7-2010

Algorithmic Temperature 1 Self-Assembly

Yunhui Fu
University of Texas-Pan American

Algorithmic Temperature 1 Self-assembly

A Thesis

By

Yunhui Fu

Submitted to the Graduate School of the
The University of Texas-Pan American
In partial fulfillment of the requirements for the degree of

Master of Science

July 2010

Major Subject: Computer Science

Algorithmic Temperature 1 Self-assembly

A Thesis

by

Yunhui Fu

COMMITTEE MEMBERS

Dr. Robert Schweller
Chair of Committee

Dr. Zhixiang Chen
Committee Member

Dr. Artem Chebotko
Committee Member

Dr. Yang Liu
Committee Member

July 2010

Abstract

Fu, Yunhui, Algorithmic Temperature 1 Self-assembly. Master of Science, July, 2010, 54 pp., 4 tables, 19 illustrations, references, 22 titles.

We investigate the power of the Wang tile self-assembly model at temperature 1, a threshold value that permits attachment between any two tiles that share even a single bond. When restricted to deterministic assembly in the plane, no temperature 1 assembly system has been shown to build a shape with a tile complexity smaller than the diameter of the shape. Our work shows a sharp contrast in achievable tile complexity at temperature 1 if either growth into the third dimension or a small probability of error are permitted. Motivated by applications in nanotechnology and molecular computing, and the plausibility of implementing 3 dimensional self-assembly systems, our techniques may provide the needed power of temperature 2 systems, while at the same time avoiding the experimental challenges faced by those systems.

Dedication

The completion of my graduate studies would not have been possible without the love and support of my family. My wife, and my baby, wholeheartedly inspired, motivated and supported me by all means to accomplish this degree.

Thank you for your love and patience.

Acknowledgments

Thanks to my advisor, Dr. Robert Schweller. Without him I never would even have gotten involved in this topic, much less figured any of it out, and his generous support has enabled me to continue with this project.

Table of Contents

Page

List of Tables

Page

List of Figures

Page

Chapter I

Overview

Imagine putting a pile of bricks together, shaking them, and then a house emerges. It's seems hard to implement, and yet it happens all the time at the micro-scale. Self-assembly, which is a wide-ranging phenomena in nature, is one of the main methods used to construct structures at the nanoscale. DNA molecules have been studied adequately in biology and chemistry. At the nanometer scale, DNA has ideal size and geometry-free functionality, allowing it to be engineered for multiple purposes. In particular, DNA is an ideal material for the implementation of self-assembly theory. Advances in biochemistry permit the the precise fabrication of complex DNA molecules, which in turn allows for the programming of DNA self-assembly systems to precisely form complex structures.

Previous Work

Inspired by the Seeman's DNA self-assembly [1] and Wang's dominoes [2], Winfree and his co-authors developed a DNA tiling system [3]. Researchers have designed techniques to fold DNA strands into special purpose shapes that can act as four sided building blocks or *tiles*. These tiles can then implement a theoretical self-assembly model known as the Wang tile self-assembly model. The goal of tile self-assembly theory is to design or program tile systems to assemble into specific shapes or patterns efficiently. The Wang tile self-assembly model is a powerful research tool for theoretical self-assembly.

Models

One of the most important theoretical models is the abstract Wang tile assembly model, or *abstract Tile Assembly Model* (aTAM). The aTAM model was first introduced by Winfree in his PhD thesis [4]. The four pairing ends of DNA tiles are presented as four sides of a square in 2D. Each of the sides has a specific glue type associated with it to be presented as a different nucleotide sequence. Each type of glue has a different bond strength, such as 0, 1, or 2 etc. The four sides of the square are denoted separately as north, east, south, and west. Two tiles may stick at a side where the glues are the same type (in implementation, the two nucleotide sequences are Watson-Crick complements) and the sum of glue strength exceeds some threshold temperature τ. Tile assembly starts from an initial tile called the *seed*, and continues by attaching copies of tile types to the growing seed one by one.

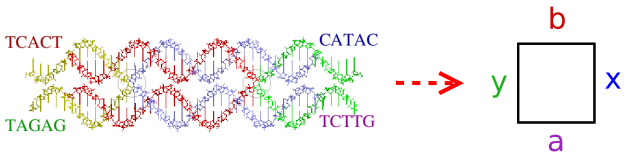


Figure 1. Mapping a DNA implementation to its corresponding Wang Tile

Subsequent work considered the following problem: What is the minimum number of distinct tile types, or *tile complexity*, required to uniquely assemble a $n \times n$ square? They show a lower bound of $\Omega(\frac{\log n}{\log \log n})$ tile types are required[5] as dictated by Kolmogorov complexity. The aTAM is a successful model to describe the relationship between tile complexity and a shape that is to be assembled.

