Complexity of Self-Assembled Shapes
(Extended Abstract⋆)

David Soloveichik and Erik Winfree

California Institute of Technology, Pasadena, CA 91125, USA
{dsolov,winfree}@caltech.edu

Abstract. The connection between self-assembly and computation suggests that a shape can be considered the output of a self-assembly "program," a set of tiles that fit together to create a shape. It seems plausible that the size of the smallest self-assembly program that builds a shape and the shape's descriptional (Kolmogorov) complexity should be related. We show that under the notion of a shape that is independent of scale this is indeed so: in the Tile Assembly Model, the minimal number of distinct tile types necessary to self-assemble an arbitrarily scaled shape can be bounded both above and below in terms of the shape's Kolmogorov complexity. As part of the proof of the main result, we sketch a general method for converting a program outputting a shape as a list of locations into a set of tile types that self-assembles into a scaled up version of that shape. Our result implies, somewhat counter-intuitively, that self-assembly of a scaled up version of a shape often requires fewer tile types, and suggests that the independence of scale in self-assembly theory plays the same crucial role as the independence of running time in the theory of computability.

1 Introduction

Self-assembly is the process by which an organized structure can spontaneously form from simple parts. The Tile Assembly Model [15, 14], based on Wang tiling [13], formalizes the two-dimensional self-assembly of square units called "tiles" using a physically plausible abstraction of crystal growth. In this model, a new tile can adsorb to a growing complex if it binds strongly enough. Each of the four sides of a tile has an associated bond type that interacts with a certain strength with matching sides of other tiles. The process of self-assembly is initiated by a single seed tile and proceeds via the sequential addition of new tiles. Confirming the physical plausibility and relevance of the abstraction, simple self-assembling systems of tiles have been built out of certain types of DNA molecules [16, 11, 10, 8]. The possibility of using self-assembly for nanofabrication of complex components such as circuits has been suggested as a promising application [5].

The view that the "shape" of a self-assembled complex can be considered the output of a computational process [2] has inspired recent interest [7, 1, 3, 6, 4].

⋆ A preprint of the full paper can be found at `http://arxiv.org`.

While it was shown through specific examples that self-assembly could be used to construct interesting shapes and patterns, it was not known in general which shapes could be self-assembled from a small number of tile types. Understanding the complexity of shapes is facilitated by an appropriate definition of shape. In our model, a tile system generates a particular shape if it produces any scaled version of that shape (Sect. 3). This definition may be thought to formalize the idea that a structure can be made up of arbitrarily small pieces. Computationally, it is analogous to disregarding computation time and is thus more appropriate as a notion of output of a *universal* computation process.[1] Using this definition of shape, we show that for any shape $\tilde{S}$, if $K_{sa}(\tilde{S})$ is the minimal number of distinct tile types necessary to self-assemble it then $K_{sa}(\tilde{S}) \log K_{sa}(\tilde{S})$ is within multiplicative and additive constants (independent of $\tilde{S}$) of the shape's Kolmogorov complexity. This theorem is proved by construction (which might be of independent interest) of a general method for converting a program that outputs a fixed size shape as a list of locations into a tile system that self-assembles a scaled version of the shape. Our result ties the computation of a shape and its self-assembly, and, somewhat counter-intuitively, implies that scaling up a shape may often allow it to be self-assembled from fewer tile types. Another consequence of the theorem is that the minimal number of tile types necessary to self-assemble an arbitrary scaling of a shape is uncomputable. Answering the same question about shapes of a fixed size is computable but NP complete [1].

2 The Tile Assembly Model

We present a description of the Tile Assembly Model based on Rothemund and Winfree [7] and Rothemund [6]. We will be working on a $\mathbb{Z} \times \mathbb{Z}$ grid of unit square locations. The **directions** $\mathcal{D} = \{N, E, W, S\}$ are used to indicate relative positions in the grid. Formally, they are functions $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \times \mathbb{Z}$: $N(i,j) = (i, j+1)$, $E(i,j) = (i+1, j)$, $S(i,j) = (i, j-1)$, and $W(i,j) = (i-1, j)$. The inverse directions are defined naturally: $N^{-1}(i,j) = S(i,j)$, etc. Let Σ be a set of **bond types**. A **tile type** t is a 4-tuple $(\sigma_N, \sigma_E, \sigma_S, \sigma_W) \in \Sigma^4$ indicating the associated bond types on the north, east, south, and west sides. Note that tile types are oriented and a rotated version of a tile type is considered to be a different tile type. A special bond type *null* represents the lack of an interaction and the special tile type $empty = (null, null, null, null)$ represents an empty space. If T is a set of tile types, a **tile** is a pair $(t, (i,j)) \in T \times \mathbb{Z}^2$ indicating that location (i,j) contains the tile type t. Given the tile $t = (t, (i,j))$, $type(t) = t$ and $pos(t) = (i,j)$. Further, $bond_D(t)$, where $D \in \mathcal{D}$, is the bond type of the respective side of t, and $bond_D(t) = bond_D(type(t))$. A **configuration** is a set of tiles such that there is exactly one tile in every location $(i,j) \in \mathbb{Z} \times \mathbb{Z}$. For

