# Improving Efficiency of 3-SAT-Solving Tile Systems

Yuriy Brun

University of Washington, Seattle, WA 98195-2350, USA
brun@cs.washington.edu

**Abstract.** The tile assembly model has allowed the study of the nature's process of self-assembly and the development of self-assembling systems for solving complex computational problems. Research into this model has led to progress in two distinct classes of computational systems: Internet-sized distributed computation, such as software architectures for computational grids, and molecular computation, such as DNA computing. The design of large complex tile systems that emulate Turing machines has shown that the tile assembly model is Turing universal, while the design of small tile systems that implement simple algorithms has shown that tile assembly can be used to build private, fault-tolerant, and scalable distributed software systems and robust molecular machines. However, in order for these types of systems to compete with traditional computing devices, we must demonstrate that fairly simple tile systems can implement complex and intricate algorithms for important problems. The state of the art, however, requires vastly complex tile systems with large tile sets to implement such algorithms.

In this paper, I present $\mathbb{S}_{FS}$, a tile system that decides 3-$SAT$ by creating $O^\star(1.8393^n)$ nondeterministic assemblies in parallel, while the previous best known solution requires $\Theta(2^n)$ such assemblies. In some sense, this tile system follows the most complex algorithm implemented using tiles to date. I analyze that the number of required parallel assemblies is $O^\star(1.8393^n)$, that the size of the system's tileset is $147 = \Theta(1)$, and that the assembly time is nondeterministic linear in the size of the input. This work directly improves the time and space complexities of tile-inspired computational-grid architectures and bridges theory and today's experimental limitations of DNA computing.

## 1 Introduction

Self-assembly is a process by which simple objects in nature combine and coordinate to form complex objects. For computer scientists, it is interesting to study self-assembly from a computational point of view as self-assembling systems have been shown capable of computing functions [2,19], assembling complex shapes [15,17], and guiding distributed robotics systems [1,12].

The tile assembly model [20,15] is a formal mathematical model that allows studying the time and space complexities of self-assembling systems. Winfree showed that the tile assembly model is Turing universal [19] by demonstrating that tile systems can emulate Turning machines. Adleman has identified two important measures of tile systems: assembly time and tileset size; in some ways, these measures are analogous to the time and space complexities of traditional computer programs [2].

Study of tile systems has led to two types of computational systems: Internet-sized distributed grids [8] and molecular computers [2]. Both types benefit from efficient tile systems with small tilesets: the speed of computational grids is proportional to the number of tile types [8] and the state of the art in DNA computation is systems with no more than tens of distinct tile types [3,14]. Winfree's universal tile systems are in some sense inefficient and require thousands of distinct tile types [19]. Lagoudakis et al. [11] presented a tile system that solves 3-$SAT$, though their best solution required $\Theta(n^2)$ distinct tile types and $\Theta(2^n)$ distinct nondeterministic assemblies to solve an $n$-variable problem, resulting in over $10^5$ distinct tile types and $10^{15}$ distinct assemblies necessary to solve a 50-variable problem. I have begun the work of reducing the complexity by designing tile systems that solve complex computational problems using relatively small tilesets (e.g., adding using 8 distinct tile types [4], multiplying using 28 [4], factoring integers nondeterministically using 50 [5], and solving two NP-complete problems nondeterministically, 3-$SAT$ using 64 [7] and *SubsetSum* using 49 [6]). However, thus far, existing tile systems implement only the most naïve, simple, and inefficient algorithms. For example, today's best known NP-complete problem-solving tile systems require $\Theta\left(2^n\right)$ distinct assemblies for an input of size $n$. While we do not know of polynomial-time algorithms to solve such problems, we do know of exponential-time algorithms with a base smaller than 2 [21]. Here, I present a tile system that implements a somewhat complex known algorithm for solving 3-$SAT$ using only $O^\star(1.8393^n)$ distinct assemblies (where the $O^\star$ notation hides constant and polynomial factors). This system uses 147 distinct tile types and demonstrates that complex algorithms can be implemented using tiles in a systematic manner, (1) directly improving the time and space complexities of tile-inspired computational-grid architectures [8] and (2) bridging theory and today's experimental limitations of DNA computing [2,3].

3-$SAT$ is a well known NP-complete problem of deciding whether a 3CNF Boolean formula is satisfiable. The naïve algorithms for solving 3-$SAT$ explore the $\Theta\left(2^n\right)$ distinct assignments, for formulae with $n$ distinct variables, checking if any one of them satisfies the formula. While we are unaware of subexponential-time algorithms to solve NP-complete problems, there are algorithms that perform in exponential time but with a base smaller than 2. Woeginger [21] provides a fairly complete survey of more-efficient exponential-time algorithms for 3-$SAT$, one of which I employ here. For my discussion, I define the $O^\star$ notation, which is similar to the $O$ notation but ignores both constant and polynomial factors. Thus I will say $O^\star(m(x))$ for a complexity of the form $O(m(x)\cdot poly(x))$. The justification for this notation is that the exponential growth of $m(x)$ will dominate all polynomial factors for large $x$. For example, if $f$ is a function such that $f(x) = O\left(1.4142^x x^4\right)$, then I write $f(x) = O^\star(1.4142^x)$. Note that the exponential term dominates and one could say $f(x) = O(1.4143^x)$ and forgo the

$O^\star$ notation altogether; however, that would not most accurately describe the functions.