The *kinetic Tile Assembly Model* (kTAM)[6, 7] was introduced to serve the purpose of experimental work. In this model tiles attach and detach probabilistically as a function of how strong glue bonds are. The rates are denoted by r_f and $r_{r,b}$, seperately. The forward (attach) rate and the reverse (detach) rate are given by:

$$r_f = k_f e^{-G_{mc}} \tag{1.1}$$

$$r_{r,b} = k_f e^{-bG_{se}} \tag{1.2}$$

Where k_f is a constant, and $G_{mc} > 0$ is measuring the tile concentration to the growth points. $G_{se} > 0$ measures the unit bond strength. The ratio $\tau = \frac{G_{mc}}{G_{se}}$ is comparable to that in the aTAM. Generally, a tile detachs faster than it attaches when $b < \tau$.

Theoretical analysis of errors is important because experimental work shows that the error rate of DNA self-assembly is high, ranging from 1% to 10%[8, 7]. There are three types of assembly errors in the kTAM model: growth errors, facet nucleation errors, and spontaneous nucleation errors[9].

The *growth errors* occur when a tile attaches by error to a position which should contain another suitable tile type. The error may not be recovered as growing positions are filled by new tile types. The *facet nucleation errors* refer to the error in which tiles attach to a position in which no tile types should be added. The *spontaneous nucleation errors* are thoes errors in which multiple groups of tiles grow in spite of the lack of seed tile.

Errors can be reduced by adjusting the parameters in the kTAM. By increasing the G_{mc} in the equation 1.1, the error rates decrese while speed of computation drops remarkably[6]. The relationship between the G_{mc} and G_{se} are shown in Figure 2.

Researcher have also developed some error correction coding schemes which try to resolve the error problems in the kTAM by changing the design of tile set. When considering the growth rate (r) and the errors (ϵ) of the tile system, the error could be reduced from $\epsilon \propto r^{1/2}$ to $\epsilon \propto r^{K/2}$ by using the *proofreading tile sets*[7] with size $K \times K$. *Snake tiles* proposed by Chen et. al [10] could reduce growth errors, and can also reduce the facet error which disturb proofreading tile sets. *Zig-zag Redundant Block* [11] has reported that it can dramatically reduce spurious nucleation rates. *Self-Healing Redundant Block* [9] was

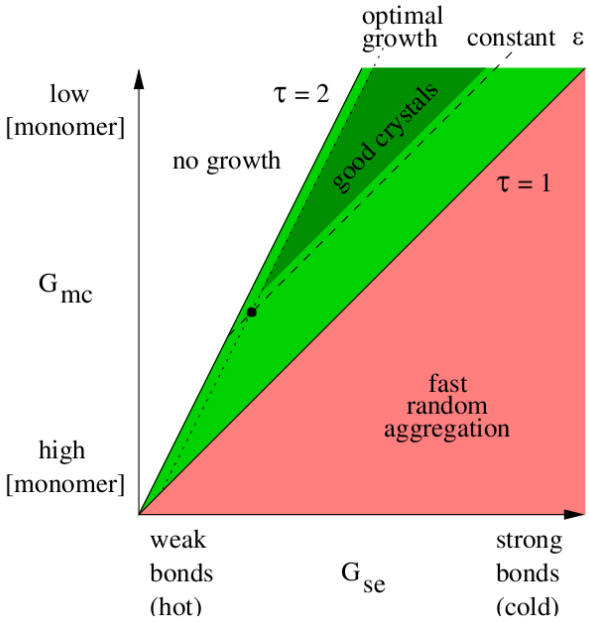


Figure 2. Size of assembly vs numbers of errors. Lines with slopes 1 and 2 represent $\tau = \frac{G_{mc}}{G_{se}} = 1$ and 2. The assembly performs well at $\tau = 2 - \theta$ for small θ and large G_{se}. (Adapted from References [7])

proposed to heal gross damage errors.

Related Work

Some recent work has been done in the area of proving lower bounds for temperature 1 self-assembly. Doty et. al [12] show a limit to the computational power of temperature 1 self-assembly for *pumpable* systems. Munich et. al [13] show that temperature 1 assembly of a shape requires at least as many tile types as the diameter of the assembled shape if no mismatched glues are permitted. In terms of positive results, Chandran et. al [14] consider the probabilistic assembly of lines with expected length n (at temperature 1) and achieve $O(\log n)$ tile complexity. Kao and Schweller [15] and Doty [16] use a variant of probabilistic self-assembly (at temperature 2) to reduce distinct tile type complexity. Demaine et. al [17] and Abel et. al [18] utilize steric hindrance to assist in the assembly and replication of shapes over a number of stages.

Simulator

Caltech's `xgrow` simulator was written by Winfree et. al[19] in C for the X Windows environment. It supports simulations for both abstract Tile Assembly Model (aTAM) and the kinetic Tile Assembly Model (kTAM). The program accept various options and can also change some parameters on the fly. The software can only support the models in 2D. The code is less maintained and has a limited GUI and parsing tile file.

