One-Dimensional Staged Self-Assembly

Erik D. Demaine[1], Sarah Eisenstat[1],
Mashhood Ishaque[2], and Andrew Winslow[2]

[1] MIT Computer Science and Artificial Intelligence Laboratory,
Cambridge, MA 02139, USA, {edemaine,seisenst}@mit.edu
[2] Department of Computer Science, Tufts University,
Medford, MA 02155, USA, {mishaque,awinslow}@cs.tufts.edu

Abstract. We introduce the problem of staged self-assembly of *one-dimensional* nanostructures, which becomes interesting when the elements are labeled (e.g., representing functional units that must be placed at specific locations). In a restricted model in which each operation has a single terminal assembly, we prove that assembling a given string of labels with the fewest steps is equivalent, up to constant factors, to compressing the string to be uniquely derived from the smallest possible context-free grammar (a well-studied $O(\log n)$-approximable problem) and that the problem is NP-hard. Without this restriction, we show that the optimal assembly can be substantially smaller than the optimal context-free grammar, by a factor of $\Omega(\sqrt{n/\log n})$ even for binary strings of length n. Fortunately, we can bound this separation in model power by a quadratic function in the number of distinct glues or tiles allowed in the assembly, which is typically small in practice.

Keywords: context-free grammar, Wang tile, DNA computing, complexity

1 Introduction

Self-assembly is the study of how small particles (typically at the nanoscale, where electrostatic forces overwhelm gravity) interact with each other to conglomerate into larger objects. In theoretical computer science, the standard model is the *tile assembly model* [14] in which the system begins with infinitely many copies of certain square tiles, each with specified glues on the four sides, and tiles translate nondeterministically in the plane until they attach to each other at matching glues. This model effectively enables performing computation, but out of simple geometric parts, and at the cost of physical space resulting from the assembly.

The most studied problem in the tile assembly model is to determine the number of distinct tile types required to assemble a given shape (made out of unit squares). An obvious upper bound is the area of the shape, but in many cases fewer tiles suffice, by building computation into the construction. For example, an $n \times n$ square requires $\Theta(\log n/\log\log n)$ distinct tile types (and glues), by embedding a base-$\log n$ counter, while an $n \times 1$ rectangle requires $\Theta(n)$ tile types. The most general result is that any shape, scaled by a sufficiently large factor, can be constructed from $O(K/\log K)$ tile types (and glues), where K is the Kolmogorov complexity of the shape [13]. Unfortunately, the scale factor is polynomial in the running time of the Turing machine generating the shape, which is at least the area of the shape. So this result does not directly address the tile complexity of a specific shape, though it suggests that it is difficult to characterize.

An alternate approach is offered by *staged self-assembly* [4] in which the system's tile set can change in a sequence of stages, in particular by an experimenter mixing two systems together. In this model, it is possible to make any shape using a constant number of tile types (and glues); as a result, the main objectives are to minimize both the number of mix operations (work for the experimenter) and the number of stages that must be executed sequentially (makespan or wait time). For example, both an $n \times n$ square and an $n \times 1$ rectangle can be assembled using $O(1)$ tiles and glues and $O(\log n)$ mixes and stages. This level of efficiency using a constant number of glues is only matched by self-assembly models incorporating concentration (e.g. the work of Kao and Schweller [7] and the PTAM model of Chandran et al. [1]) and assemble a given shape with high probability.

Our personal communication with bioengineers suggests that the staged assembly model is natural and practical, essentially exposing the experimenter's ability to perform actions as part of a computation/assembly. Furthermore, the results are more practical, as it is difficult in practice to design many different glues that attract only in pairs, without any attraction between unpaired glues. Assembling a 1000×1 rectangle would be impractical without staging (requiring 1000 tile types and 999 distinct glues), but is extremely practical with staging (requiring only 6 tile types, 3 distinct glues and 10 stages).

In this paper, we aim to characterize the resources required to staged-assemble a *one-dimensional* object. Just making a $1 \times n$ rectangular shape is trivial, so this direction has so far been overlooked. But in practice, experimenters often want to build an object that not only takes on a desired shape but also carries out a desired functionality. A typical example is to arrange nanodots or bioagents in a particular pattern within a shape.[3] We model this problem as constructing a *labeled* shape, where each unit square has a label within a fixed alphabet, and each tile type used to build the shape also has a label, which must match in construction. Thus the input to the problem is a string of labels, and the goal is to find a staged assembly with few glues, mixes, and stages. In fact we show that four glues and $O(\log n)$ stages always suffice, so a single objective remains: minimize the mixes.

The problem of computing a minimal tile assembly system that produces a labeled shape has been studied previously. Heuristic approaches have been developed to find the smallest tile set that uniquely assembles an input labeled shape [10,6], and the problem of finding a minimum-size tile set for a labeled square using a restricted form of self-assembly has been shown to be NP-hard [3].

[3] Personal communication with Hyunmin Yi, 2008–2010.

We successfully characterize the number of mixes required to staged-assemble a string in two natural situations. In the first setting (Section 4), we restrict mixing operations to produce a single terminal assembly for use in the next mix. This restriction seems to be common to all previous staged-assembly algorithms [4], but we do not know it to be practically motivated. If the number of glues is constant, we show that finding the minimum number of mixings in this setting is NP-hard. In the second setting (Section 5.5), we allow multiple "parallel assemblies" resulting from a mix (a new though natural idea), and consider the more natural restriction that the number of glues is constant. In both settings, we show that the minimum number of mixes is within a constant factor of the smallest context-free grammar that generates exactly the given string. The latter problem is well-studied, has a polynomial-time $O(\log n)$-approximation algorithm based on Lempel-Ziv compression, and likely has no $o(\log \log n)$-approximation [2,9,11,12].

We show that our relations are nearly tight by constructing a family of strings (Section 5) showing a separation in power between (unrestricted) staged assembly and context-free grammars. Specifically, an n-bit binary string can be assembled using $O(k)$ glues with $O(k)$ mixes but requires a context-free grammar of size $\Omega(k^2)$, for a ratio of $\Omega(k)$. Our upper bound shows that the worst-case separation is $O(k^2)$. As a function of n (with an unbounded number of glues), we prove that the ratio is $\Omega(\sqrt{n/\log n})$. In practice, small feature sizes make the number of glues typically small, in which case context-free grammars are actually a good approximation to optimal staged self-assemblies.

The labeled 1D staged self-assembly model offers a balance of tractability, being easier than general staged assembly by reducing the dimension to 1, yet harder (and more practical) by adding labels to the target shape. The connections we show to context-free-grammar compression illustrate that the problem is difficult, yet for the case of many glues, still not fully understood. The approximation algorithm resulting from our study is simple and efficient, having been implemented in an online web system described in Section 6, which is currently being considered for practical use by the Tufts Department of Bioengineering, in a setting where labeled 1D assemblies are of significant interest.