While the naïve algorithms explore each of the possible $2^n$ truth assignments to the $n$ variables, a more intricate algorithm can explore a subset of those assignments by noting the following fact: if the Boolean formula contains the clause $(x_1 \vee \neg x_2 \vee x_3)$, then the algorithm need not explore any of the $2^{(n-3)}$ assignments with $x_1 = x_3 = \mathit{FALSE}$ and $x_2 = \mathit{TRUE}$ because this clause would not be satisfied by any of those assignments. Instead, the algorithm explores only assignment with (1) $x_1 = \mathit{TRUE}$, or (2) $x_1 = x_2 = \mathit{FALSE}$, or (3) $x_1 = \mathit{FALSE}$, $x_2 = \mathit{TRUE}$, and $x_3 = \mathit{TRUE}$. Thus, deciding an $n$-variable $m$-clause Boolean formula can be done by recursively deciding three Boolean formulae: each with one fewer clause and with one, two, and three fewer variables, respectively. Thus if $T(n, m)$ denotes the time necessary to decide an $n$-variable $m$-clause Boolean formula, then $T(n, m) = O(1) + T(n-1, m-1) + T(n-2, m-1) + T(n-3, m-1)$. This recurrence has the closed form solution $T(n, m) = O^\star(1.8393^n)$ [21]. By examining the branching step, it is possible to improve the algorithm further to an $O^\star(1.6181^n)$ algorithm [13]. Using quantitative analysis of the number of resulting 2-clauses from such branching improves the time complexity to $O^\star(1.5783^n)$ [16]. The champion algorithm using this technique achieves a time complexity of $O^\star(1.4963^n)$ [9,10], and other techniques result in even faster algorithms [21]. It is not my goal to explore the fastest such algorithm here, but rather to demonstrate that it is possible to implement one such complex algorithm using a tile system with a small tileset. I will thus concentrate on developing a tile system that follows the $O^\star(1.8393^n)$ algorithm, and argue that since the other algorithms are similar, it is possible to design tile systems for those algorithms as well.

## 2 Tile Assembly Model

The tile assembly model [20,15] is a formal model of crystal growth. It was designed to model self-assembly of molecules such as DNA. It is an extension of a model proposed by Wang [18]. The model was fully defined in [15], and the definitions I use are similar to those. Full formal definitions can be found in [7].

Intuitively, the model has *tiles*, or squares, that stick or do not stick together based on various *binding domains* on their four sides. Each tile is a four tuple of binding domains, one on each (north, east, south, and west) of its sides. The special *empty* $= \langle \mathit{null}, \mathit{null}, \mathit{null}, \mathit{null} \rangle$ tile denotes an empty position. The four binding domains, elements of a finite alphabet $\Sigma$, define the type of the tile. The strength of the binding domains are defined by the *strength function* $g \colon \Sigma \times \Sigma \to \mathbb{N}$. A mapping from positions on a 2-D grid to tiles is called a *configuration*, and a tile may *attach* in empty positions on the grid if the total strength of all the binding domains on that tile that match its neighbors exceeds the current *temperature*. Finally, a *tile system* $\mathbb{S}$ is a triple $\langle T, g, \tau \rangle$, where $T$ is a finite set of tiles, $g$ is a strength function, and $\tau \in \mathbb{N} = \mathbb{Z}_{\geq 0}$ is the temperature.

Starting from a *seed configuration* $S$, tiles may attach to form new configurations. If that process terminates, the resulting configuration is said to be *final.* At some times, it may be possible for more than one tile to attach at a given position, or there may be more than one position where a tile can attach. If for all sequences of tile attachments, all possible final configurations are identical, then $\mathbb{S}$ is said to produce a *unique* final configuration on $S$. The *assembly time* of the system is the minimal number of steps it takes to build a final configuration, assuming maximum parallelism.

In solving NP-complete problems, it is important to compute a particular subset of functions: the characteristic functions of subsets of the natural numbers. A characteristic function of a set has value 1 on arguments that are elements of that set and value 0 on arguments that are not elements of that set. Typically, in computer science, programs and systems that compute such functions are said to decide the set. Since for all constants $n \in \mathbb{N}$, the cardinalities of $\mathbb{N}^n$ and $\mathbb{N}$ are the same, one can encode an element of $\mathbb{N}^n$ as an element of $\mathbb{N}$. Thus it makes sense to talk about deciding subsets of $\mathbb{N}^n$. Let $\Omega \subseteq \mathbb{N}^m$ be a set. A tile system $\mathbb{S} = \langle T, g, \tau \rangle$ *nondeterministically decides* $\Omega$ with identifier tile $r \in T$ iff for all $\boldsymbol{a} \in \mathbb{N}^m$, there exists a seed configuration $S$ that encodes $\boldsymbol{a}$ and for all final configurations $F$ that $\mathbb{S}$ produces on $S$, $r \in F(\mathbb{Z}^2)$ iff $\boldsymbol{a} \in \Omega$, and there exists at least one final configuration $F$ with $r$ attached. In other words, the *identifier* tile $r$ attaches to one or more of the nondeterministic executions iff the seed encodes an element of $\Omega$. I call the set of tiles used to encode the input $\Gamma$.