The Iowa State University `TAS` tile assembly simulator was written by Patitz[20] in C++ with cross GUI platform library `wxWidgets` [1]. `TAS` can only support aTAM, but it can simulate the systems in 2D and 3D, and it offers a tile set editor. It provides a friendly GUI to support forwarding or rewinding of assembly growth, zooming, and inspection of tiles.

Contribution

We had some assumptions about the DNA self-assembly systems: temperature 2 or higher is required to carry out general-purpose computation in a tile assembly system, negative glue strengths have to be used in the tile system to get the general-purpose computation, etc. In contrast, we show that temperature 1 self-assembly in 3 dimensions, even when growth is restricted to at most 1 step into the third dimension, is capable of simulating a large class of temperature 2 systems, in turn permitting the simulation of arbitrary Turing machines and the assembly of $n \times n$ squares in near optimal $O(\log n)$ tile complexity. Further, we consider temperature 1 probabilistic assembly in 2D, and show that with a logarithmic scale up of tile complexity and shape scale, the same general class of temperature $\tau = 2$ systems can be simulated with high probability, yielding Turing machine simulation and $O(\log^2 n)$ assembly of $n \times n$ squares with high probability.

We further developed a software environment for our research projects. It includes the converter which converts arbitrary zig-zag tile sets from temperature $\tau = 2$ to $\tau = 1$; it

[1] `wxWidgets` homepage: http://www.wxwidgets.org.

also include the tile set generator that creates various self-assembly tile sets, such as parity systems with snake error correction tile sets[10], $n \times n$ squares, or even $n \times n \times n$ cubes in 3D, binary tile sets etc. There's also a simulator in the software package, which can support the simulation of models, such as aTAM, kTAM, 2hTAM etc. The 2hTAM is the multi-seed self-assembly model we are studying presently. We found no software systems which support multi-seed self-assembly before we finished our tile set simulation system.

Dissertation Outline

The thesis consists of two parts. The first part (Chapter II) describes the algorithms we developed in the research area of self-assembly at temperature 1. The second part (Chapter III) describes the internal structure of the simulation software for various self-assembly models.

Chapter II

Self-Assembly at Temperature 1

This chapter is the simplified edition of paper [21].

Overview

Many studies focus on tile self-assembly at temperature 2 (temperature denotes how many distinct bonds are required for two tiles to attach together) in 2 dimensions and some results were achieved. In this paper, we first describe a method to simulate arbitrary Turing Machines in the tile assembly system, and then we explore the power of temperature 1 self-assembly and compare it to the more well studied temperature 2 tile assembly [21]. We specifically consider temperature 1 self-assembly in 3D and find that, surprisingly, the assembly of $n \times n$ squares can be achieved using only $O(\log n)$ distinct tile types, and even simulation of arbitrary Turing machines is possible. This is very close to what is achievable at temperature 2. However, the best known temperature 1 constructions cannot beat $2n-1$ distinct tiles types to build a square, and have not been yet been able to simulate any sophisticated computation. Further, we get similar results after the introduction of random algorithms at temperature 1 in 2D. We show that probabilistic 2D temperature 1 self-assembly can generate $n \times n$ squares with high probability using only $O(\log^2 n)$ distinct tile types, and simulation of the Turing machine can be achieved efficiently with an arbitrarily small chance of making a computational error. Our research shows that self-assembly systems at temperature 1 have close to the same power of as temperature 2 self-assembly if either growth into the third dimension is permitted, or if probabilistic assembly is considered. This

result may yield practical advantages in a laboratory setting as a number of errors such as growth errors and facet roughening errors are unique to temperature 2 or higher systems.

Definitions

A ***tile system*** is a quadruple, $\langle T, G, \tau, s \rangle$, where T, G are all finite sets and

1. T is the set of tile types, each tile type t is a oriented square with the north, east, south and west glue types taken from an alphabet Σ.

2. G is glue strength function from $\Sigma \times \Sigma$ to $\mathbb{Z}^+$; it's assumed that $G(x, y) = G(y, x)$ for $x, y \in \Sigma$, and $\forall x \in \Sigma$, $G(\texttt{null}, \texttt{x}) = \texttt{0}$.

3. $\tau \in \mathbb{Z}^+$ is the threshold of the assembly, called *temperature*,

4. $s \in T$ is the start tile called the *seed* tile.

$|T|$ is referred to as the *tile complexity* of the system. In this paper we only consider temperature $\tau = 1$ and $\tau = 2$ systems.

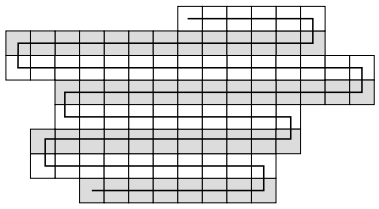


Figure 3. A zig-zag tile system alternates growth from left to right at each vertical level.