---

[1] The production of a shape of a fixed size cannot be considered the output of a universal computation process: whereas questions about the result of universal computation are often uncomputable, any question about a shape of a fixed-size can be answered with a finite simulation of the self-assembly process [7], because in the model considered here, once a tile is added, it cannot be removed. Thus questions about shapes of fixed size are decidable.

any configuration A, we write $A(i, j)$ to indicate the tile at location (i, j). We will write a configuration as a set of non-empty tiles; all other tiles are implicitly *empty*.

A **strength function** $g : \Sigma \times \Sigma \to \mathbb{Z}$, where $null \in \Sigma$, defines the interactions between adjacent tiles: we say that a tile t_1 interacts with its neighbor t_2 with strength $\Gamma(t_1, t_2) = g(\sigma, \sigma')$ where σ is the bond type of tile t_1 that is adjacent to the bond type σ' of tile t_2.[2] In most previous works on self-assembly, as in this work, strength functions are restricted with the following properties: (1) g is symmetric (the effect that one tile has an another is equal to the effect that the other has on the first), (2) $g(\sigma, null) = 0$ (the lack of an interaction is normalized to zero), (3) g is *non-negative* (there are no "adverse" interactions counteracting other interactions), (4) g is *diagonal*: $g(\sigma, \sigma') = 0$ if $\sigma \neq \sigma'$ (only sides with matching bond types interact). Property 4 shows the roots of the Tile Assembly Model in the Wang tiling model. Our results confirm that non-negativity is not a major restriction as Theorem 1 is valid for strength functions taking on negative values. However, if property 4 is relaxed then information in tile systems can be represented more compactly (using fewer tile types [4]), potentially leading to a different form of Theorem 1.

A **tile system T** is a quadruple (T, t_s, g, τ) where T is a finite set of non-empty tile types, t_s is a special **seed tile** with $type(t_s) \in T$, g is a strength function, and τ is the threshold parameter. Self-assembly is defined by a relation between configurations. Suppose A and B are two configurations, and t is a tile such that $A = B$ except at $pos(t)$ and $A(pos(t)) = null$ but $B(pos(t)) = t$. Then we write $A \to_\mathbf{T} B$ if $\sum_{D \in \mathcal{D}} \Gamma(t, A(D(pos(t)))) \geq \tau$. This means that a tile can be added to a configuration iff the sum of its interaction strengths with its neighbors reaches τ. The relation $\to_\mathbf{T}^*$ is the reflexive transitive closure of $\to_\mathbf{T}$.

Whereas a configuration can be any arrangement of tiles (not necessarily connected), we are interested in the subclass of configurations that can result from a self-assembly process. Formally, the tile system and the relation $\to_\mathbf{T}^*$ define the partially ordered set of **assemblies**: $Prod(\mathbf{T}) = \{A \text{ s.t. } \{t_s\} \to_\mathbf{T}^* A\}$, and the set of **terminal assemblies**: $Term(\mathbf{T}) = \{A \in Prod(\mathbf{T}) \text{ and } \nexists B \neq A \text{ s.t. } A \to_\mathbf{T}^* B\}$. A tile system **T uniquely produces** A if $\forall B \in Prod(\mathbf{T})$, $B \to_\mathbf{T}^* A$ (which implies $Term(\mathbf{T}) = \{A\}$).

The tile systems used in our constructions have $\tau = 2$ with the strength function ranging over $\{0, 1, 2\}$. It is known that $\tau = 1$ systems with strength function ranging over $\{0, 1\}$ are rather limited [7, 6].