2 Context-Free Grammars

Definition 1. *A context-free grammar (CFG) is defined as a 4-tuple* $(\Sigma, \Gamma, S, \Delta)$ *where* Σ *is a set of* terminal symbols*,* Γ *is a set of* non-terminal symbols*,* S *is a special element of* Γ *called the* start symbol *and* Δ *is a set of* productions*.*

Each production consists of a *left-hand side* containing a single non-terminal symbol, and a *right-hand side* containing a (non-empty) sequence of terminal and non-terminal symbols. A CFG *derives* a string by repeated replacement of non-terminal symbols with strings of terminal and non-terminal symbols according to the productions in Δ, beginning with the single symbol S. The *language* of a CFG is the set of derivable strings consisting solely of terminal symbols.

Definition 2. *The size of a context-free grammar G, denoted $|G|$, is the total number of symbols appearing on the right-hand sides of the productions in G.*

Note that this definition counts the *total* number of symbols, so symbols appearing multiple times contribute to the count multiple times. In this paper we consider only CFGs that are *deterministic* (with only one production per left-hand side) and generate exactly one string.

Definition 3. *A* restricted context-free grammar *(written RCFG) is a CFG which is deterministic and has a language consisting of a single string.*

For an RCFG G deriving a string s, the *parse tree* of G is the tree created by beginning with a single node with label S (the start symbol), and adding children to a leaf node in a left-to-right order for each production applied. The result is a tree where each internal node is a non-terminal symbol, each leaf node is a terminal symbol, and a left-to-right traversal of the leaves gives the string s.

Definition 4. *The* smallest grammar problem *is the following: given an input string s, find the smallest RCFG deriving s.*

Any RCFG G has exactly one parse tree. Each internal node in the parse tree has a corresponding non-terminal symbol from G. If we merge all such nodes with the same non-terminal, the result is a *directed acyclic graph* (DAG) called the *parse DAG*. See Figure 1 for an example of an RCFG and its parse tree and parse DAG.

$$A \rightarrow BC$$
$$B \rightarrow DC$$
$$C \rightarrow aa$$
$$D \rightarrow cc$$

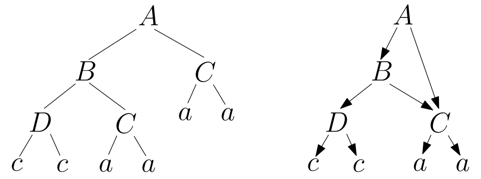


Fig. 1: A restricted context-free grammar (RCFG) and its corresponding parse tree and parse DAG.

3 Staged Self-assembly

In this section we describe the 1D labeled staged self-assembly system model. The model described here is a variant of the staged self-assembly model defined in [4]. In this model, individual building blocks are 2D square-shaped *tiles* that translate in the plane. Each tile has four sides (north, south, east, and west) and has *glues* on its east and west sides. Each tile also has a label $(a, b, c, \ldots)$. We denote a tile $x_1[x_2]x_3$ where x_1 is the west glue, x_2 is the label, x_3 is the east glue (e.g., $1[a]2$).

Tiles combine when the pair of glues on the west side of one tile and east side of the other are *complementary.* We denote glue values by numbers (e.g., $1, 2, \ldots$), and the complementary glues are denoted $1', 2', \ldots$. In this paper we use the convention that east glues are always complementary (prime) glues. So a tile $1[a]2$ actually has east glue $2'$.

When tiles combine they create *assemblies* (and we consider tiles to be a special case of assemblies). The labels of each individual tile combine to form a *label string* of the assembly consisting of the labels of the combined tiles in order. For example, $1[a]2$ and $2[b]3$ combine to form the assembly $1[ab]3$. Note that the east glue of $1[a]2$ and west glue of $2[b]3$ have disappeared: they are on the interior of the assembly and are omitted for clarity. Assemblies can also be combined to form larger assemblies. The size of an assembly is the number of tiles it contains.

Initially each tile type exists in a separate *bin.* When bins are mixed, the assemblies present in each bin are free to attach to each other. The products of each mixing are *terminal assemblies*: assemblies that do not attach to any other assemblies, and we refer to the terminal assemblies of a mixing as the bin's contents. All other assemblies produced during the mixing are assumed to be filtered out before the bin is combined with other bins. Any bin may be used in as many mixings as desired.

A self-assembly system instance is defined by the starting tiles and a mix DAG defining bins and the orders in which they are mixed (see Figure 2). The mix DAG is a *rooted DAG*: a DAG with only one node (the *root*) without in-edges. where each node represents a bin and the edges leaving it point to the bins whose contents are mixed into this bin. Each leaf of the DAG is a bin of a single tile type.

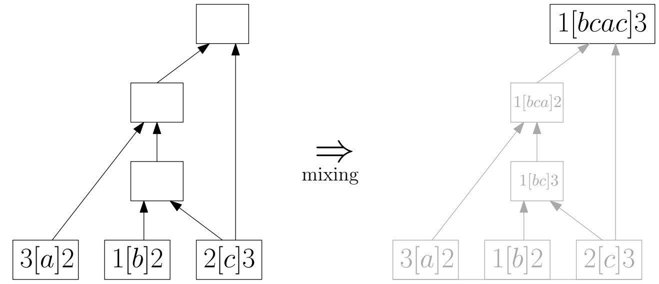


Fig. 2: A self-assembly system (SAS) consisting of its mix DAG and initial bins (left), and generated assembly (right).

Definition 5. *A self-assembly system (SAS) is a one-dimensional labeled staged self-assembly instance that produces a single goal assembly and is defined by a mix DAG and a unique tile type for each leaf of the DAG.*

Definition 6. *The size of a SAS A (denoted $|A|$) is the number of edges in its mix DAG.*

The goal assembly produced by a SAS must appear in the bin corresponding to the root node of the mix DAG. Reading the labels of an assembly from west to east defines a string which we call the *label string* of the assembly. The label string of the goal assembly is the string *generated* by the SAS.

In previous staged-assembly constructions [4], each bin has a single assembly produced in it by mixing the contents of two other bins (which also contain single items). However the model as defined does not require that each bin contains a single assembly. A mix DAG in which one or more bins has multiple assemblies is said to use *bin parallelism.* We distinguish a self-assembly system instance that does not use bin parallelism as a *single self-assembly system* (SSAS).

Definition 7. *A single self-assembly system (SSAS) is a SAS in which no bin contains more than one distinct assembly.*

Definition 8. *The* minimum SSAS problem *is the following: given an input string s, find a smallest SSAS generating an assembly with label string s.*

4 Equivalence between RCFGs and SSASs

In this section we show that converting between an RCFG G deriving a string s and a SSAS instance A assembling a labeled assembly with label s is possible with only a constant-factor scaling. As a result, any algorithm generating an $O(f(n))$-approximation to *either* the minimum grammar problem or the minimum SSAS problem implies an $O(f(n))$-approximation algorithm for the other.

4.1 Converting RCFGs to SSASs

Let G be an RCFG deriving a string s. We begin by converting G to an equivalent RCFG G' with at most two symbols on the right-hand side of each production (such a CFG is called a *binary CFG*).

