP(next(t)) ← "W"
28:                 t ← adj_tile
```

7 Resetting Procedure

This section details the resetting procedure once an assembly A_i has become A_{2i}. We first discuss local resetting that occurs within each sub-assembly, followed by the global reset that unifies the resulting assembly to restart the process.

Local Resetting

From Algorithm 12, each time a sub-assembly is copied, the *num_times_copied* field of the pseudo-seed/original seed is increased by one. Once this field is equal to the length of its *neighbors* field, or if the pseudo-seed/origin seed is terminal, then this sub-assembly is no longer needed to create other sub-assemblies. Thus, starting from the pseudo-seed/original seed, a signal is sent to label each tile with a "ready to reset" state. This state is accompanied by the *counter* field of each tile being set to the length of the *neighbors* field to ensure that all tiles within each sub-assembly are marked with the resetting state. Let t_p denote the pseudo-seed seed/original seed for some sub-assembly. Algorithm 13 explicitly details how this procedure is undertaken.

Global Resetting

From the local resetting that occurs within each sub-assembly, tiles are marked terminal or not depending on the number of new neighbors each tile will have. As each tile is updated, global resetting occurs from the new terminal tiles. This resetting includes setting each tiles *neighbors*

Algorithm 9 Retracing signal to mark the tile being copied as finished. "W", "M", "N" and "F" denote "waiting", "maybe", "not completed", and "completed" states, respectively.

Input: A tile t
Output: A tile t_p

```
1:  procedure RetraceSignal
2:      t.transfer ← None
3:      t_p ← None
4:      prev_tile ← retrieve_tile(t, breadcrumb(t))
5:      if len(t.caps) = t.neighbors - 1 ∧ prev_tile.copy_direction = d then t_p ← t
6:      while prev_tile.status ≠ "W" do          ▷ Retrace signal to tile that was just copied
7:          t.state_breadcrumb(t) ← t.temp
8:          prev_tile.state_breadcrumb(prev_tile) ← "M"
9:          t.temp ← None
10:         if len(t.caps) = len(t.neighbors) - 1 then
11:             if prev_tile.copy_direction = d then
12:                 Place t.transfer in direction d
13:             else
14:                 prev_tile.caps.append(OPP(breadcrumb(t)))
15:         t ← prev_tile
16:     t.state_breadcrumb(t) ← "N"                ▷ Update fields of the copied tile
17:     prev_tile.status ← "F"
18:     return t_p
```

field to the *new_neighbors* field, as well as updating the following fields to their default values: *state*, *copy_direction*, *pseudo_seed*, *will_be_pseudo_seed*, *sub_assembly_copied*, *caps*, *copied*, *num_times_copied*, *temp*, *transfer*, *new_neighbors*, *first_tile*, *can_place* and *counter*.

Additionally, the directions to the new key tiles are updated, as well as the location of these key tiles as well, which turns A_{2i} into the new generator. These directions are updated starting from the terminal tiles for each tile t based on the following logic:

1. If $t.new\_key\_tile = True$, set $t.key\_d = *$ for each direction which t is a key tile.
2. If $t.terminal$ and $t.new\_key\_tile = False$, then each $t.key\_d$ field is set to $t.origin\_direction$.
3. For each adjacent tile t_a not in the direction of the original seed, if $t_a.key\_d$ is not in the direction of t, set $t.key\_d$ to the direction of t_a.
4. Otherwise, $t.key\_d$ must be in the direction of the original seed.

Algorithm 14 explicitly details how this procedure is undertaken. Once the original seed is reset, then the newly created assembly A_{2i} becomes the new generator for the next step, and the entire procedure restarts. An important thing to note is each tile t is only reset if $t.transfer = r$ and if $t.counter = 0$, which ensures all tiles in the respective sub-assembly have been given this reset signal.

8 Correctness

The following theorems are used to prove the main results of this paper. Intuitively, a *base-assembly* is an assembly which represents a valid output from the seed initialization as described in Section 5. These assemblies have 4 key tiles with exactly 1 path between any 2 tiles in the assembly. Thus, by taking a base-assembly A_i to base-assembly A_{2i}, this ensures that A_{2i} can become the new "seed" assembly for the system. A *base-sub-assembly* is essentially the same, except field values for tiles are not required to be defaulted. Examples of base-assemblies and base-sub-assemblies are shown in Figure 5. '

▶ **Theorem 1.** *Algorithms 5, 6, 7, 8, 9, 10, 11, 12, 13, and 14 from Sections 6 and 7 occur sequentially.*

Algorithm 10 Locating the next tile to be placed. "Y" denotes a "complete" state.