A ***zig-zag tile set*** is a tile assembly system in which the following tow properties hold:

1. The assembly sequence, which specifies the order in which tile types are attached to the assembly along with their position of attachment, is unique. For such systems denote the type of the i^{th} tile to be attached to the assembly as $type(i)$, and denote the coordinate location of attachment of the i^{th} attached tile by $(x(i), y(i))$.

2. For each i in the assembly sequence of a zig-zag system, if $y(i-1)$ is even, then either $x(i) = x(i-1)+1$ and $y(i-1) = y(i)$, or $y(i) = y(i-1)+1$. For odd $y(i-1)$, either $x(i) = x(i-1)-1$ and $y(i-1)$, or $y(i) = y(i-1)+1$.

A ***Turing machine*** is a 7-tuple[22], $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{accept} \neq q_{reject}$.

Simulate an Arbitrary Turing Machine with Tile Self-Assembly

Lemma 2.3.1. *A Turing Machine can be simulated by a zig-zag tile set in the aTAM model.*

Proof. We prove this lemma by constructing a zig-zag tile system. Refer to Figure 4 for details.

The set of all of the glues at the north or south sides of a tile represent the alphabet set Γ. The g_n of a tile is the output and g_s is the input in our Turing machine, so the action of placing one tile with different g_n and g_s on the supertile is one write operation, placing one tile with the same g_n and g_s is one read operation.

Basically, the tiles at the west growth direction are used for the main calculation, and the tiles on the east will be used to copy the glues from south to north.

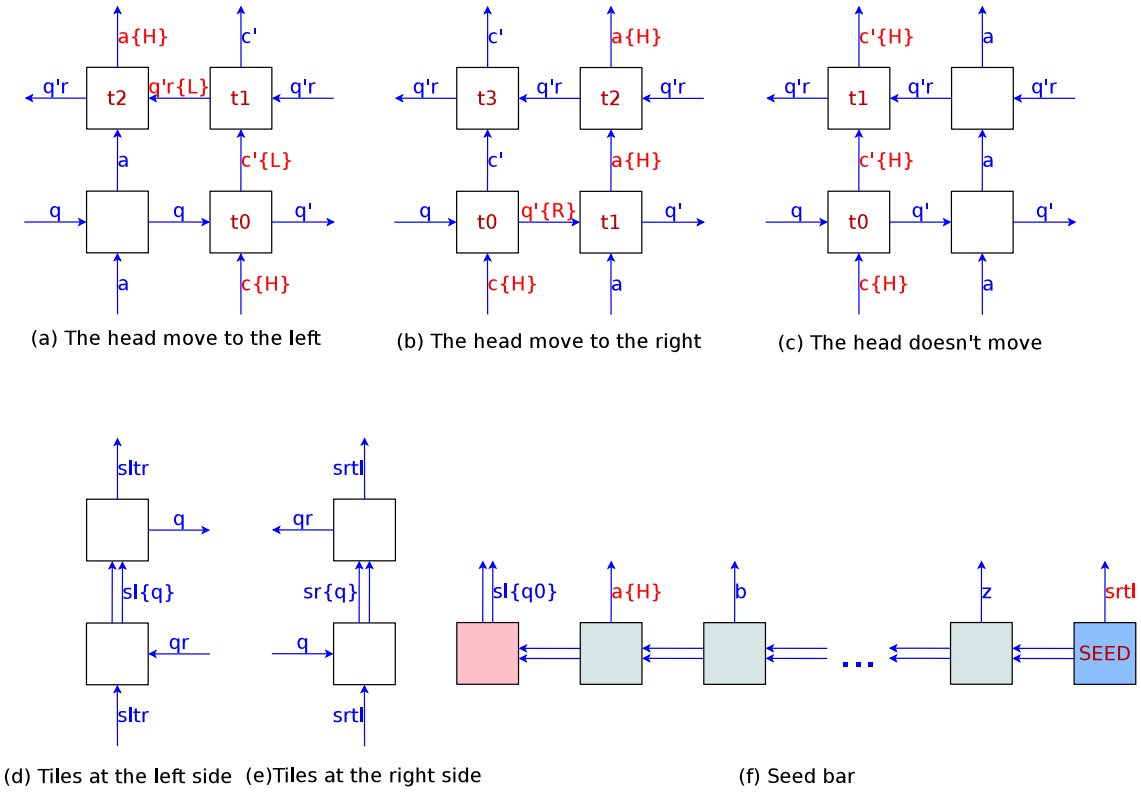


Figure 4. **Zig-Zag Tile Structure for Contructing Turing Machine**. **(a)** $\delta(q, c) = (q', c', L)$;**(b)** $\delta(q, c) = (q', c', R)$; **(c)** $\delta(q, c) = (q', c', \phi)$; **(d)** Left side; **(e)** Right side; **(f)** Seed bar: q_0 and input sequence $a, b, \ldots, z$