Recall that each rule is represented by a subgraph (a *sub-DAG*) of the parse DAG consisting of a root node (the left-hand side symbol) and its children (the right-hand side symbols). This sub-DAG can obviously be converted into a binary DAG with at most twice as many edges, and so is at most twice the size of the original sub-DAG. So each rule can be expanded to a set of binary rules with at most twice as many symbols, as each edge in the DAG corresponds to a symbol in a rule. As a result, G is at most doubled in size and thus $|G'| \leq 2|G|$.

Next we convert each production of G' to a SSAS mixing. However, a problem occurs if the same non-terminal appears as a right-hand side symbol in several production rules. Recall that a production in the grammar specifies the left-to-right order in which the right-hand side symbols appear, while the west-to-east order in which assemblies attach is determined by their glues. To produce exactly the assembly desired in a mixing requires combining its subassemblies *with the correct glues*.

To resolve this issue, we construct several copies of every assembly: one for each possible west/east glue pair. Since the grammar is binary, at most two assemblies are mixed in each bin and so three glue pairs is enough to uniquely specify the mixing product. Given a production $A \to BC$, we create six bins and six mixings that assemble the six west/east glue pair combinations for A from the six west/east glue pair combinations for B and C (see Table 1).

Table 1: The set of mixings to produce all necessary glue pair variations for assembly A in the production $A \to BC$.

A glues	$(1,2)$	$(1,3)$	$(2,1)$	$(2,3)$	$(3,1)$	$(3,2)$
B glues	$(1,3)$	$(1,2)$	$(2,3)$	$(2,1)$	$(3,2)$	$(3,1)$
C glues	$(3,2)$	$(2,3)$	$(3,1)$	$(1,3)$	$(2,1)$	$(1,2)$

Lemma 1. *A parse DAG for a binary RCFG G' deriving string s can be converted to a valid SSAS A of size at most $6|G'|$ that constructs an assembly with label string s.*

Proof. We build the mix DAG of A in the following way: For each symbol (terminal and non-terminal) create 6 bins for the glue-pair variants of the symbol. For each production $A \to BC$ of G', mix the 6 bins of B and C into the 6 bins of A as in Table 1. The resulting mix DAG has 6 bins for each symbol of G', each containing a unique glue-pair variant of an assembly with label string corresponding to the string derived by the symbol in G'. Each edge of the parse DAG of G' is converted to 6 edges in the mix DAG of A, one for each glue-pair variant. So $|A| \leq 6|G'|$. □

Theorem 1. *Given an RCFG G deriving a string s, the algorithm described in Section 4.1 computes a SSAS instance A with $|A| \leq 12|G|$ that produces an assembly with label string s.*

Proof. The algorithm converts G to a binary RCFG G', and then converts G' to a mix DAG for A. By Lemma 1, $|A| \leq 6|G'|$. So $|A| \leq 6|G'| \leq 12|G|$. □

4.2 Converting SSASs to RCFGs

Let A be a SSAS constructing an assembly with label string s. We perform a one-to-one mapping from the nodes and edges mix DAG of A to the nodes and edges of the parse DAG of a grammar G. For each leaf bin of A, create a terminal symbol in G equal to the label string of the tile in the bin. For each non-leaf bin of A, create a non-terminal symbol in G. For each mixing in A combining the contents of bins $b_1, b_2, \ldots, b_k$ into bin B, create a production in G with B on the left-hand side and b_1 through b_k on the right-hand side in the order they combine when mixed in B.

Theorem 2. *For any SSAS A constructing an assembly with label string s, an RCFG G deriving s can be constructed from A such that $|G| = |A|$.*

Proof. The terminal symbols of G are equal to the label strings of their corresponding tiles. Each mixing in A produces a single assembly with a label string equal to the string derived by the corresponding non-terminal symbol in G, because the production orders the right-hand side symbols in the same order that they combine in A. So the start symbol of G derives a string equal to the label string of the assembly produces in the root of the mix DAG of A. So G derives s. Each edge of the mix DAG of A causes a right-hand side symbol to appear in a production of G. So $|G| = |A|$. □

4.3 Approximation Equivalence

The conversions presented above in Sections 4.1 and 4.2 immediately imply that approximation algorithms for either problem transfer to the other, at a constant-factor loss.

Corollary 1. *An $O(f(n))$-approximation algorithm for the smallest grammar problem exists if and only if an $O(f(n))$-approximation algorithm for the minimum SSAS problem exists.*

In practice this theorem makes computing efficient SSAS instances easier, as several $O(\log n)$-approximation algorithms to the minimum grammar problem exist [2,11,12]. We have taken advantage of this fact to produce a software tool (described in Section 6) for finding $O(\log n)$ approximations to the minimum SSAS problem in $O(n)$ time using the algorithm by Sakamoto [12].

This result also suggests that finding an improved approximation algorithm for the minimum SSAS problem is unlikely. In 2002, Lehman showed that a polynomial-time approximation algorithm for the smallest grammar problem with factor $o(\frac{\log n}{\log \log n})$ would enable progress on “a difficult algebraic problem in a well-studied area” [9].

4.4 NP-hardness of Minimum SSAS with k Glues

Lehman also showed an NP-hardness result concerning the approximation of the smallest grammar [9]. Unfortunately, the constant-factor conversion in Sections 4.1 and 4.2 cannot be used to extend the NP-hardness to the minimum SSAS-problem as well, as the mixing versus production rule issue covered in Section 4.1 prevents a simple mapping between instances of the two problems. In this section, we give an alternative proof of the NP-hardness of computing a smallest SSAS when the number of glues is restricted. More formally, we show the following:

Theorem 3. *For any fixed $k \geq 6$, given an input string s it is NP-hard to find a smallest SSAS A generating an assembly with label string s such that the number of glues used by A is at most k.*

Proof. We begin with some definitions. For convenience, let the character # be a wildcard character, to be replaced with a new character in each occurrence in the final string. Using that shorthand, we define the function $\textsc{Padding}(t) = \# \circ t \circ \#^{k-3}$. This padding has a nice property: given an assembly $1[t]2$, it is possible to use a single mixing step to construct the assembly $3[\textsc{Padding}(t)]4$. This mixing step involves $k-1$ bins (the maximum possible), $k-2$ of which contain unique tiles not used elsewhere in the string. Furthermore, given an assembly $1[t]2$, it is possible to use two mixings to construct an assembly whose label is the string $\textsc{Padding}(\textsc{Padding}(t))$ with any (non-matching) pair of glues on its left and right sides, even the glues 1 and 2.

Our proof is a reduction from the k-coloring problem, which is known to be NP-hard. In the k-coloring problem, the goal is to assign k colors to the vertices of a graph such that no edge connects two vertices of the same color. Our reduction relies on the relationship between assigning colors to vertices, and assigning glues to the sides of tiles in the assembly.