Input: A tile t
Output: A tile t_a

```
procedure LocateNextTile
    if t.terminal then                          ▷ Retrace steps to a tile with an uncopied neighbor
        adj_tile ← retrieve_tile(t, t.neighbors)
        adj_tile.state_OPP(t.neighbors) ← "Y"
        t ← adj_tile
        while num_comp_sub-assemblies(t) = len(t.neighbors) do
            for n in t.neighbors do
                adj_tile ← retrieve_tile(t, n)
                if num_comp_sub-assemblies(adj_tile) < len(t.neighbors) then
                    adj_tile.state_OPP(n) ← "Y"
                    t ← adj_tile
                    break
    adj_tile ← retrieve_tile(t, next_tile_placed(t))          ▷ Select next tile to copy
    adj_tile.state_OPP(next_tile_placed(t)) ← "Y"
    adj_tile.can_place ← True
    t_a ← t
    return t_a
```

Algorithm 11 Locating the pseudo-seed of the created sub-assembly. "?" denotes a "searching" signal.

Input: A tile $t_{OPP(d_c)}$
Output: A tile t_p

```
procedure LocatePseudoSeed
    stack ← [t_OPP(d_c)]
    while len(stack) > 0 do                    ▷ Propagate signal to find pseudo-seed
        t ← stack.pop()
        if t.pseudo_seed then
            t_p ← t
            return t_p
        t.copy_direction ← "?"
        for d in t.neighbors do
            adj_tile ← retrieve_tile(t, d)
            if adj_tile.copy_direction ≠ "?" then stack.append(adj_tile)
```

▶ **Theorem 2.** *Given $A_i^{(a,b)}$, which represents a sub-assembly prior to applying Algorithm 5, and $A_i^{(x,y)}$, which represents the created sub-assembly after applying Algorithms 5, 6, 7, 8, 9, and 10 to $A_i^{(a,b)}$. $(A_i^{(x,y)})_\Lambda = (A_i^{(a,b)})_\Lambda + (j \cdot c^i, k \cdot d^i) \mid (j,k) \in \{(1,0),(0,1),(-1,0),(0,-1)\}$.*

▶ **Theorem 3.** *Given A_i, which represents the assembly prior to applying Algorithms 5 - 14. After global resetting, as described in Algorithm 14, the resulting assembly A is a base-assembly with $(A)_\Lambda = (A_{2i})_\Lambda$.*

8.1 Main Results

We now state the main results of this paper.

▶ **Theorem 4.** *For every feasible generator G, there exists a seeded TA system Γ which strictly builds the corresponding fractal with $O(1)$ states, transitions, and affinities.*

Proof. By construction. Given a generator G, initialize an assembly A as described in Section 5. Then, following Sections 6 and 7 (using Algorithms 5, 6, 7, 8, 9, 10, 11, 12, 13, and 14), create system Γ which, by Theorem 3, creates a new assembly A_{2i} with $(A_{2i})_\Lambda = G_{2i}$. Since A_{2i} is a base-assembly, repeat with A_{2i} as the new generator for Γ. Doing so infinitely then strictly builds the corresponding fractal.

Algorithm 12 Resetting the copied sub-assembly. "r" denotes a "reset-ready" state.

Input: A tile $t_{OPP(d_c)}$
Output: None

```
procedure ResetCopiedSubAssembly
    stack ← [t_OPP(d_c)]
    while len(stack) > 0 do                          ▷ Propagate reset-ready state to all neighbors
        t ← stack.pop()
        t.copy_direction ← "r"
        for d in t.neighbors do
            adj_tile ← retrieve_tile(t, d)
            if adj_tile.copy_direction ≠ "r" then
                stack.append(adj_tile)
                t.counter++
        t.status ← None                              ▷ Reset fields for tiles once signal is propagated
        t.caps ← None
        t.temp ← None
        t.transfer ← None
        t.can_place ← False
        t.first_tile ← False
        if t.pseudo_seed ∨ t.original_seed then t.num_times_copied++
```

Algorithm 13 Marking a sub-assembly with "ready to reset" states. "r" denotes a "reset-ready" state.