First, let's consider the equation $\delta(q, c) = (q', c', R)$. The details of this equation are shown in Figure 4b. The head of the TM is located at the c position and the current state is q. Then the record of the current position of tape is changed to c', the current state is changed to q', and head is moved to the right (R). Three types of the tiles, which are indicated by t_0, t_1, and t_2 in Figure 4b, are used to simulate this process. The glue notation at the south of the tile t_0 means that the head is at the column of tile t_0 and input alphabet is c. The glue notation (q) at the west of the tile t_0 means that the current state is q. The glues at the north and east indicate the output and next state separately. The east glue $(q'\{R\})$ of the tile t_0 indicates the next state (q') and carries the extra information to notify the next tile that the head is coming.

The equation $\delta(q, c) = (q', c', L)$ indicate that the head will move to left. Figure 4a

shows how to move the head to the left. Again, the head is at the column of tile t_0, the current state is q, and input is c. Then the tile t_0 outputs north glue $c'\{L\}$, and east glue q'. The tile t_1 will translate the glue information $(c'\{L\})$ to state information $(q'r\{L\})$ which contains the command to move left. At last, the tile t_2 is added to combine the header notation(H) with the glue of the current column.

If the head doesn't move after reading one character from tape, the corresponding tile is t_0 shown in Figure 4c. Figure 4d and Figure 4e show the tiles at the left and right sides of the zig-zag structure seperatly. Figure 4f shows the seed bar, which initializes the turing machine, including the start state q_0, the input sequence $(a, b, \ldots, z)$ etc. □

Algorithms

The detailed algorithm for simulating any Turing machine with a zig-zag tile set at temperature $\tau = 2$ in 2D is listed as Algorithm 1.

Deterministic Assembly in 3D at Temperature 1

In this section, we introduce how a zig-zag tile set of temperature 2 in 2D can be converted to a tile set of temperature 1 in 3D. For any temperature 2 zig-zag tile system in 2D, with σ_T denoting the set of distinct strength 1 north or south glue types occurring in T, the corresponding 3D temperature 1 tile system has only a $O(\log |\sigma_T|)$ tile complexity and scale factor.

First, we extend the model to assembly in 3D by adding two new glues, "up" and "down", to Wang cubes (which are called tiles in 2D tile systems). The cubes attach to the seed cubes on the sides of north, east, south, west, up, and down, if the total glue strength between cubes exceeds the threshold τ. We also introduce new notation in Figure 5 for the 3D tile assembly.

To assign tile types to the new temperature 1 assembly system, take all west growing

Algorithm 1: TURING-MACHINE-TO-ZIGZAG()

Input: q_0, the start state
str, the input string at the tape
δ, the state transition functions
Output: $T_{2d,2t}$, zig-zag tile set at temperature 2 in 2D which is converted from the Turing Machine

$Q \leftarrow \phi$; /* All of the states of the δ. */
$\Sigma \leftarrow \phi$; /* the input/output of the δ and the alphabet at the tape. */
foreach $f \in \delta$ **do**
  $Q \leftarrow Q \cup \{$ current state of $f\} \cup \{$ next state of $f\}$;
  $\Sigma \leftarrow \Sigma \cup \{$ input of $f\} \cup \{$ output of $f\}$;
foreach $c \in str$ **do** $\Sigma \leftarrow \Sigma \cup \{c\}$;
foreach $q \in Q$ **do**
  /* A four sided Wang tile denoted by the ordered quadruple $tile(\text{north}(t), \text{east}(t), \text{south}(t), \text{west}(t))$ */
  $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{tile(sl\{q\}, q_r, sltr, \phi)\}$; /* left, down */
  $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{tile(sltr, q, sl\{q\}, \phi)\}$; /* left, up */
  $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{tile(sr\{q\}, \phi, srtl, q)\}$; /* right, down */
  $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{tile(srtl, \phi, sr\{q\}, q_r)\}$; /* right, up */
  foreach $c \in \Sigma$ **do**
    $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{tile(c, q, c, q)\}$; /* tiles for copying */
    $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{tile(c, q_r, c, q_r)\}$; /* tiles for copying */
    $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{tile(c\{H\}, q_r, c\{H\}, q_r)\}$; /* auxiliary tiles: t_2 */
    $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{tile(c\{H\}, q, c, q\{R\})\}$; /* auxiliary tiles: t_1 */
    $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{tile(c\{H\}, q_r\{L\}, c, q_r)\}$; /* auxiliary tiles: t_2 */
foreach $f \in \delta$ **do**
  switch *head moving* **do**
    case *LEFT*
      $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{tile(f.output\{L\}, f.state_{out}, f.input\{H\}, f.state_{in})\} \cup \{tile(f.output, f.state_{out}r, f.input\{L\}, f.state_{out}r\{L\})\}$;
    case *RIGHT*
      $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{tile(f.output, f.state_{out}\{R\}, f.input\{H\}, f.state_{in})\}$;
    case *No moving*
      $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{tile(f.output\{H\}, f.state_{out}, f.input\{H\}, f.state_{in})\}$;
$T_{2d,2t} \leftarrow T_{2d,2t} \cup \{tile(srtl, \phi, \phi, gr)\}$; /* SEED */
$T_{2d,2t} \leftarrow T_{2d,2t} \cup \{tile(sl\{q_0\}, gl, \phi, \phi)\}$; /* SEED_L */
for $i \leftarrow 0$ **to** $(|str| - 1)$ **do**
  $c \leftarrow i$th alphabet of str ; /* Each of the tiles in Seed Bar */
  if $i = 0$ **then** $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{tile(c\{H\}, s_i, \phi, gl)\}$;
  else if $i = (|str| - 1)$ **then** $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{tile(c, gr, \phi, s_{i-1})\}$;
  else $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{tile(c, s_i, \phi, s_{i-1})\}$;
return $T_{2d,2t}$;