Suppose that we are given an undirected graph $G = (V, E)$ and a number of colors $k \geq 6$. For each vertex $v \in V$, assign two unique strings L_v and R_v, each string having length $k-1$. For each edge $(u, v) \in E$, assign a unique string $C_{(u,v)}$ of length $k-1$. Furthermore, all characters should be unique, so that when all strings $C_{(u,v)}$, L_v, and R_v are concatenated together, no character is repeated.

In the assembly constructing our string, we want each such string $C_{(u,v)}$, L_v, or R_v to be the label of the assembly in exactly one bin. This would create a consistent mapping from each such label to a pair of glues, left and right. When a pair of labels is placed adjacent to each other in the string we wish to construct, this places restrictions on the glues assigned to those labels. In this way, we create a coloring of vertices.

To construct constraints of this type, we use the following building block:

$$\textsc{Constraint}(t_1, t_2) = \textsc{Padding}(\textsc{Padding}(\#^{k-3} \circ t_1 \circ (t_2 \circ \#^{k-2})))$$

Suppose we are given assemblies $g[t_1]1$ and $1[t_2]h$, where $g, h \neq 1$, but g may or may not equal h. Then we can pick another pair of distinct glues i, j that are different from 1, g, and h. We can then construct the assembly $1[t_2 \circ \#^{k-2}]j$ using a single mixing. We can then combine that assembly with the assembly $g[t_1]1$ and $k-3$ bins containing unique characters to construct the assembly $i[\#^{k-3} \circ t_1 \circ (t_2 \circ \#^{k-2})]j$. As a result, we can construct an assembly with label $\textsc{Constraint}(t_1, t_2)$ with any pair of (non-matching) glues on the left and right, as long as the right glue on the assembly for t_1 matches the left glue on the assembly for t_2.

We have three types of constraints. First, we want the glue on the right side of the assembly for L_v to match the glue on the left side of the assembly for R_v. This can be accomplished using strings of the form $\textsc{Constraint}(L_v, R_v)$ for each vertex $v \in V$. The left side of the assembly for $C_{(u,v)}$ should have a glue matching the coloring of u. Similarly, the right side of the assembly for $C_{(u,v)}$ should have a glue matching the coloring of v. This can be accomplished by having the following strings for each edge $(u, v) \in E$: $\textsc{Constraint}(L_u, C_{(u,v)})$ and $\textsc{Constraint}(C_{(u,v)}, R_v)$. We then concatenate these $|V| + 2|E|$ constraint strings together to create the string s.

What is the optimal way to construct s with an SSAS? First, note that if the label of an assembly occurs only once in s, then the assembly cannot not be included in more than one mixing. If each assembly were used in at most one mixing, then the optimal SSAS would ensure that every mixing involved $k - 1$ subassemblies, and the structure of the SSAS would effectively be a tree. Using an assembly more than once corresponds to taking such a tree and merging subtrees, thereby removing edges. In s, the only substrings that occur more than once are substrings of L_v, R_v, and $C_{(u,v)}$, and each such substring has length $\leq k - 1$. So a merge can remove at most $k - 1$ edges. To remove as many edges as possible, each merge should remove $k-1$ edges, and we should perform as many merges as possible. Hence, an optimal SSAS contains exactly one assembly for each string L_v, R_v, or $C_{(u,v)}$, and reuses the assembly to generate every occurrence of the string in s.

Now suppose that we have an optimal SSAS for s. Therefore, there is exactly one assembly for each of L_v, R_v, and $C_{(u,v)}$, and the assembly is used everywhere that the string occurs in s. Construct the corresponding graph coloring by assigning vertex $v \in V$ the color corresponding to the glue on the right side of L_v. Because $L_v R_v$ is a substring of s, the glue on the left side of R_v must match the glue on the right side of L_v. Because $L_u C_{(u,v)}$ is a substring of s, the left glue of $C_{(u,v)}$ must be the same as the right glue of L_u. Because $C_{(u,v)} R_v$ is a substring of s, the right glue of $C_{(u,v)}$ must be the same as the left glue of R_v. Because $C_{(u,v)}$ is the label of a single assembly, we know that the glue on the left side cannot match the glue on the right side (otherwise it would form an infinite tile). As a result, we know that for any edge (u, v), the glue assigned to u must be distinct from the glue assigned to v. Hence, the coloring is valid.

Given a valid coloring, how do we construct an optimal SSAS? First, we construct one assembly for each of L_v, R_v, and $C_{(u,v)}$. Given a vertex v colored with color i, we construct the assembly L_v with glue i on the right side and the assembly R_v with glue i on the left side. For the assembly $C_{(u,v)}$, the glue on the left should correspond to the color of u, while the glue on the right should correspond to the color of v. These glue assignments satisfy the constraints encoded in s, so we can construct an assembly for each constraint string as outlined above. To combine the assemblies for the constraint strings, we take advantage of our ability to assign arbitrary glues to the left and right sides of the assembly for each constraint string. By assigning glues correctly, we can use $(k-1)$-way mixes to construct the final assembly, resulting in an optimal SSAS.

5 Separation Between SASs and RCFGs

Now we show that the general 1D staged self-assembly model (SAS) is *not* equivalent to RCFGs. The proof is constructive: we give a set of strings and describe a set of SAS instances that produce assemblies with these label strings. We then show that any RCFG producing these strings is asymptotically larger than the SAS instance producing the corresponding label string.

It might appear obvious that allowing bin parallelism should allow a reduction in the amount of work needed to construct an assembly. However, using parallelism has two costs that make saving work difficult. First, for any minimal SAS, no two assemblies in the same bin may share a common glue (this is proven in Lemma 3). As a result, additional parallelism requires more unique glues, which in turn requires more starting bins, and thus more work. Second, since the goal of an assembly system is to construct a single goal assembly, bins with parallelism must eventually be "collapsed" into a single bin with a single object (otherwise the parallelism was extraneous). Collapsing bins with parallelism involves adding tiles to join the various assemblies together, and since the glues on each assembly are unique, creating and mixing the joining tiles requires additional work proportional to the amount of parallelism in the bin.

5.1 A Set of Strings S_k

To derive an asymptotic bound between SASs and RCFGs, we use a special set of strings that can be built by small SASs but require large RCFGs. Each string consists of a sequence of *interleavings* of pairs of smaller strings. We will consider only odd values of k for the remainder of the paper.

Let $\textsc{Binary}(i, \ell)$ be the binary representation of i of length ℓ. The following is a function used to double every character in a string:

$$\textsc{Double}(b_1 b_2 b_3 \dots b_n) = b_1 b_1 b_2 b_2 b_3 b_3 \dots b_n b_n$$

We wish to encode a number of distinct "characters" in binary. To construct a suitably hard-to-compress string, we want to ensure that the beginning and end of each encoding are clearly delineated. To that end, we define the following strings for all values of k and all values of $i < 2k$:

Definition 9. $A_{k,i} = (01) \circ \textsc{Double}(\textsc{Binary}(i, 1 + \lceil \log k \rceil)) \circ (01)$.