Input: A tile t_p
Output: None

```
procedure MarkingSubAssembly
    stack ← [t_p]
    while len(stack) > 0 do                          ▷ Propagate reset-ready state
        t ← stack.pop()
        t.transfer ← "r"
        t.counter ← len(t.neighbors) - 1
        if t.pseudo_seed then t.counter++
        if len(t.new_neighbors) > 1 then             ▷ Update terminal field
            t.terminal ← False
        else
            t.terminal ← True
            t.key_d ← None
        for n in t.neighbors do                      ▷ Propagate signal to all neighbors
            adj_tile ← retrieve_tile(t, n)
            if adj_tile.transfer ≠ "r" then
                stack.append(adj_tile)
                t.counter−−
```

Algorithm 14 Global Resetting from Terminal Tiles. "N" denotes a not completed state. "r" denotes a "reset-ready" state.

```
procedure GlobalResetting
    stack ← [all terminal tiles]
    while len(stack) > 0 do
        t ← stack.pop()
        t.counter−−
        if t.counter = 1 ∧ t.transfer = "r" then   ▷ Reset current tile if signal has been propagated
            reset(t)
            for n in t.neighbors do
                adj_tile ← retrieve_tile(t, n)
                if t.terminal ∧ t.new_key_tile then
                    if t.key_d then adj_tile.key_d ← OPP(n)
                if adj_tile.transfer = "r" then stack.append(adj_tile)
                t.state_d ← "N"
            t.transfer ← None
```

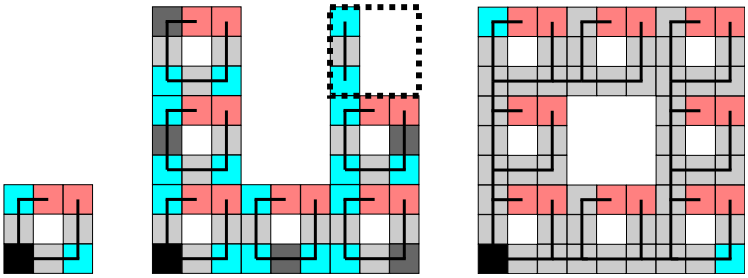

Figure 5 Examples of base-assemblies and base-sub-assemblies. Key tiles are colored aqua, terminal tiles are colored red, pseudo-seeds are colored dark gray and the black tile denotes the origin tile. The center assembly is not a base-assembly as it 1) does not have one key tile per direction, 2) does not have paths between all tiles and the origin tile (black), and 3) not all fields are defaulted (the top right corner is currently being copied). Additionally, the dotted region is not a base-sub-assembly, as there does not exist one key tile per direction and there is no pseudo-seed.

Since each tile can have at most 4 neighbors, the information stored within each tile (as described in Section 4.1) must be constant. Thus, the number of transition and affinity rules must also be constant. ◀

▶ **Theorem 5.** *There exists a single seeded TA system Γ that, given as input any feasible generator G as a seed, will strictly build the corresponding fractal for G.*

Proof. An important detail about the constructions from Sections 5, 6, and 7 is that tile placement and signal propagation are not hard-coded to the shape of the given generator. Following from Theorem 4, by considering all possible valid states, transitions, and affinity rules, there must then exist a single seeded TA system Γ which can strictly build any feasible generator at temperature 1. ◀

9 Conclusion

This paper extends the work of [11], showing that there not only exists a seeded TA system that can strictly build any fractal with a feasible generator, but also that there exists a *single* system that can do so at temperature 1. This approach takes advantage of the ability to encode information in the state of each tile, allowing for signal transmission and synchronization despite the non-deterministic nature of the model at hand and the geometric bottlenecks of many fractals. While this paper fully characterizes fractal generation in seeded TA, there are still some interesting open questions at hand:

- Are there more efficient ways to build such fractals at higher temperatures? In other words, is it possible to reduce the number of states, transitions, and affinities by encoding some information in the strength of each bond?
- Does there exist a natural extension to the model that can handle non-feasible generators?

References

1 Robert M. Alaniz, David Caballero, Sonya C. Cirlos, Timothy Gomez, Elise Grizzell, Andrew Rodriguez, Robert Schweller, Armando Tenorio, and Tim Wylie. Building squares with optimal state complexity in restricted active self-assembly. *Journal of Computer and System Sciences*, 138:103462, 2023. doi:10.1016/j.jcss.2023.103462.

2 Kimberly Barth, David Furcy, Scott M Summers, and Paul Totzke. Scaled tree fractals do not strictly self-assemble. In *Unconventional Computation and Natural Computation: 13th International Conference, UCNC 2014, London, ON, Canada, July 14-18, 2014, Proceedings 13*, pages 27–39. Springer, 2014. doi:10.1007/978-3-319-08123-6_3.