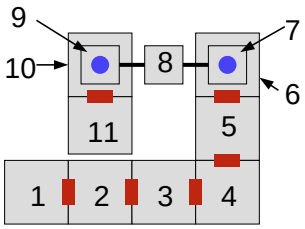


Figure 5. For 3 dimensional tile assemblies, we create figures with the notation depicted above. First, large tiles denote tiles placed in the $z = 0$ plane, while the smaller squares denote tiles placed in the $z = 1$ plane. The red connectors between bottom tiles denotes some unique glue shared by the tiles in the figure, as does the thin black line connecting tiles in the top plane. Blue circles denote a unique glue connecting the bottom of the top tile with the top of the tile below it. In this example, the tiles are numbered showing the implied order of attachment of tiles assuming the '1' tile is a seed tile.

tile types that have an east glue of type 'x' for some strength 1 glue 'x'. If there happens to be both east growing and west growing tile types that have glue 'x' as an east glue, first split all such tile types into two separate tiles, one for each direction.

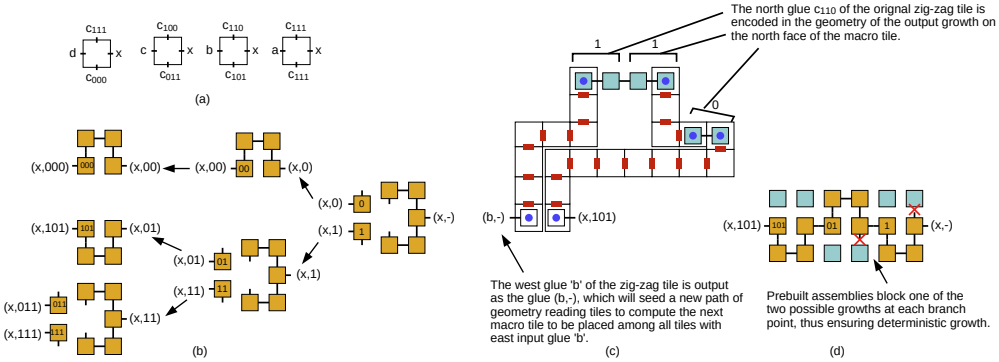


Figure 6. This tile set depicts how one tile from a collection of input tile types all sharing the same east glue x are simulated by combining strength 1 glues bonds with geometrical blocking to simulate cooperative binding and correctly place the correct macro tile.

For the set of west growing tile types with east glue x, a collection of tile types are added to the simulation set as a function of x and the subset of σ_T glues that appear on the south face of the collected tile types. The tile types added are depicted in Figure 6. In the example from Figure 6 there are 4 distinct tile types that share an east x glue type. As these tiles are west growing tile types, the temperature 2 simulation places each of these tiles using the cooperative bonding of glue x and glue c_{111} in the case of the right most tile

type. The east and south glue types of a west growing tile can be thought of as *input glues*, which uniquely specify which tile type is placed, in turn specifying two *output glues*, glue type a to the west and glue type c_{111} to the north in the case of the right most tile type. At temperature 1, we cannot directly implement this double input function by cooperative bonding as even a single glue type is sufficient to place a tile. Instead, we use glue type to encode the east input, and geometry of previously assembled tiles to encode the south input.