Note that each such string has length $6 + 2\lceil \log k \rceil$.

We wish to use these characters to construct a string with a lot of structure (so that it is efficiently constructible using a SAS) but minimal repetition (so that it is not efficiently constructible using a CFG). To minimize repetition, we choose a string with the property that no sequential pair of substrings $A_{k,i}$ is repeated. We define the following functions, which are permutations for $0 \leq x < k$:

$$\pi_{k,0}(x) = 2x \bmod k \qquad\qquad \pi_{k,1}(x) = 2x + 1 \bmod k$$

We use these two simple functions to construct a more complex permutation.

Definition 10. *Say that the bits of* $\textsc{Binary}(i, \ell)$ *are* $b_1, \ldots, b_\ell$. *Then*

$$\Pi_{k,\ell,i}(j) = \pi_{k,b_\ell}(\pi_{k,b_{\ell-1}}(\ldots \pi_{k,b_2}(\pi_{k,b_1}(j)) \ldots)).$$

Because k is odd, this function is a permutation for $0 \leq j < k$. In addition, as long as $0 \leq i < 2^\ell$, this function has the property that $\Pi_{k,\ell,i}(j) = \left(2^\ell \cdot j + i\right) \bmod k$. This means that for fixed values of k, ℓ, and j, each value of i such that $0 \leq i < k$ will generate a different value of $\Pi_{k,\ell,i}(j)$.

This permutation can be used to ensure that no sequential pair of characters is repeated. To do so, we construct pairs of characters as follows:

$$C_{k,i,j} = A_{k,j} \circ (01)^{\lceil \log k \rceil} \circ A_{k,k+\Pi_{k,\lceil \log k \rceil,i}(j)} \circ (01)^{\lceil \log k \rceil}.$$

We concatenate these pairs to construct $P_{k,i} = C_{k,i,0} \circ C_{k,i,1} \circ \ldots \circ C_{k,i,k-1}$. Note that the length of each $C_{k,i,j}$ is $12 + 8\lceil \log k \rceil$, and therefore the length of each $P_{k,i}$ is $(12 + 8\lceil \log k \rceil) \cdot k$.

We concatenate each $P_{k,i}$ to get the string we wish to compress:

Definition 11. $S_k = 01 \circ P_{k,0} \circ 01 \circ P_{k,1} \circ 01 \circ \ldots \circ 01 \circ P_{k,k-1} \circ 01$

In the next two subsections we give bounds on compressing S_k using both a RCFG and a SAS.

5.2 A SAS Upper Bound for S_k

Now we describe a self-assembly system using bin parallelism that produces an assembly with S_k as its label string. The system is broken down into several subsystems described in this section. A diagram of the SAS for S_3 is seen in Figure 3.

Constructing $A_{k,i}$ for all $0 \leq i < 2k$ Say that we are given $2k$ glue pairs x_i, y_i, and that we want to assemble $x_i[A_{k,i}]y_i$. for each $0 \leq i < 2k$. In addition, say that we are given three additional glues g_0, g_1, and g_2 for use in our construction. Let $\ell = 1 + \lceil \log k \rceil$.

For each binary string s of length $\leq \ell$, we construct two bins: I_s and F_s. Let $s = t \circ b$, where $b \in \{0, 1\}$. I_s will contain an assembly with glue g_0 on the left, glue g_1 on the right, and the label $\textsc{Double}(t) \circ b$. F_s will contain an assembly with glue g_0 on the left, glue g_2 on the right, and the label $\textsc{Double}(s)$. The assembly in bin I_s will be constructed by adding the tile $g_2[b]g_1$ to the assembly in bin F_t. The assembly in bin F_s will be constructed by adding the tile $g_1[b]g_2$ to the assembly in bin I_s.

To finish this construction, we add the constant-sized assemblies $x_i[01]g_0$ and $g_2[01]y_i$ to the bin $F_{\textsc{Binary}(i,\ell)}$. This ensures that for $0 \leq i < 2k$, the bin $F_{\textsc{Binary}(i,\ell)}$ contains an assembly with the label $A_{k,i}$. The total number of bins required for this construction is $\Theta(k)$.

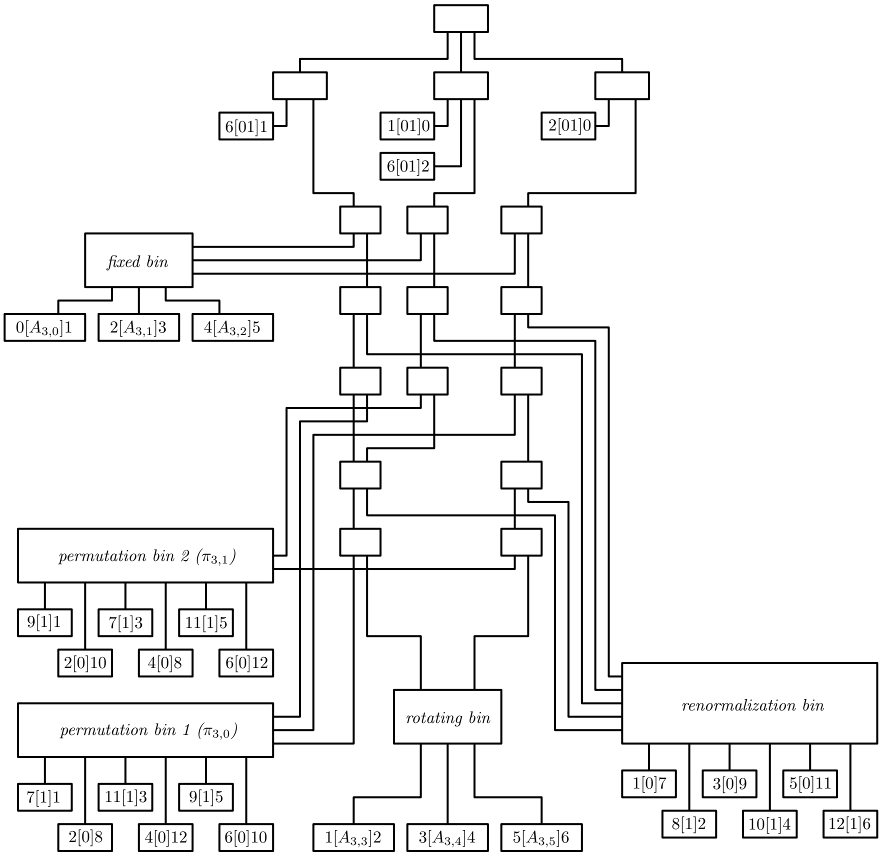


Fig. 3: The mix DAG for a SAS generating an assembly with label string S_3.