3 Arbitrarily Scaled Shapes and Their Complexity

In this section, we introduce the model for the output of the self-assembly process used in this paper. Let S be a finite set of locations on $\mathbb{Z} \times \mathbb{Z}$. The adjacency graph

---

[2] More formally,

$$\Gamma(t_1, t_2) = \begin{cases} g(bond_{D^{-1}}(t_1), bond_D(t_2)) \text{ if } \exists D \in \mathcal{D} \text{ s.t. } pos(t_1) = D(pos(t_2)); \\ 0 \text{ otherwise.} \end{cases}$$

$G(S)$ is the graph on S defined by the adjacency relation where two locations are considered adjacent if they are directly north/south, or east/west of one another. We say that S is a **coordinated shape** if $G(S)$ is connected.[3] The **coordinated shape of assembly** A is the set $S_A = \{(i,j) \text{ s.t. } A(i,j) \neq empty\}$. Note that S_A is a coordinated shape because A contains a single connected component.

For any set of locations S, and any $c \in \mathbb{Z}^+$, we define a c**-scaling of S** as

$$S^c = \{(i,j) \text{ s.t. } (\lfloor i/c \rfloor, \lfloor j/c \rfloor) \in S\}.$$

Geometrically, this represents a "magnification" of S by a factor c. Note that a scaling of a coordinated shape is itself a coordinated shape: every node of $G(S)$ gets mapped to a c^2-node connected subgraph of $G(S^c)$ and the relative connectivity of the subgraphs is the same as the connectivity of the nodes of $G(S)$. A parallel argument shows that if S^c is a coordinated shape, then so is S. We say that coordinated shapes S_1 and S_2 are **scale-equivalent** if $S_1^c = S_2^d$ for some $c, d \in \mathbb{Z}^+$. Two coordinated shapes are **translation-equivalent** if they can be made identical by translation. We write $S_1 \cong S_2$ if S_1^c is translation-equivalent to S_2^d for some $c, d \in \mathbb{Z}^+$. Scale-equivalence, translation-equivalence and $\cong$ are equivalence relations. We call $\tilde{S}$, the equivalence class of coordinated shapes under "$\cong$", the **shape** $\tilde{S}$. We say that $\tilde{S}$ is the **shape of assembly** A if $S_A \in \tilde{S}$. The view of computation performed by the self-assembly process espoused here is the production of a shape as the "output" of the self-assembly process. Intuitively, the idea of scale-invariance attempts to formalize the notion that a physical object can be constructed from arbitrarily small pieces.

Having defined the notion of shapes, we turn to their descriptional complexity. As usual, the Kolmogorov complexity of a binary string x with respect to a universal Turing machine U is $K_U(x) = \min\{|p| \text{ s.t. } U(p) = x\}$. (See the exposition of Li and Vitanyi [9] for an in-depth discussion of Kolmogorov complexity.) Let us fix some "standard" universal machine U. We call the Kolmogorov complexity of a coordinated shape S to be the size of the smallest program outputting it as a list of locations:[4,5]

$$K(S) = \min\{|s| \text{ s.t. } U(s) = \langle S \rangle\}.$$

The Kolmogorov complexity of a shape $\tilde{S}$ is:

$$K(\tilde{S}) = \min\left\{|s| \text{ s.t. } U(s) = \langle S \rangle \text{ for some } S \in \tilde{S}\right\}.$$

---

[3] We say "coordinated" to make explicit that a fixed coordinate system is used. We reserve the unqualified term "shape" for when we ignore scale and translation.

[4] Note that $K(S)$ is within an additive constant of $K(x)$ where x is some other effective description of S, such as a characteristic function. Since our results are asymptotic, they are independent of the specific representation choice. One might also consider invoking a two dimensional computing machine, but it is not fundamentally different for the same reason.

[5] Notation $\langle \cdot \rangle$ indicates some standard binary encoding of the object(s) in the brackets. In the case of coordinated shapes, it means an explicit binary encoding of the set of locations. Integers, tuples or other data structures are similarly given simple explicit encodings.