3 Florent Becker, Daniel Hader, and Matthew J. Patitz. Strict self-assembly of discrete self-similar fractals in the abstract tile-assembly model, 2024. arXiv:2406.19595.

4 Florent Becker, Daniel Hader, and Matthew J. Patitz. Strict self-assembly of discrete self-similar fractals in the abstract tile-assembly model. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 2025.

5 Angel A. Cantu, Austin Luchsinger, Robert Schweller, and Tim Wylie. Signal Passing Self-Assembly Simulates Tile Automata. In *31st International Symposium on Algorithms and Computation*, volume 181 of *ISAAC'20*, pages 53:1–53:17, 2020. doi:10.4230/LIPIcs.ISAAC.2020.53.

6 Cameron T. Chalk, Dominic A. Fernandez, Alejandro Huerta, Mario A. Maldonado, Robert T. Schweller, and Leslie Sweet. Strict self-assembly of fractals using multiple hands. *Algorithmica*, 76(1):195–224, September 2016. doi:10.1007/s00453-015-0022-x.

7 Constantine Glen Evans. *Crystals that count! Physical principles and experimental investigations of DNA tile self-assembly*. PhD thesis, California Institute of Technology, 2014.

8 David Furcy and Scott M Summers. Scaled pier fractals do not strictly self-assemble. *Natural Computing*, 16:317–338, 2017. doi:10.1007/S11047-015-9528-Z.

9 Jacob Hendricks, Meagan Olsen, Matthew J. Patitz, Trent A. Rogers, and Hadley Thomas. Hierarchical self-assembly of fractals with signal-passing tiles. *Natural computing*, 17:47–65, November 2018. doi:10.1007/s11047-017-9663-9.

10 Jacob Hendricks and Joseph Opseth. Self-assembly of 4-sided fractals in the two-handed tile assembly model. In Matthew J. Patitz and Mike Stannett, editors, *Unconventional Computation and Natural Computation*, pages 113–128, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-58187-3_9.

11 Ryan Knobel, Adrian Salinas, Robert Schweller, and Tim Wylie. Building discrete self-similar fractals in seeded tile automata. URL: https://par.nsf.gov/biblio/10562746.

12 Jennifer E. Padilla, Matthew J. Patitz, Robert T. Schweller, Nadrian C. Seeman, Scott M. Summers, and Xingsi Zhong. Asynchronous signal passing for tile self-assembly: Fuel efficient computation and efficient assembly of shapes. *International Journal of Foundations of Computer Science*, 25:459–488, 2014. doi:10.1142/S0129054114400061.

13 Matthew J. Patitz. An introduction to tile-based self-assembly. In Jérôme Durand-Lose and Nataša Jonoska, editors, *Unconventional Computation and Natural Computation*, pages 34–62, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-32894-7_6.

14 Matthew J Patitz and Scott M Summers. Self-assembly of discrete self-similar fractals. *Natural Computing*, 9:135–172, 2010. doi:10.1007/S11047-009-9147-7.

15 Matthew J. Patitz and Scott M. Summers. Self-assembly of discrete self-similar fractals. *Natural computing*, 9:135–172, August 2010. doi:10.1007/s11047-009-9147-7.

16 Paul W. K. Rothemund. *Theory and experiments in algorithmic self-assembly*. PhD thesis, University of Southern California, 2001. URL: https://go.openathens.net/redirector/utrgv.edu?url=https://www.proquest.com/dissertations-theses/theory-experiments-algorithmic-self-assembly/docview/276028769/se-2.

17 Paul W. K Rothemund, Nick Papadakis, and Erik Winfree. Algorithmic self-assembly of dna sierpinski triangles. *PLOS Biology*, 2(12):null, December 2004. doi:10.1371/journal.pbio.0020424.

18 David Soloveichik and Erik Winfree. Complexity of self-assembled shapes. *SIAM Journal on Computing*, 36(6):1544–1569, 2007. doi:10.1137/S0097539704446712.

19 Erik Winfree. *Algorithmic self-assembly of DNA*. PhD thesis, California Institute of Technology, 1998. URL: https://go.openathens.net/redirector/utrgv.edu?url=https:

//www.proquest.com/dissertations-theses/algorithmic-self-assembly-dna/docview/304420561/se-2.

20 Damien Woods, David Doty, Cameron Myhrvold, Joy Hui, Felix Zhou, Peng Yin, and Erik Winfree. Diverse and robust molecular algorithms using reprogrammable dna self-assembly. *Nature*, 567(7748):366–372, 2019. doi:10.1038/S41586-019-1014-9.