I have given informal definitions to assist the reader in understanding the system I discuss in this paper and I refer the reader to [7] for more formal definitions.

## 3 Solving 3-SAT Efficiently with Tiles

Implementing algorithms in the tile assembly model is not unlike implementing algorithms using Turing machines, or programming using a low-level language, such as assembly. The complexity of that process has led to only simple algorithms implemented into tile systems. Here, I propose the tile system $\mathbb{S}_{FS}$ ($FS$ stands for "fast satisfiability") that implements the $O^\star(1.8393^n)$ algorithm for solving 3-$SAT$. The algorithm's running time implies that $\mathbb{S}_{FS}$ will create $O^\star(1.8393^n)$ distinct assemblies to decide an $n$-variable formula.

$\mathbb{S}_{FS}$ is a combination of several subsystems, each with a distinct job. Figure 1 shows the general placement of the distinct subsystems on a 2-D grid. The overall system will construct a right triangle, starting from region I, which encodes a Boolean formula $\phi$. Region II will examine the eastmost clause of $\phi$ and determine which literals have not been assigned a value (at the start of the computation, there will always be 3 unassigned literals in each clause of a 3-$SAT$ formula, but as the algorithm makes assignment decisions, clauses may have fewer such literals). Region III will make the nondeterministic decision on what

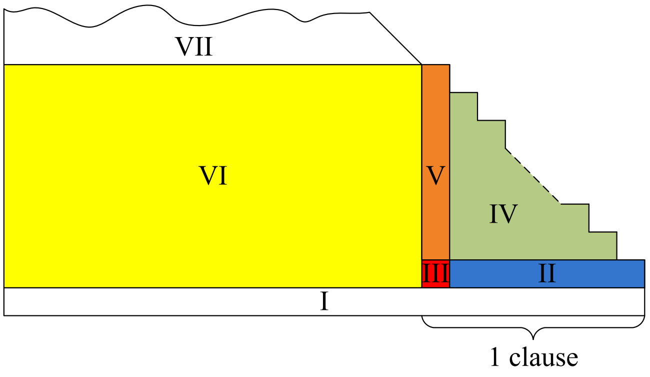

**Fig. 1.** A schematic of the seven regions $\mathbb{S}_{FS}$ will use to decide 3-$SAT$.

assignments to make regarding the unassigned literals in the eastmost clause. Region IV will prepare the literals of the eastmost clause to be assigned by the decision made in region III, and region V will make those assignments. Region VI will simplify the rest of $\phi$ based on those assignments. At the top of region VI, the simplified $\phi$, with one fewer clause, will emerge to serve as the input (like region I) for the remainder of the computation in region VII. That is, $\mathbb{S}_{FS}$ will operate recursively in Region VII on the simplified $\phi$.

The rest of this section demonstrates that $\mathbb{S}_{FS}$ decides 3-$SAT$ requiring only $O^\star(1.8393^n)$ distinct assemblies. Due to space limitations, I am unable to include the appropriate details and proofs here.

## 3.1 Notations and Definitions

Let $\phi$ be a Boolean formula. Let $n$ be the number of distinct variables and $m$ be the number of clauses in $\phi$. A literal over a variable $x$ is an element of $\{x, \neg x\}$. As is common, I assume that no clause of $\phi$ contains more than one literal over the same variable.

For all $m$, for all $n$, for all $n$-variable, $m$-clause 3CNF Boolean formulae $\phi$, $\phi$ is a *general* 3CNF Boolean formula iff each of the three literals of each clause either is identically a variable, is the negation of a variable, or is represented by $TRUE$ or $FALSE$ and no pair of literals within each clause are over the same variable.