We define the **tile-complexity** of a coordinated shape S and shape $\tilde{S}$ respectively as:

$$K_{sa}(S) = \min \left\{ \begin{array}{l} n \text{ s.t. } \exists \text{ a tile system } \mathbf{T} \text{ of } n \text{ tile types that uniquely produces} \\ \text{assembly } A \text{ and } S \text{ is the coordinated shape of } A \end{array} \right\}$$

$$K_{sa}(\tilde{S}) = \min \left\{ \begin{array}{l} n \text{ s.t. } \exists \text{ a tile system } \mathbf{T} \text{ of } n \text{ tile types that uniquely produces} \\ \text{assembly } A \text{ and } \tilde{S} \text{ is the shape of } A \end{array} \right\}.$$

4 Relating Tile-Complexity and Kolmogorov Complexity

The essential result of this paper is the description of the relationship between the Kolmogorov complexity of any scale-invariant shape and the number of tile types necessary to self-assemble it.

Theorem 1. *There exist constants a_0, b_0, a_1, b_1 such that for any shape $\tilde{S}$,*

$$a_0 K(\tilde{S}) + b_0 \leq K_{sa}(\tilde{S}) \log K_{sa}(\tilde{S}) \leq a_1 K(\tilde{S}) + b_1. \tag{1}$$

Note that since any tile system of n tile types can be described by $O(n \log n)$ bits, the theorem implies there is a way to construct a tiling system such that asymptotically at least a constant fraction of these bits is used to "describe" the shape rather than any other aspect of the tiling system.

Proof (of Theorem 1). To see that $a_0 K(\tilde{S}) + b_0 \leq K_{sa}(\tilde{S}) \log K_{sa}(\tilde{S})$, realize that there exists a constant size program p_{sa} that, given a binary description of a tile system, simulates its self-assembly, making arbitrary choices where multiple tile additions are possible. If the self-assembly process terminates, p_{sa} outputs the coordinated shape of the terminal assembly as the binary encoding of the list of locations in it. Any tile system $\mathbf{T}$ of n tile types with any diagonal strength function and any threshold τ can be represented by a string $d_{\mathbf{T}}$ of $4n\lceil \log 4n \rceil + 16n$ bits: For each tile type, the first of which is assumed to be the seed, specify the bond types on its four sides. There are no more than $4n$ bond types. In addition, for each tile type t specify for which of the 16 subsets $L \subseteq \mathcal{D}$, $\sum_{D \in L} g(bond_D(t)) \geq \tau$. Note that the possibility of negative bond strengths is taken into account, but a diagonal strength function is assumed. If $\mathbf{T}$ is a tile system uniquely producing an assembly that has shape $\tilde{S}$, then $K(\tilde{S}) \leq |p_{sa} d_{\mathbf{T}}|$. The left inequality in eq. 1 follows with the multiplicative constant $a_0 = 1/4 - \varepsilon$ for arbitrary $\varepsilon > 0$.

We prove the right inequality in eq. 1 by developing a construction (sketched in Section 5 and detailed in the full paper) showing how for any program s s.t. $U(s) = \langle S \rangle$, we can build a tile system $\mathbf{T}$ of $64\lceil \frac{|p|}{\log |p|} \rceil + b$ tile types, where b is a constant and p is a string consisting of a fixed program p_{sb} and s (i.e. $|p| = |p_{sb}| + |s|$), that uniquely produces an assembly whose shape is $\tilde{S}$ such that $S \in \tilde{S}$. Program p_{sb} and constant b are both independent of S. The right inequality in eq. 1 follows with the multiplicative constant $a_1 = 64 + \varepsilon$ for arbitrary $\varepsilon > 0$. $\square$
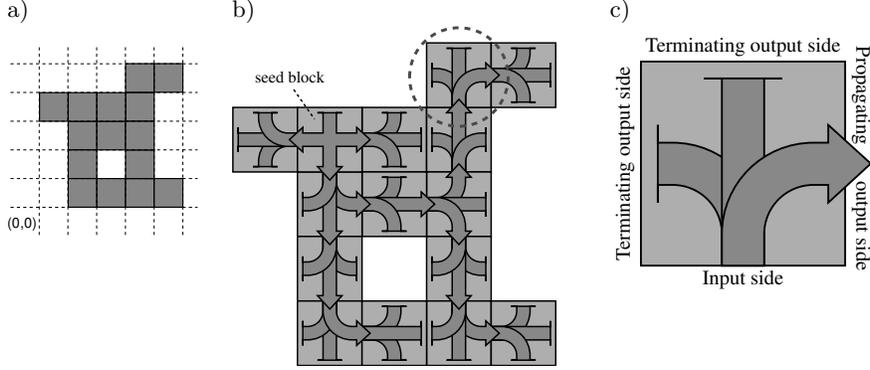
Fig. 1. Forming a shape out of blocks: a) A coordinated shape S. b) An assembly composed of c by c blocks that grow according to transmitted instructions such that the shape of the final assembly is $\tilde{S}$. Arrows indicate information flow and order of assembly. (Not drawn to scale.) The seed block and the circled growth block are schematically expanded in Fig. 2. c) The nomenclature describing the types of block sides.