In more detail, the tile types specified in Figure 6 (b) constitute a nondeterministic chain of tiles whose possible assembly paths for a binary tree of depth log of the number of distinct south glue types observed in the tile set being simulated. In the given example, the tree starts with an input glue $(x, -)$. This glue knows the tile to its east has a west glue of type x, but has no encoding of what glue type is to the south of the macro tile to be placed. This chain of tiles nondeterministically places either a 0 or a 1 tile, which in turn continues growth along two separate possible paths, one denoted by glue type $(x, 0)$, and the other by glue type $(x, 1)$. By explicitly encoding all paths of a binary prefix tree ending with leaves for each of the south glues of the input tile types, the decoding tiles nondeterministically pick exactly one south glue type to pair with the east glue type x, and output this value as a glue specifying which of the 4 tile types should be simulated at this position.

Now, to eliminate the non-determinism in the decoding tiles, we ensure that the geometry of previouly placed tiles in the $z = 1$ plane is such that at each possible branching point in the binary tree chain, exactly 1 path is blocked, thus removing the non-determinism in the assembly as depicted in Figure 6 (d). This prebuilt geometry is guaranteed to be in place by the correct placement of the simulated macro tile placed south of the current macro tile. Once the proper tile type to be simulated is decoded, the 2 output values, a and c_{111} in the case of the rightmost tile type of Figure 6 (a), must be propagated west and north respectively. This is accomplished by the collection of tile types depicted in Figure 6 (c). Now that the north and west output glues have been decoded, this macro tile will assemble

a geometry of *blocking tiles* to ensure that a tile using this north glue as a south glue input will deterministically decode the correct glue binary string. In particular, pairs of tiles are placed in the plane $z = 2$ for each bit of the output binary string. The pair is placed to locations vertically higher for 1 bits than for 0 bits. The next row of macro tiles will then be able to decode this glue type encoded in geometry by applying a binary tree of decoder tiles similar to those shown in Figure 6 (b).

The complete conversion algorithm from a temperature 2 zig-zag system to a temperature 1 3D system has a large number of special cases. However, the example worked out in this proof sketch gets at the heart of the idea. The fully detailed conversion algorithm, with all cases detailed is described in the Appendix sec:detailconvertzigzag.

Probabilistic Assembly in 2D at Temperature 1

Similar to the temperature 1 3D tile system, the converstion from a temperature 2 2D zig-zag tile system to temperature 1 2D probabilistic tile system also focuses on the binary encoding of the north and south sides of the tile types. It use special structure to make sure the right tile types could be attached with high probability in 2D. The adding of tile types are depicted in Figure 7. The key idea is is to buffer the length of the geometric blocks to encode each bit by a factor of k for some desired parameter.

We can analyze the probability that a given 0 bit is correctly decoded by bounding the probability of flipping a coin k times and never getting a single tail. For a zig-zag tile system that makes r tile attachments to get to its terminal supertile state, a straightforward analysis shows that setting $k = O(\log(r) + \log(\sigma_T))$ yields a constant bound on the probability of failing even once throughout the entire assembly. From that we get the following theorem.

Theorem 2.5.1. *For any zig-zag tile system $\Gamma = \langle T, s, 2\rangle$ whose terminal assembly has size r tile positions, there exists a temperature 1 2D probabilistic tile system that simulates Γ without error with probability at least c for any constant $c < 1$. The scale_x of this system is*

$O(\log r + \log^2 \sigma_T)$, *the* $scale_y$ *is* 4, *and the tile complexity (total) is* $O(|T|(\log r + \log^2 \sigma_T))$, *yielding a* $O(\log r + \log^2 \sigma_T)$ *tile complexity scale factor.*

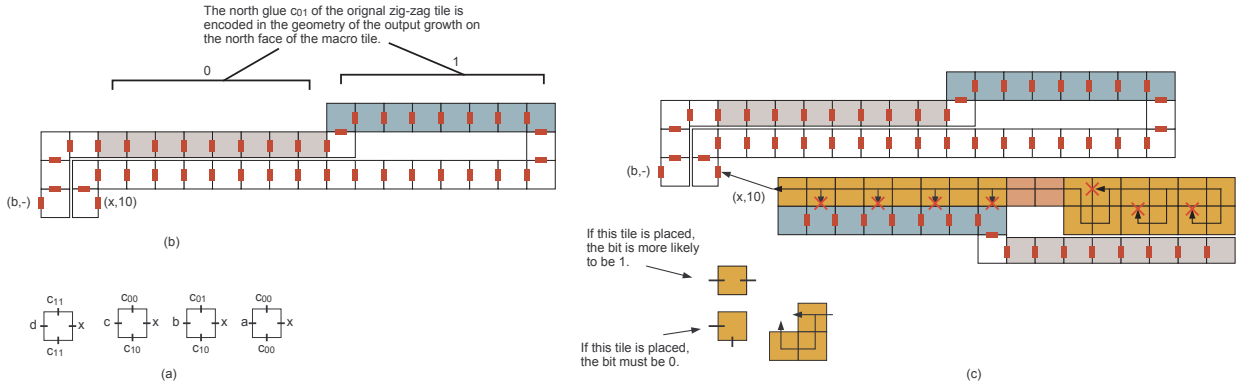


Figure 7. This tile set depicts the macro tiles used to transform a zig-zag system into a probabilistic 2D assembly system.