The tile system $\mathbb{S}_{FS}$ will operate at temperature 2, and will use a fairly straightforward strength function $g_{FS}$ over the set of binding domains $\Sigma_{FS} = \{null$, $\mathsf{t}$, $\mathsf{bt}$, $\mathsf{bbt}$, $\mathsf{ft}$, $\mathsf{fft}$, $\mathsf{fbt}$, $\mathsf{bft}$, $\mathsf{T}$, $\mathsf{F}$, @, @$^\star$, $\mathsf{0}$, $\mathsf{1}$, $\mathsf{0t}$, $\mathsf{1t}$, $\mathsf{0f}$, $\mathsf{1f}$, $\mathsf{x}$, $\neg\mathsf{x}$, $\mathsf{x}^\star$, $\neg\mathsf{x}^\star$, $\mathsf{c}$, #, 0#, 1#, #f, #t, ::, 0:, 1:, 2:, 3:, 0:$^\star$, 1:$^\star$, 2:$^\star$, 1:1, 1:1$^\star$, 2:1, 2:1$^\star$, 3:1, 2:2, 2:2$^\star$, 3:2, 2:12, 2:12$^\star$, 3:12, $^{\star\star}$, $^\star$, $|\}$. For the most part, $g_{FS}$ will match two identical binding domains to 1, and two different binding domains to 0, with a few special wildcard binding domains that will map to 1 even with some unmatching domains. In other words, all attachments are either strength 0 or 1, as described in Figure 2.

| | Intuitively, $g_{FS}$ is such that: | Formally, $g_{FS}\colon \Sigma_{FS} \times \Sigma_{FS} \to \{0,1\}$ such that: |
|---|---|---|
| 0. | *null* does not attach to anything, | for all $\sigma \in \Sigma_{FS}$, $g_{FS}(null, \sigma) = g_{FS}(\sigma, null) = 0$, |
| 1. | every binding domain attaches to itself, | for all $\sigma \in \Sigma_{FS} \setminus \{null\}$, $g_{FS}(\sigma, \sigma) = 1$, |
| 2. | # attaches to 0, 1, x, ¬x, 1f, 1t, 0f, 0t, T, and F, | for all $\sigma \in \{0, 1, \mathsf{x}, \neg\mathsf{x}, \mathsf{1f}, \mathsf{1t}, \mathsf{0f}, \mathsf{0t}, \mathsf{T}, \mathsf{F}\}$, $g_{FS}(\#, \sigma) = g_{FS}(\sigma, \#) = 1$, |
| 3. | :: attaches to 0:, 1:, 2:, 1:1, 2:1, 2:2, and 2:12, | for all $\sigma \in \{0{:}, 1{:}, 2{:}, 1{:}1, 2{:}1, 2{:}2, 2{:}12\}$, $g_{FS}({::}, \sigma) = g_{FS}(\sigma, {::}) = 1$, |
| 4. | #f attaches to 0f, 1f, and F, | for all $\sigma \in \{\mathsf{0f}, \mathsf{1f}, \mathsf{F}\}$, $g_{FS}(\#\mathsf{f}, \sigma) = g_{FS}(\sigma, \#\mathsf{f}) = 1$, |
| 5. | #t attaches to 0t, 1t, and T, | for all $\sigma \in \{\mathsf{0t}, \mathsf{1t}, \mathsf{T}\}$, $g_{FS}(\#\mathsf{t}, \sigma) = g_{FS}(\sigma, \#\mathsf{t}) = 1$, |
| 6. | 0# attaches to 0, 0f, and 0t, | for all $\sigma \in \{0, \mathsf{0f}, \mathsf{0t}\}$, $g_{FS}(0\#, \sigma) = g_{FS}(\sigma, 0\#) = 1$, |
| 7. | 1# attaches to 1, 1f, and 1t, | for all $\sigma \in \{1, \mathsf{1f}, \mathsf{1t}\}$, $g_{FS}(1\#, \sigma) = g_{FS}(\sigma, 1\#) = 1$, |
| 8. | @⋆ attaches to @ and ⋆, | for all $\sigma \in \{@, \star\}$, $g_{FS}(@^\star, \sigma) = g_{FS}(\sigma, @^\star) = 1$, |
| 9. | and no other pairs of binding domains attach. | and for all other pairs $\sigma, \sigma' \in \Sigma_{FS}$, $g_{FS}(\sigma, \sigma') = g_{FS}(\sigma', \sigma) = 0$. |

**Fig. 2.** The strength function $g_{FS}$.

## 3.2 Clause Examination (Region II)

In this Section, I define the tile system $\mathbb{S}_{EXAM}$, which will become the part of $\mathbb{S}_{FS}$ that will operate in region II, as denoted in Figure 1. Since $\mathbb{S}_{FS}$ will operate on the first clause to fill in regions II through VI, and then recurse on the remaining simplified formula in region VII, for each clause there is a distinct copy of each region.

The goal of $\mathbb{S}_{EXAM}$ is to examine the first (eastmost) clause in the formula for the number of unassigned literals. Figure 3(a) shows the 37 tiles of $T_{EXAM}$ that perform this examination. I define the function $p$ that maps clauses of general 3CNF Boolean formulae to binding domains. Let $c$ be a clause of a general 3CNF Boolean formula. Then, if $c$ contains the literal $TRUE$, then $p(c) = \mathsf{T}$; otherwise, the value of $p(c)$ is defined by Figure 4. $\mathbb{S}_{EXAM}$ will attach just to the north of an encoding of clause $c$ and will propagate that encoding one row north and make the west binding domain of the westmost tile be the value of $p(c)$.