Our result can be used to show that the tile-complexity of shapes is uncomputable:

Corollary 1. K_{sa} *of shapes is uncomputable. In other words, the following language is undecidable:* $\tilde{L} = \left\{ (l, n) \text{ s.t. } l = \langle S \rangle \text{ for some } S \in \tilde{S} \text{ and } K_{sa}(\tilde{S}) \leq n \right\}$.

Language $\tilde{L}$ should be contrasted with $L = \{(l, n) \text{ s.t. } l = \langle S \rangle \text{ and } K_{sa}(S) \leq n\}$ which is decidable (but hard to compute in the sense of NP-completeness [1]).

Proof (of Corollary 1). We essentially parallel the proof that Kolmogorov complexity is uncomputable. If $\tilde{L}$ were decidable, then we can make a program that computes $K_{sa}(\tilde{S})$ and subsequently uses Theorem 1 to compute an effective lower bound for $K(\tilde{S})$. Then we can construct a program p that given n outputs some coordinated shape S (as a list of locations) such that $K(\tilde{S}) \geq n$ by enumerating shapes and testing with the lower bound, which we know must eventually exceed n. But this results in a contradiction since $p\langle n \rangle$ is a program outputting $S \in \tilde{S}$ and so $K(\tilde{S}) \leq |p| + \lceil \log n \rceil$. But for large enough n, $|p| + \lceil \log n \rceil < n$. $\qquad \square$

5   Sketch of the Programmable Block Construction

5.1   Overview

The uniquely produced terminal assembly A of our tile system will consist of square "blocks" of c by c tiles. There will be one block for each location in S. Consider the coordinated shape in Fig. 1(a). An example assembly A is graphically represented in Fig. 1(b), where each square represents a block containing c^2 tiles. Self-assembly initiates in the *seed block*, which contains the seed tile, and proceeds according to the arrows illustrated between blocks. Thus if there

is an arrow from one block to another, it indicates that the growth of the second block (a *growth block*) is initiated from the first. A terminated arrow indicates that the block does not initiate the self-assembly of an adjacent block in that direction. Fig. 1(c) describes our nomenclature: an arrow comes into a block on its input side, arrows exit on propagating output sides, and terminated arrows indicate terminating output sides. The seed block has four output sides, which can be either propagating or terminating. Each growth block has one input and three output sides, which are also either propagating or terminating.

The input/output connections of the blocks form a spanning tree rooted at the seed block. During the progress of the self-assembly of the seed block, a computational process determines the input/output relationships of the rest of the blocks in the assembly. This information is propagated from block to block (along the arrows in Fig. 1(b)) during self-assembly and describes the shape of the assembly. By following the instructions each growth block receives in its input, the block decides where to start the growth of the next block and what information to pass to it in turn. The scaling factor c is set by the size of the seed block. The computation in the seed block ensures that c is large enough that there is enough space to do the necessary computation within the other blocks.

We present a sketch of a general construction that represents a Turing–universal way of guiding large scale self-assembly of blocks based on an input program p. In the following section, we describe the architecture of seed and growth blocks on which arbitrary programs can be executed. In section 5.3 we discuss the programming of p that is required to grow the blocks in the form of a specific shape. For a complete presentation of our construction, including the proof that the terminal assembly is uniquely produced, see the full version of this paper.