Fixed and Rotating Bins The fixed bin contains the following set of tiles:

$$0[A_{k,0}]1, 2[A_{k,1}]3, \ldots, (2k-2)[A_{k,k-1}](2k-1)$$

The rotating bin contains the following set of tiles:

$$1[A_{k,k+0}]2, 3[A_{k,k+1}]4, \ldots, (2k-1)[A_{k,k+(k-1)}](2k)$$

Permutation and Renormalization Bins Permuting the assemblies in the rotating bin is simulated by attaching *permutation tiles* to the east and west ends of those assemblies. The permutations $\pi_{k,0}$ and $\pi_{k,1}$ are implemented as two sets of $2k$ tiles, each set in a separate *permutation bin*. A third set of $2k$ tiles are put in a *renormalization bin* used to solve a technical issue with the permutation bins.

The permutation bin for $\pi_{k,0}$ has tiles that replace the primal glues of assembly i ($2i+1$ and $2i+2$) with the dual glues of assembly $\pi_{k,0}(i)$ ($2\pi_{k,0}(i)+$

$1 + (2k+1)$ and $2\pi_{k,0}(i) + 2 + (2k+1)$) for all assemblies $0 \le i \le k-1$. The permutation bin for $\pi_{k,1}$ is constructed analogously. Each tile attaches to either the east or west end of the assembly and correspondingly has the primal and dual glues on its east and west sides. The tiles attaching to the east end of the assembly have the label 0; the tiles attaching to the west end of the assembly have the label 1.

The renormalization bin has a pair of tiles for changing the dual glues of assembly i ($(2i+1)+(2k+1)$ and $(2i+2)+(2k+1)$) to its primal glues ($(2i+1)$ and $(2i+2)$). The tiles attaching to the east end of each assembly have the label 1; the tiles attaching to the west end of each assembly have the label 0.

Creating Interleaved Assemblies The permutation and renormalization bins are applied in a branching manner to produce all permutation sequences of length ℓ. First $\pi_{k,0}$ and $\pi_{k,1}$ are mixed separately with the rotating bin, then $\pi_{k,0}$ and $\pi_{k,1}$ are each mixed separately with the product of both of these mixings, etc. After each mixing with a permutation bin, the renormalization bin is mixed with the product. After all permutation sequences are created, the fixed bin is mixed with each, creating single assemblies with label strings $P_{k,i}$ for all $0 \le i < k$.

Combining Interleaved Assemblies The final step is to combine each assembly with label $P_{k,i}$ into a single assembly. Each assembly is contained in a separate bin after its production, and has glue 0 on its west side and glue $2k$ on its east side. To the assembly with label $P_{k,i}$, the tiles $(2k+1+i)[1]0$ and $(2k)[0](2k+2+i)$ are added. Then these assemblies are combined to produce a single long assembly with glue $(2k+1)$ on the west side, and glue $(3k+1)$ on the east. To finish off the assembly, two more tiles are added: $(null)[0](2k+1)$ and $(3k+1)[1](null)$. This ensures that the final result is an assembly with the label S_k and null glues on both ends.

Theorem 4. *The SAS described in Section 5.2 has size $O(k)$.*

Proof. Break the SAS into the following sections:

1. Creating the a-bin and b-bin.
2. Creating the permutation and renormalization bins.
3. Creating the interleaved assemblies.
4. Combining interleave assemblies.

Item 1 requires $O(k)$ edges to create a tile for each symbol in A_k or B_k respectively and mixing them together. Item 2 requires $O(k)$ edges to create three bins each with a pair of tiles for each c-buffered element of the b-bin. Item 3 requires $O(k)$ edges: this portion of the mix DAG resembles an upside-down tree and contains no more than two leaves per permutation assembly. Item 4 requires $O(1)$ edges per assembly (and thus $O(k)$ edges total) to add two location-specifying tiles and combine it with the other assemblies into a single bin. In total k interleave assemblies (one per shift) are created, so $O(k)$ edges are in this portion of the mix DAG. Combining interleave assemblies is done by adding at most two tiles to each interleave assembly followed by combining them into a single bin. A constant number of edges exist for each assembly, so $O(k)$ edges exist in this portion of the mix DAG. □

5.3 An RCFG Lower Bound for S_k

The following definition and theorem are taken from [11].

Definition 12. *As defined in [5], the size of the LZ-factorization of a string s (denoted $|LZ(s)|$) is the number of elements generated by the LZ77 algorithm without self-referencing.*

Theorem 5. *For an RCFG G generating a string s, $|LZ(s)| \leq |G|$.*

Lemma 2. *All factors in the LZ-factorization of S_k have size $< 16\lceil \log k \rceil + 26$.*

Proof. Assume, for the sake of contradiction, the LZ-factorization of S_k contains some factor y of size $\geq 16\lceil \log k \rceil + 26$. Then the factor is long enough that there must be some i, j such that $C_{k,i,j}$ is a substring of y. Let x be the part of the string preceding y. Then by the definition of LZ factorization, y is a substring of x, and therefore $C_{k,i,j}$ is a substring of x.

$C_{k,i,j}$ contains as a substring the string $A_{k,j}$. To ensure the correct parity on runs of characters, the portion of x where $A_{k,j}$ is found must have been completely generated by some other A_{k,j^*}. Then it must be that $A_{k,j} = A_{k,j^*}$, and by Definition 9, it follows that $j = j^*$. So the portion of x where $C_{k,i,j}$ is found must have been completely generated by some other $C_{k,i^*,j}$, where $i \neq i^*$. Then $A_{k,k+\Pi_{k,\lceil \log k \rceil,i}(j)} = A_{k,k+\Pi_{k,\lceil \log k \rceil,i^*}(j)}$. By Definition 9, it follows that $k + \Pi_{k,\lceil \log k \rceil,i}(j) = k + \Pi_{k,\lceil \log k \rceil,i^*}(j)$. Therefore, by Definition 10, $i = i^*$, which gives us a contradiction. □

Theorem 6. *The smallest CFG that can be used to construct S_k has size $\Omega(k^2)$.*

Proof. By Lemma 2, the maximum length of an LZ factor is $16\lceil \log k \rceil + 29$. The sum of the lengths of the LZ factors is equal to $|S_k| = \Theta(k^2 \log k)$. Hence, the number of LZ factors is $\Omega(k^2)$. By Theorem 5, the size of the minimum grammar must therefore be $\Omega(k^2)$. □

5.4 Asymptotic Separation of SASs and RCFGs for S_k

Separation refers to the minimum difference in size between an RCFG and a SAS generating the same (label) string. Here we show the separation achieved for the strings S_k, where k is the number of glues used to generate the label string S_k by the SAS in Section 5.3 and n is the length of S_k.

Corollary 2. *The strings S_k have separation $\Omega(k)$.*

Proof. By Theorem 6, any RCFG generating S_k has size $\Omega(k^2)$. By Theorem 4, a self-assembly system of size $O(k)$ exists that produces an assembly with label string S_k. So the ratio of the size of any grammar generating S_k to the size of some SAS instance is $\Omega(k)$. □

Corollary 3. *The strings S_k have separation $\Omega(\sqrt{n/\log n})$.*

Proof. The length of S_k is $\Theta(k^2 \log k)$. So $k = \Theta(\sqrt{n/\log n})$. By Corollary 2, the separation is $\Omega(k)$. So the separation is also $\Omega(\sqrt{n/\log n})$. □

Given that the number of glues is limited in practice, it is natural to consider whether $\Omega(k)$ separation is possible for k glues where $k \ll n$. We show this is possible for $k = \Theta(\log n)$.