## 3.3 Assignment Selection (Region III)

In this Section, I define the tile system $\mathbb{S}_{SELECT}$, which will become the part of $\mathbb{S}_{FS}$ that will operate in region III, as denoted in Figure 1.

The goal of $\mathbb{S}_{SELECT}$ is to nondeterministically select an assignment over the variables of the eastmost clause just as the $O^\star(1.8393^n)$ algorithm would. Thus, if $\mathbb{S}_{EXAM}$ finds that the clause has three unassigned literals, $\mathbb{S}_{SELECT}$ will pick either (1) the first literal to be true, or (2) the first literal to be false and the

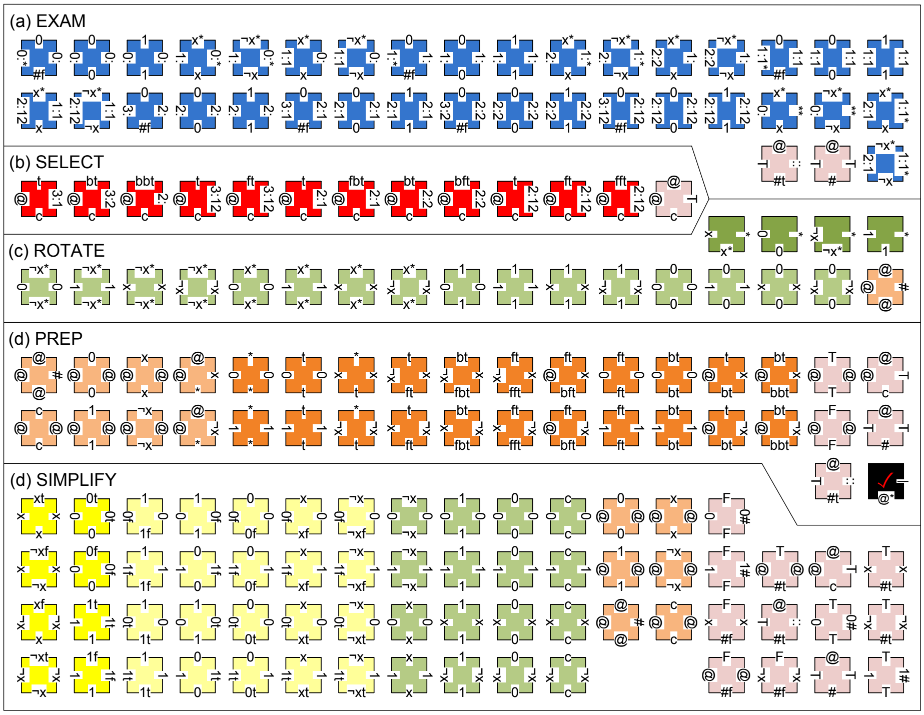

**Fig. 3.** Tiles of $T_{EXAM}$ (a), $T_{SELECT}$ (b), $T_{ROTATE}$ (c), $T_{PREP}$ (d), and $T_{SIMPLIFY}$ (e). Together, these sets form $T_{FS}$ with 147 distinct tiles.

second to be true, or (3) the first two literals to be false and the third to be true. Alternatively, if $\mathbb{S}_{EXAM}$ finds that the clause has its first literal already assigned and the other two unassigned, $\mathbb{S}_{SELECT}$ will pick to ignore the first literal and either (1) the second literal to be true, or (2) the second literal to be false and the third to be true. And so on.

Figure 3(b) shows the 13 tiles of $T_{SELECT}$ that perform the assignment selection. $\mathbb{S}_{SELECT}$ will attach just to the west of the westmost tile attached by $\mathbb{S}_{EXAM}$ and nondeterministically select one of the possible assignments in $ac(c)$ as that tile's north binding domain.

## 3.4 Clause Rotation (Region IV)

In this Section, I define the tile system $\mathbb{S}_{ROTATE}$, which will become the part of $\mathbb{S}_{FS}$ that will operate in region IV, as denoted in Figure 1.

The goal of $\mathbb{S}_{ROTATE}$ is to rotate a horizontally positioned clause encoding to be vertically positioned. This rotation later allows $\mathbb{S}_{SIMPLIFY}$ to simplify the formula. $\mathbb{S}_{ROTATE}$ will present the clause encoded in the north binding domains of part of its seed as the west binding domains of the completed right triangle. Figure 3(c) shows the 21 tiles of $T_{ROTATE}$ that perform the rotation.

| $c$'s literals | | | $p(c)$ | $ac(c)$ |
|---|---|---|---|---|
| first | second | third | | |
| *FALSE* | *FALSE* | *FALSE* | 3: | {F} |
| *FALSE* | *FALSE* | unassigned | 3:1 | {bbt} |
| *FALSE* | unassigned | *FALSE* | 3:2 | {bt} |
| *FALSE* | unassigned | unassigned | 3:12 | {bt, bft} |
| unassigned | *FALSE* | *FALSE* | 2: | {t} |
| unassigned | *FALSE* | unassigned | 2:1 | {t, fbt} |
| unassigned | unassigned | *FALSE* | 2:2 | {t, ft} |
| unassigned | unassigned | unassigned | 2:12 | {t, ft, fft} |

**Fig. 4.** For every clause $c$ of a general 3CNF Boolean formula, $p(c) = \mathsf{T}$ and $ac(c) = \{@\}$ if $c$ contains the literal *TRUE*, and otherwise, the values of $p(c)$ and $ac(c)$ are defined by this table. The goal of the $\mathbb{S}_{EXAM}$ system will be to produce, on the west side of the westmost tile in region II, the value $p(c)$ of the examined clause, and the goal of the $\mathbb{S}_{SELECT}$ system will be to produce one of the elements of $ac(c)$ on the north side of region III.