5.2   Architecture of the Blocks

The internal structure of a growth block is shown in Fig. 2(a). The first part is a Turing machine simulation, which is based on [12, 7]. The machine simulated is a universal Turing machine that takes its input from the propagating output side of the previous block. This TM has an output alphabet $\{0, 1, S\}^3$ and an input alphabet $\{(000), (111)\}$ on a two-way tape. The output of the simulation, as 3-tuples, is propagated until the diagonal. The diagonal propagates each member of the 3-tuples crossing it to one of the three output sides, like a prism separating the colors of the spectrum. This allows the single TM simulation to produce three separate strings targeted for the three output sides. The "S" symbol in the output of the TM simulation is propagated like the other symbols. However, it acts in a special way when it crosses the boundary tiles at the three output sides of the block, where it starts a new block. The output sides that receive the "S" symbol become propagating output sides and the output sides that do not receive it become terminating output sides. Obviously, the TM simulation decides which among the three output sides will become propagating output sides, and what information they should contain, by outputting appropriate
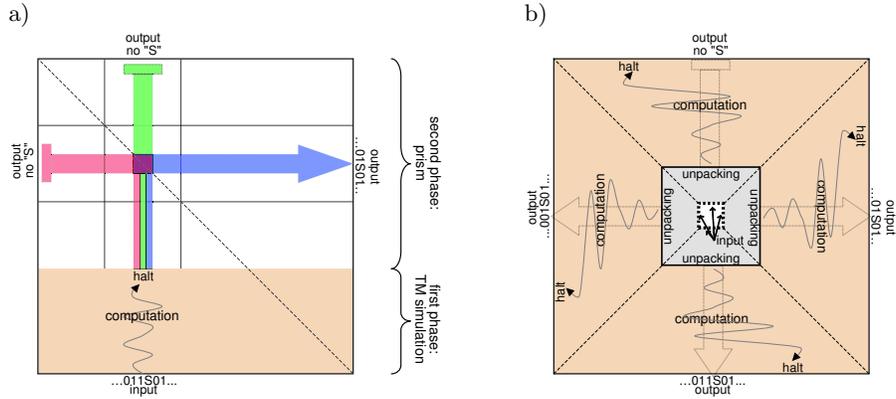
Fig. 2. Internal structure of a growth block (a) and seed block (b).

tuples. Subsequent blocks will use this information as a program, as discussed in section 5.3.

An outline of the seed block is shown in Fig. 2(b). While growth blocks contain a single TM simulation that outputs a different string to each of the three output sides, the seed block contains four *identical* TM simulations that output different strings to each of the four output sides. This is possible because the border tile types transmit information selectively: the computation in the seed block is performed using 4-tuples as the alphabet in a manner similar to the growth blocks, but only one of the elements of the 4-tuple traverses the border of the seed block. As with growth blocks, if the transmitted symbol is "S", the outside edge initiates the assembly of the adjoining block. The point of having four identical TM simulations is to ensures that the seed block is square: while a growth block uses the length of its input side to set the length of its output sides (via the diagonal), the seed block does not have any input sides. (Remember that it is the seed block that sets the size of all the blocks.)

The initiation of the Turing machine simulations in the seed block is done by tile types encoding the program p that guides the block construction. The natural approach to provide this input is using 4 rows (one for each TM) of unique tiles encoding one bit per tile; however, this does not result in an asymptotically optimal encoding. We follow Adleman et al [3] and encode on the order of $\log n / \log \log n$ bits per tile where n is the length of the input. This representation is then unpacked into a one-bit-per-tile representation used by the TM simulation (Fig. 2(b)). Adleman et al's method requires $O(n / \log n)$ tiles to encode n bits, leading to the asymptotically optimal result of Theorem 1.

5.3 Programming Blocks and the Value of the Scaling Factor c

In order for our tile system to produce some assembly whose shape is $\tilde{S}$, instructions encoded in p must guide the construction of the blocks by deciding on which side of which block a new block begins to grow and what is encoded on

the edge of each block. For our purposes, we take $p = p_{sb}\langle s \rangle$ (i.e. p_{sb} takes s as input), where s is a program that outputs the list of locations in the shape S. p_{sb} runs s to obtain this list and plans out a spanning tree t over these locations (it can just do a depth-first search) starting from some arbitrarily chosen location that will correspond to the seed block. The information passed along the arrows in Fig. 1(b) is $p_{gb}\langle t, (i,j) \rangle$ which is the concatenation of a program p_{gb} to be executed within each growth block, and an encoding of the tree t and the location (i,j) of the block into which the arrow is heading. When executed, $p_{gb}\langle t, (i,j) \rangle$ evaluates to a 3-tuple encoding of $p_{gb}\langle t, D(i,j) \rangle$ together with symbol "S" for each propagating output side D. Thus, each growth block passes $p_{gb}\langle t, D(i,j) \rangle$ to its Dth propagating output side as directed by t. Note that program p_{sb} in the seed tile must also run long enough to ensure that c is large enough that the computation in the growth blocks has enough space to finish without running into the sides of the block or into the diagonal. Nevertheless, the scaling factor c is dominated by the building of t in the seed block, as the computation in the growth blocks takes only $poly(|S|)$. Since the building of t is dominated by the running time of s, we have $c = poly(time(s))$.