Definition 13. *Define the recursive string $T_{k,t} = 01 \circ T_{k,t-1} \circ 01 \circ T_{k,t-1} \circ 01$, where $T_{k,1} = S_k$. The length of $T_{k,t}$ is $\Theta(2^t |S_k|) = \Theta(2^t k^2 \log k)$.*

Theorem 7. *The strings $T_{k,k}$ have separation $\Omega(k)$ and use $\Theta(\log n)$ glues.*

Proof. Since $T_{k,k}$ has S_k as a substring, any CFG generating $T_{k,k}$ has size $\Omega(k^2)$ by Theorem 6. To construct a SAS to generate this string, we first use the SAS described in Section 5.2 to generate an assembly $a[S_k]b$. We can then add a constant number of tiles to get two assemblies $c[1S_k0]e$ and $e[1S_k0]d$, which when combined create the assembly $c[1S_k01S_k0]d$. We then add two more tiles to construct the assembly $a[01S_k01S_k01]b$. This process can then be repeated k times. In total $O(k)$ additional work is performed, so the new SAS has size $O(k)$. The length n of the string is $\Theta(2^k k^2 \log k)$, so $k = \Theta(\log n)$. □

5.5 Upper Bounds for Separation of SASs and RCFGs

The SASs described in Section 5.2 constructing S_k used $O(k)$ distinct glue pairs to achieve a separation of $\Omega(k)$. We now show bounds on the worst-case separation in terms of the number of glues k and the length of the string n.

Lemma 3. *Given a minimal SAS A, any two distinct assemblies A_1 and A_2 in the same bin must have different glues on either the west side or the east side.*

Proof. For the sake of contradiction, say that there is a distinct pair of assemblies A_1 and A_2 with matching glues on both the west and east sides of the assemblies. Because the accessible glues on both assemblies are identical, any assembly which adheres to A_1 must also adhere to A_2, and vice versa. Hence, for every superassembly of A_1, there is a corresponding superassembly of A_2 in the same bin with the same accessible glues, but a different label sequence. Any attempt to merge two such assemblies to create a single assembly results in an infinite label sequence. So the SAS A cannot produce a single goal assembly, violating the definition of a SAS. □

Corollary 4. *Given a minimal SAS A using k glues to produce a string s, each bin in A contains at most k^2 distinct assemblies.*

Lemma 4. *Given a SAS A using k glues and generating an assembly with label string s, an RCFG of size $O(k^2|A|)$ generating s can be constructed.*

Proof. For each bin in A and each distinct assembly in that bin, construct one bin in the SSAS B. By Corollary 4, the number of bins in B will be at most k^2 times the number of bins in A.

Now consider what happens when ℓ bins in A are simultaneously mixed to produce a single bin c containing several assemblies. How many edges must we add to B to ensure that each assembly in c is correctly constructed in B? To determine this, we define G to be a directed graph with a node corresponding to each glue and, for each distinct input assembly $g_1[s]g_2$, a directed edge from g_1 to g_2. Then each distinct assembly in c corresponds to a source-sink pair in G, and each possible way to construct that assembly corresponds to a path in G from the source of the assembly to the sink of the assembly.

Say that there exist three glues g_1, g_2, g_3 such that (g_1, g_2) and (g_2, g_3) are edges in G but (g_1, g_3) is not an edge in G. Then we can mix the assembly corresponding to the edge (g_1, g_2) with the assembly corresponding to the edge (g_2, g_3) to get an assembly with glue g_1 to the west and glue g_3 to the east. This is equivalent to adding the edge (g_1, g_3) to G. Each such mixing requires us to add a constant number of nodes and edges to the mix DAG B, and increases the number of edges in G by 1. The graph G can never have more than k^2 edges, so repeated mixings of this type add a total of $O(k^2)$ work to B. Hence, any mixing of bins in A can be replaced by $O(k^2)$ binary mixes in B. As a result, $|B|$ has size $O(k^2|A|)$, and can therefore be converted to an RCFG with size $O(k^2|A|)$ by Theorem 2. □

Theorem 8. *With respect to the total number of distinct glue pairs k, the separation for any string is $O(k^2)$.*

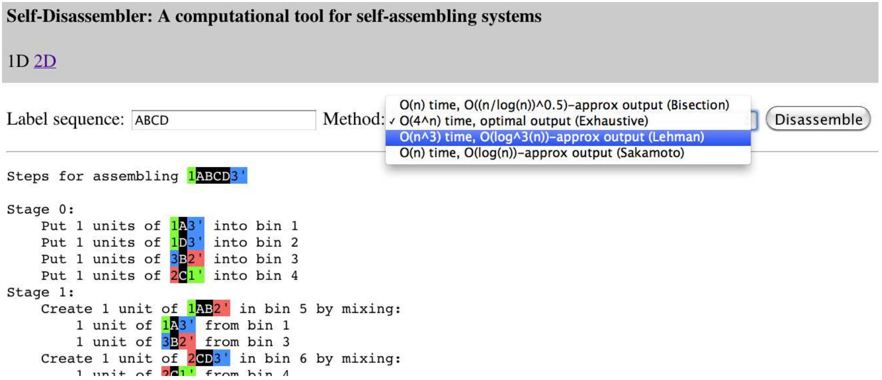
Proof. Let A be a SAS using k glues that generates a string s. By Lemma 4, there is an RCFG of size $O(k^2|A|)$ that generates s. So separation is at most $O(k^2)$. □

Theorem 9. *With respect to the length of the string n, the separation for any binary string is $O\big((n/\log n)^{2/3}\big)$.*

Proof. Let A be a SAS generating a string s of length n. Let k be the number of glues used in A. Either $k = O\big((n/\log n)^{1/3}\big)$ or $k = \omega\big((n/\log n)^{1/3}\big)$. If $k = O\big((n/\log n)^{1/3}\big)$ then by Lemma 4 there is an RCFG of size $O(k^2|A|) = O\big((n/\log n)^{2/3} \cdot |A|\big)$ generating s. So the separation is at most $O\big((n/\log n)^{2/3}\big)$. Now suppose k is $\omega\big((n/\log n)^{1/3}\big)$. Then $|A| = \omega\big((n/\log n)^{1/3}\big)$. Lemma 2 of Section 2.2 in [9] shows that there is an RCFG of size $O(n/\log n)$ generating s. Hence, the separation is $o\big((n/\log n)^{2/3}\big)$. So in both cases the separation is $O\big((n/\log n)^{2/3}\big)$. □

6 Implementation and Experimentation

Section 4 demonstrated the equivalence of approximation algorithms for the minimum CFG and minimum SSAS problems. In this section we present an easy-to-use software tool named ‘Self-Disassembler’ utilizing this equivalence to quickly compute efficient SSASs for user-specified goal assemblies. A screenshot of the system is seen in Figure 4.