## 3.5 Assignment Preparation (Region V)

In this section, I define the tile system $\mathbb{S}_{PREP}$, which will become the part of $\mathbb{S}_{FS}$ that will operate in region V, as denoted in Figure 1.

The goal of $\mathbb{S}_{PREP}$ is to turn the assignment selected by $\mathbb{S}_{SELECT}$ into up to three literals that evaluate to *TRUE*. In other words, to apply the assignment to the clause. This application of the assignment is the final preparation before $\mathbb{S}_{SIMPLIFY}$ can simplify the formula. $\mathbb{S}_{PREP}$ will present the up to three literals as the west binding domains of the column in which it operates. Figure 3(d) shows the 36 tiles of $T_{PREP}$.

## 3.6 Formula Simplification (Region VI)

In this Section, I define the tile system $\mathbb{S}_{SIMPLIFY}$, which will become the part of $\mathbb{S}_{FS}$ that will operate in region VI, as denoted in Figure 1.

The goal of $\mathbb{S}_{SIMPLIFY}$ is to simplify the formula by replacing instances of the up to three literals prepared by $\mathbb{S}_{PREP}$ with *TRUE* and negations of those literals with *FALSE*. $\mathbb{S}_{SIMPLIFY}$ will present an encoding of the simplified formula as the north binding domains of the rectangle in which it operates. Figure 3(e) shows the 63 tiles of $T_{SIMPLIFY}$ that perform the simplification.

## 3.7 Solving 3-SAT

Thus far, I have described five tile systems: $\mathbb{S}_{EXAM}$, $\mathbb{S}_{SELECT}$, $\mathbb{S}_{ROTATE}$, $\mathbb{S}_{PREP}$, and $\mathbb{S}_{SIMPLIFY}$ that I intend to use to solve 3-*SAT*. These systems will operate in regions II, III, IV, V, and VI in Figure 1, respectively. The tile system $\mathbb{S}_{FS}$ combines these five systems to select a truth assignment for the first (eastmost) clause of a Boolean formula $\phi$, simplify the rest of $\phi$ based on that assignment, and recurse (in region VII) on the simplified $\phi$ with one fewer clause.