6 Discussion and Open Questions

Because the Kolmogorov complexity of a string depends on the universal Turing machine chosen, the complexity community adopted a notion of additive equivalence, where additive constants are ignored. However, Theorem 1 includes multiplicative constants as well, which are not customarily discounted. It might be possible to use a more clever method of unpacking (Sect. 5.2) and a seed block construction that reduces the multiplicative constant a_1 of Theorem 1. Correspondingly, there might be a more efficient way to encode any tile system than described in the proof of the theorem, and thereby increase a_0.

It is most likely that our block construction can be easily adapted to use a different encoding of the input, leading to a different form of Theorem 1 for variations on the Tile Assembly Model. Recent work by Aggarwal et al [4] shows that allowing non-diagonal strength functions allows a dramatic change in tile complexity.

The programmable block construction may have other applications as well. For instance, it is easy to reprogram it to simulate, using few tile types, a large deterministic $\tau = 1$ tile system for which a short algorithmic description of the tile set exists. We believe a slightly extended version of the block construction can also be used to provide compact tile sets that simulate other $\tau = 2$ tile systems that have short algorithmic descriptions.

The scaling factor $c = poly(time(s))$ is extremely large since $|S|$ is presumably enormous for cases where our results are of interest and s must output every location in S. Are there special instances where it is not necessary to run the program s outputting S to completion but query it in a few locations relevant to the immediate block being built?

References

1. L. Adleman, Q. Cheng, A. Goel, M.-D. Huang, D. Kempe, P. M. de Espanes, and P. W. K. Rothemund. Combinatorial optimization problems in self-assembly. In *Proc. of STOC*, 2002.

2. L. M. Adleman. Toward a mathematical theory of self-assembly (extended abstract). Technical report, University of Southern California, 1999.

3. L. M. Adleman, Q. Cheng, A. Goel, and M.-D. A. Huang. Running time and program size for self-assembled squares. In *ACM Symposium on Theory of Computing*, pages 740–748, 2001.

4. G. Aggarwal, M. Goldwasser, M. Kao, and R. T. Schweller. Complexities for generalized models of self-assembly. In *Symposium on Discrete Algorithms*, 2004.

5. M. Cook, P. W. K. Rothemund, and E. Winfree. Self-assembled circuit patterns. In *DNA Based Computers 9*, pages 91–107, 2004.

6. P. W. K. Rothemund. *Theory and Experiments in Algorithmic Self-Assembly*. PhD thesis, University of Southern California, Los Angeles, 2001.

7. P. W. K. Rothemund and E. Winfree. The program-size complexity of self-assembled squares (extended abstract). In *ACM Symposium on Theory of Computing*, pages 459–468, 2000.

8. T. H. LaBean, H. Yan, J. Kopatsch, F. Liu, E. Winfree, J. H. Reif, and N. C. Seeman. Construction, analysis, ligation, and self-assembly of DNA triple crossover complexes. *Journal of the Americal Chemical Society*, 122:1848–1860, 2000.

9. M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, second edition, 1997.

10. C. Mao, T. H. LaBean, J. H. Reif, and N. C. Seeman. Logical computation using algorithmic self-assembly of DNA triple-crossover molecules. *Nature*, 407:493–496, 2000.

11. C. Mao, W. Sun, and N. C. Seeman. Designed two-dimensional DNA holliday junction arrays visualized by atomic force microscopy. *Journal of the Americal Chemical Society*, 121:5437–5443, 1999.

12. R. M. Robinson. Undecidability and nonperiodicity of tilings of the plane. *Inventiones Mathematicae*, 12:177–209, 1971.

13. H. Wang. Proving theorems by pattern recognition. II. *Bell Systems Technical Journal*, 40:1–42, 1961.

14. E. Winfree. Simulations of computing by self-assembly, Caltech CS TR 1998.22.

15. E. Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, Pasadena, 1998.

16. E. Winfree, F. Liu, L. A. Wenzler, and N. C. Seeman. Design and self-assembly of two dimensional DNA crystals. *Nature*, 394:539–544, 1998.