Self-Disassembler: A computational tool for self-assembling systems

1D 2D

Label sequence: ABCD

Method:
O(n) time, O((n/log(n))^0.5)–approx output (Bisection)
✓ O(4^n) time, optimal output (Exhaustive)
O(n^3) time, O(log^3(n))–approx output (Lehman)
O(n) time, O(log(n))–approx output (Sakamoto)
Disassemble

```
Steps for assembling 1ABCD3'

Stage 0:
    Put 1 units of 1A3' into bin 1
    Put 1 units of 1D3' into bin 2
    Put 1 units of 3B2' into bin 3
    Put 1 units of 2C1' into bin 4
Stage 1:
    Create 1 unit of 1AB2' in bin 5 by mixing:
        1 unit of 1A3' from bin 1
        1 unit of 3B2' from bin 3
    Create 1 unit of 2CD3' in bin 6 by mixing:
        1 unit of 2C1' from bin 4
```

Fig. 4: A screenshot of Self-Disassembler: an online tool implementing algorithms for computing SSAS systems.

To use Self-Disassembler, the user provides an input assembly as a sequence of characters typed into the text box. A drop-down menu provides a list of minimum CFG approximation algorithms with asymptotic running times and approximation ratios. Pressing ‘Disassemble’ sends the input string to the web server, where a grammar approximation algorithm is applied and the resulting grammar is converted into a SSAS. Additional information such as bin and stage assignments and volume measurements for each mixing is also computed. The result is then sent back to the user and displayed in a color-coded ‘recipe’ format.

The four selectable algorithms available to the user are ‘Exhaustive’, ‘Bisection’, ‘Lehman’, and ‘Sakamoto’. The Exhaustive algorithm is a simple brute-force $O(4^n)$ exponential algorithm that solves the smallest CFG problem exactly. The Bisection algorithm [8] is a $O((n/\log(n))^{0.5})$-approximation algorithm that recursively adds production rules that merge two equal-sizes halves of the string, followed by merging any nonterminals that produce the same substring. The Lehman algorithm is a $O(\log^3(n))$-approximation algorithm described in [2]. Finally, the Sakamoto algorithm is the $O(\log(n))$-approximation algorithm described in [12].

Performance experiments were run with an offline version of the software to evaluate the speed and output SSAS sizes using the four algorithm choices. All running times are measured on a Macbook Pro laptop with a 2.66 GHz Intel Core i7 processor and 4 GB of RAM, with implementations in Python using the Python 2.6.5 interpreter. We found that for small $n \leq 20$ all algorithms (including the Exhaustive algorithm) performed nearly identically, with the exception of the speed of the Exhaustive algorithm, which became too slow for $n \geq 14$.

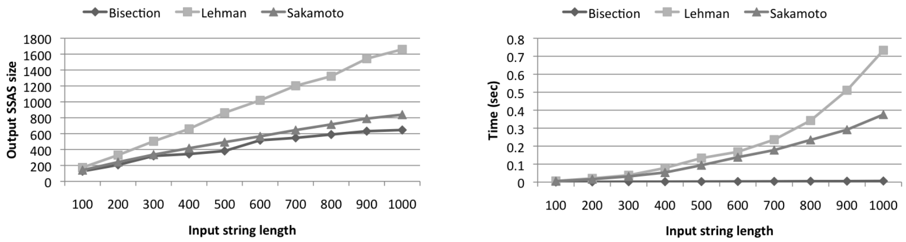


Fig. 5: Plots of the output SSAS sizes and speed of the three smallest grammar approximation algorithms. Each data point is averaged from a sample of 10 random binary strings.

For larger n ($100 \leq n \leq 1000$), the Bisection algorithm outperformed the Lehman and Sakamoto algorithms in both speed and output SSAS size for random binary strings. Additional experiments in which strings consisting of repeated concatenations of a common string of length $1 \leq i \leq n$, i.e. strings with complexity ranging from $\Theta(\log(n))$ to $\Theta(2^n)$ were also performed. As string complexity decreased, the output SSAS sizes in these cases became signficantly smaller and all three algorithms ran significantly faster, but the relative performance of the algorithms remained similar.

Acknowledgments

We thank Martin Demaine, André Schulz, Diane Souvaine, and Hyunmin Yi for helpful discussions, and anonymous reviewers for helpful suggestions.

References

1. Harish Chandran, Nikhil Gopalkrishnan, and John Reif. The tile complexity of linear assemblies. In *ICALP 2009*, volume 5555 of *Lecture Notes in Computer Science*, pages 235–253. 2009.
2. Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, April Rasala, Amit Sahai, and abhi shelat. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 792–801, New York, NY, USA, 2002. ACM.
3. Eugen Czeizler and Alexandru Popa. Synthesizing minimal tile sets for complex patterns in the framework of patterned dna self-assembly. In *DNA Computing and Molecular Programming*, volume 7433 of *Lecture Notes in Computer Science*, pages 58–72. 2012.
4. Erik Demaine, Martin Demaine, Sndor Fekete, Mashhood Ishaque, Eynat Rafalin, Robert Schweller, and Diane Souvaine. Staged self-assembly: nanomanufacture of arbitrary shapes with $O(1)$ glues. *Natural Computing*, 7:347–370, 2008.

5. Martin Farach and Mikkel Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20:388–404, 1998.
6. Mika Göös and Pekka Orponen. Synthesizing minimal tile sets for patterned dna self-assembly. In Yasubumi Sakakibara and Yongli Mi, editors, *DNA Computing and Molecular Programming*, volume 6518 of *Lecture Notes in Computer Science*, pages 71–82. Springer Berlin / Heidelberg, 2011.
7. Ming-Yang Kao and Robert Schweller. Randomized self-assembly for approximate shapes. In *ICALP 2008*, volume 5125 of *Lecture Notes in Computer Science*, pages 370–384. 2008.
8. John Kieffer, En hui Yang, Gregory Nelson, and Pamela Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Transactions on Information Theory*, 46:1227–1245, July 2000.
9. Eric Lehman. *Approximation Algorithms for Grammar-Based Data Compression*. PhD thesis, MIT, 2002.
10. Xiaojun Ma and Fabrizio Lombardi. Synthesis of tile sets for dna self-assembly. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(5):963–967, may 2008.
11. Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. In Alberto Apostolico and Masayuki Takeda, editors, *Combinatorial Pattern Matching*, volume 2373 of *Lecture Notes in Computer Science*, pages 20–31. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45452-7_3.
12. Hiroshi Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms*, 3(2–4):416–430, 2005.
13. David Soloveichik and Erik Winfree. Complexity of self-assembled shapes. In *DNA Computing*, volume 3384 of *Lecture Notes in Computer Science*, pages 344–354. 2005.
14. Erik Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, Caltech, 1998.