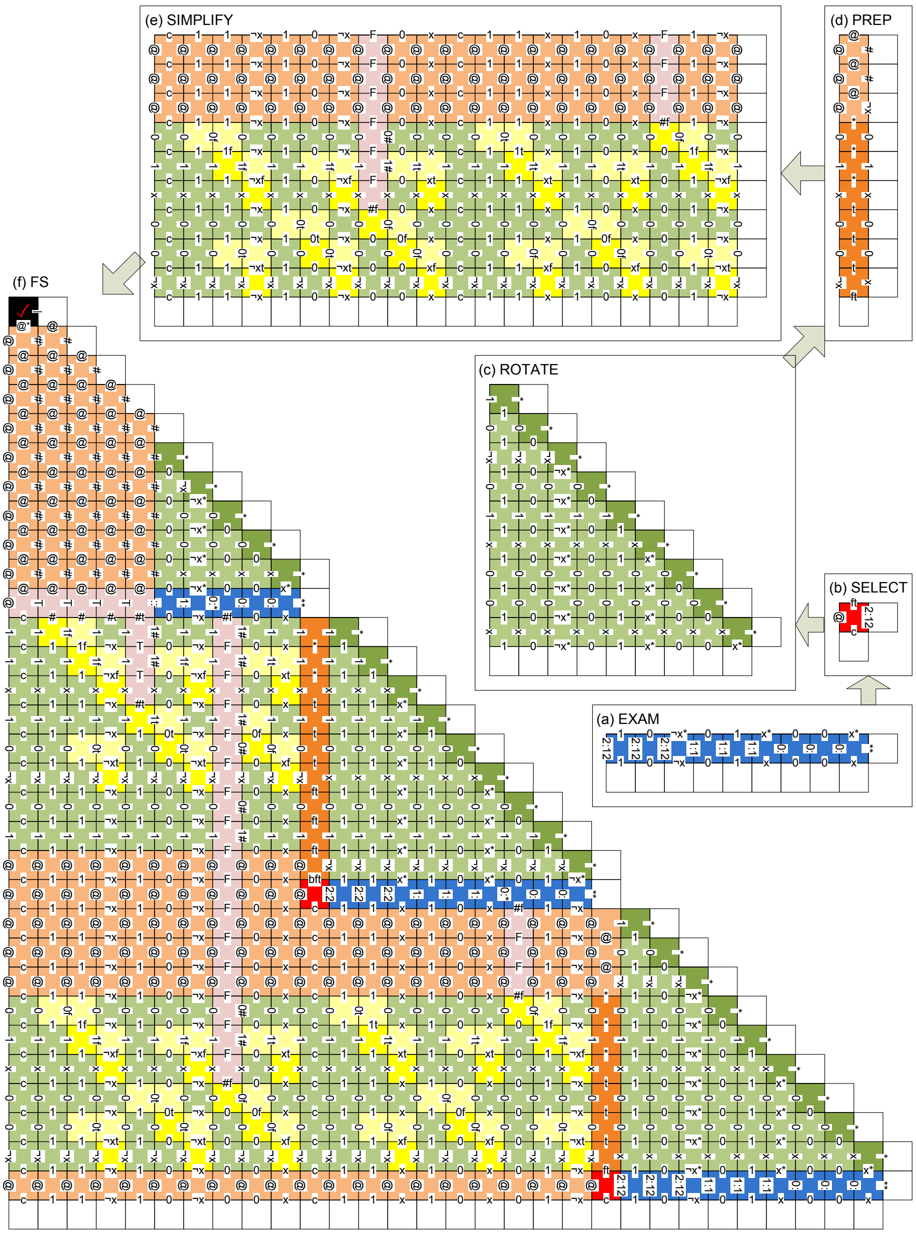

**Fig. 5.** Example executions of $\mathbb{S}_{EXAM}$ (a), $\mathbb{S}_{SELECT}$ (b), $\mathbb{S}_{ROTATE}$ (c), $\mathbb{S}_{PREP}$ (d), $\mathbb{S}_{SIMPLIFY}$ (e), and $\mathbb{S}_{FS}$ (f). $\mathbb{S}_{FS}$ operates on the Boolean formula $\phi = (\neg x_3 \vee \neg x_2 \vee x_0) \wedge (x_3 \vee x_2 \vee \neg x_1) \wedge (\neg x_2 \vee x_1 \vee x_0)$. The clear tiles are parts of the seed and the shaded tiles are computational. This $\mathbb{S}_{FS}$ execution nondeterministically selects the assignment $x_0 = \mathit{FALSE}$, $x_1 = \mathit{TRUE}$, $x_2 = \mathit{FALSE}$, and $x_3 = \mathit{TRUE}$; because that assignment satisfies $\phi$, the black ✓ tile attaches in the northwest corner.

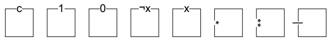

**Fig. 6.** The 8 tiles of $\Gamma_{FS}$ used to encode inputs to $\mathbb{S}_{FS}$.

$\mathbb{S}_{FS}$ will nondeterministically create only $O^\star(1.8393^n)$ distinct assemblies to decide whether $\phi$ is satisfiable. Note that there are 147 distinct tiles that $\mathbb{S}_{FS}$ uses (the distinct tiles of Figure 3), and that each nondeterministic assembly assembles in time linear in the size of the input.

I will use the 8 tiles in $\Gamma_{FS}$, shown in Figure 6, to encode the input $\phi$. Informally, I will encode the formula's literals in row 0, such that the literals of each clause are together and place the special clause tile to the west of each clause. I will place the tiles with $^{\star\star}$ and $^\star$ west binding domains on the diagonal to the north and west of the $\phi$, and I will place the tile with $|$ west binding domain as the northmost and westmost tile in the diagonal. The clear (unshaded) tiles in Figure 5(f) show the seed $S_{FS\phi}$ that encodes the 3-variable 3-clause Boolean formula $(\neg x_3 \vee \neg x_2 \vee x_0) \wedge (x_3 \vee x_2 \vee \neg x_1) \wedge (\neg x_2 \vee x_1 \vee x_0)$. The rest of Figure 5(f) shows a sample execution of $\mathbb{S}_{FS}$ on that seed. This execution assigns $x_0 = \mathit{FALSE}$ and $x_1 = \mathit{TRUE}$ in the first recursive step of the algorithm and then assigns $x_2 = \mathit{FALSE}$ and $x_3 = \mathit{TRUE}$ in the second step. In the third step, the algorithm finds that the third clause is already satisfied. Because this particular execution selected an assignment that satisfied the entire formula, the black ✓ tile is attached in the northwest corner.

**Theorem 1.** *Let $T_{FS} = T_{EXAM} \cup T_{SELECT} \cup T_{ROTATE} \cup T_{PREP} \cup T_{SIMPLIFY}$. Then $\mathbb{S}_{FS} = \langle T_{FS}, g_{FS}, 2\rangle$ nondeterministically decides 3-SAT with the black ✓ tile from $T_{PREP}$ as the identifier tile. Further, for all $n$-variable Boolean formula $\phi$, $\mathbb{S}_{FS}$ can nondeterministically form only $O^\star(1.8393^n)$ distinct assemblies.*

## 4 Contributions

I have presented a novel tile system $\mathbb{S}_{FS}$ that solves 3-*SAT* by nondeterministically creating $O^\star(1.8393^n)$ assemblies in parallel, for an $n$-variable Boolean formula. Each assembly assembles in time linear in the input size, explores some truth assignment, and attaches a special ✓ tile iff that assignment satisfies the formula. $\mathbb{S}_{FS}$ uses $147 = \Theta(1)$ distinct tile types. In some sense, $\mathbb{S}_{FS}$ implements the most complex algorithm using tiles to date. As a result, it helps bridge the gap between theoretical explorations of self-assembly, which require large tilesets to implement complex algorithms, and experimental endeavors, which have been able to combine up to 20 distinct tiles in a single experiment. Further, existing tile-inspired distributed software systems [8] can leverage $\mathbb{S}_{FS}$ directly to reduce their computational time requirements.

## Acknowledgments

This work was sponsored in part by the National Science Foundation, under Grant #0937060 to the Computing Research Association for the CIFellows Project.

## References

1. Abelson, H., Allen, D., Coore, D., Hanson, C., Homsy, G., Knight Jr., T.F., Nagpal, R., Rauch, E., Sussman, G.J., Weiss, R.: Amorphous computing. Communications of the ACM 43(5), 74–82 (2000)
2. Adleman, L.: Towards a mathematical theory of self-assembly. Tech. Rep. 00-722, Department of Computer Science, University of Southern California, Los Angeles, CA (2000)
3. Barish, R., Rothemund, P.W.K., Winfree, E.: Two computational primitives for algorithmic self-assembly: Copying and counting. Nano Letters 5(12), 2586–2592 (2005)
4. Brun, Y.: Arithmetic computation in the tile assembly model: Addition and multiplication. Theoretical Computer Science 378(1), 17–31 (2007)
5. Brun, Y.: Nondeterministic polynomial time factoring in the tile assembly model. Theoretical Computer Science 395(1), 3–23 (2008)
6. Brun, Y.: Solving NP-complete problems in the tile assembly model. Theoretical Computer Science 395(1), 31–46 (2008)
7. Brun, Y.: Solving satisfiability in the tile assembly model with a constant-size tileset. Journal of Algorithms 63(4), 151–166 (2008)
8. Brun, Y., Medvidovic, N.: Preserving privacy in distributed computation via self-assembly. Tech. Rep. USC-CSSE-2008-819, Center for Software Engineering, University of Southern California (2008)
9. Kullmann, O.: Worst-case analysis, 3-SAT decision and lower bounds: Approaches for improved SAT algorithms. DIMACS Series in Discrete Mathematics and Theoretical Computer Science 35, 261–313 (1997)
10. Kullmann, O.: New methods for 3-SAT decisions and worst-case analysis. Theoretical Computer Science 223, 1–72 (1999)
11. Lagoudakis, M.G., LaBean, T.H.: 2D DNA self-assembly for satisfiability. DIMACS Series in Discrete Mathematics and Theoretical Computer Science 54, 141–154 (1999)
12. McLurkin, J., Smith, J., Frankel, J., Sotkowitz, D., Blau, D., Schmidt, B.: Speaking swarmish: Human-robot interface design for large swarms of autonomous mobile robots. In: Proceedings of the AAAI Spring Symposium, Stanford, CA, USA (March 2006)
13. Monien, B., Speckenmeyer, E.: Solving satisfiability in less than $2^n$ steps. Discrete Applied Mathematics 10(3), 287–296 (1985)
14. Rothemund, P.W.K., Papadakis, N., Winfree, E.: Algorithmic self-assembly of DNA Sierpinski triangles. PLoS Biology 2(12), e424 (2004)
15. Rothemund, P.W.K., Winfree, E.: The program-size complexity of self-assembled squares. In: Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC 2000), Portland, OR, USA, May 2000, pp. 459–468 (2000)
16. Schiermeyer, I.: Solving 3-satisfiability in less than $1.579^n$ steps. Computer Science Logic 702, 379–394 (1993)
17. Soloveichik, D., Winfree, E.: Complexity of self-assembled shapes. SIAM Journal on Computing 36(6), 1544–1569 (2007)
18. Wang, H.: Proving theorems by pattern recognition. II. Bell System Technical Journal 40, 1–42 (1961)

19. Winfree, E.: Algorithmic Self-Assembly of DNA. Ph.D. thesis, California Institute of Technology, Pasadena, CA, USA (June 1998)
20. Winfree, E.: Simulations of computing by self-assembly of DNA. Tech. Rep. CS-TR:1998:22, California Institute of Technology, Pasadena, CA, USA (1998)
21. Woeginger, G.J.: Exact algorithms for NP-hard problems: a survey. Combinatorial Optimization - Eureka, You Shrink! pp. 185–207 (2003)