Temperature 1 Tile System Complexity

We can get the following results for the temperature 1 tile system. The proof is omitted and the details are in the full version paper[21].

Lemma 2.6.1. *For any* n, *there exists a 2D, temperature* $\tau = 2$ *zig-zag tile system, with north/south glueset* g *of size* $O(1)$, *that uniquely assembles a* $\log n \times n$ *rectangle. Further, this rectangle can be designed so that a unique, unused glue appears on the east side of the northeast placed tile (this allows the completed rectangle to seed a new assembly, as utilized in Theorem 2.6.2).*

Theorem 2.6.2. *For any* $n \geq 1$, *there exists a 3D temperature* $\tau = 1$ *tile system with tile complexity* $O(\log n)$ *that uniquely assembles an* $n \times n$ *square.*

Complexity Analysis of Zig-Zag Turing Machine

In the case of a $\tau = 2$, 2D zig-zag tile system, the number of the tile types for state transfer (t_0 in the Figure 4) is $|\delta|$. And the number of the auxiliary tile types for state

transfer (t_1 and t_2 in Figure 4a, 4b) is $|\Sigma| \times |Q| \times 2$ (move left) and $|\Sigma| \times |Q| \times 2$ (move right). The number of tile t_1 in Figure 4c does not need be included because it is counted in the auxiliary tile types (t_2 in Figure 4b). The other tile types are the tiles for transfering alphabet information, which have a count of $|Q| \times |\Sigma| \times 2$ (2 indicate the east direction and west direction of the growth).

Some special types of tiles in both sides of the zig-zag structure are needed to change the growth direction, costing $4|Q|$ tiles. If the length of the tape is n, then the number of the tile types for constructing the seed bar will cost $n + 2$ tiles (See Figure 4f).

The total tile types is $|T| = |\delta| + |\Sigma| \times |Q| \times 2 + |\Sigma| \times |Q| \times 2 + |Q| \times |\Sigma| \times 2 + 4|Q| + n + 2 = |\delta| + 6|Q||\Sigma| + 4|Q| + n + 2$. This yields and asymptotic upper bound of $O(|\delta| + |Q||\Sigma| + n)$ tile types. Considering that $|Q| \leq |\delta| \leq |Q|^2|\Sigma|^2$ and $n \leq |\Sigma|$, the complexity of tile types is $O(|\delta| + |Q||\Sigma| + n) = O(|\delta|)$.

Each of the state transfers costs two lines of the zig-zag structure (east direction line and west direction line above). Given an input tape and start state, if the number of the state transfers to be passed before it stops is r, then the lines of the tile structure will be $2r$. So the space used in calculation is $O(nr)$.

In the case of a $\tau = 1$, 3D zig-zag tile system, the length of each mapped tile depends on the length of binary encoding code of glues (The details of the algorithms are shown in Appendix A). If the number of glues of the 2D zig-zag tiles is G, then $G = O(|\delta|)$. The complexity of tile types in 3D is $O(|\delta| \log G) = O(|\delta| \log |\delta|)$, the complexity of space is $O(nr \log G) = O(nr \log |\delta|)$.

In the case of a $\tau = 1$, 2D probabilistic zig-zag tile system, the length of each mapped tile depend on the parameter K and the length of binary encoding code of glues (The details of the algorithms are shown in Appendix A). The parameter $K = O(\log r + \log \frac{1}{\epsilon})$ of the $\tau = 1$ 2D probabilistic zig-zag tile system is a constant that depends on the probability of correct archieving a result. The space complexity is $O(nrK \log G) = O(nr \log |\delta| the)$, and

the number of tile types is $O(|\delta| \log |\delta|)$.

Table 1. The Complexity of the Zig-Zag Turing Machine

	$\tau = 2$ **2D**	$\tau = 1$ **3D**	$\tau = 1$ **2D Prob.**
Space	$O(nr)$	$O(nr \log \|\delta\|)$	$O(nr \log \|\delta\|)$
Tiles	$O(\|\delta\|)$	$O(\|\delta\| \log \|\delta\|)$	$O(\|\delta\| \log \|\delta\|)$

$|\delta|$: the number of state functions;
n: the length of the tape (size of alphabet);
r: the number of the state transfers to be passed before it stop;

Conclusions and Future Work

We have developed automatic conversion software to generate temperature 1 3D or 2D systems given any input zig-zag tile set. We have further verified the correctness of our algorithms and converter software by implementing a number of tile systems such as Sierpinski tile sets and binary counters. We have run the simulations on Caltech's Xgrow simulator[19] and ours. The system performs well.

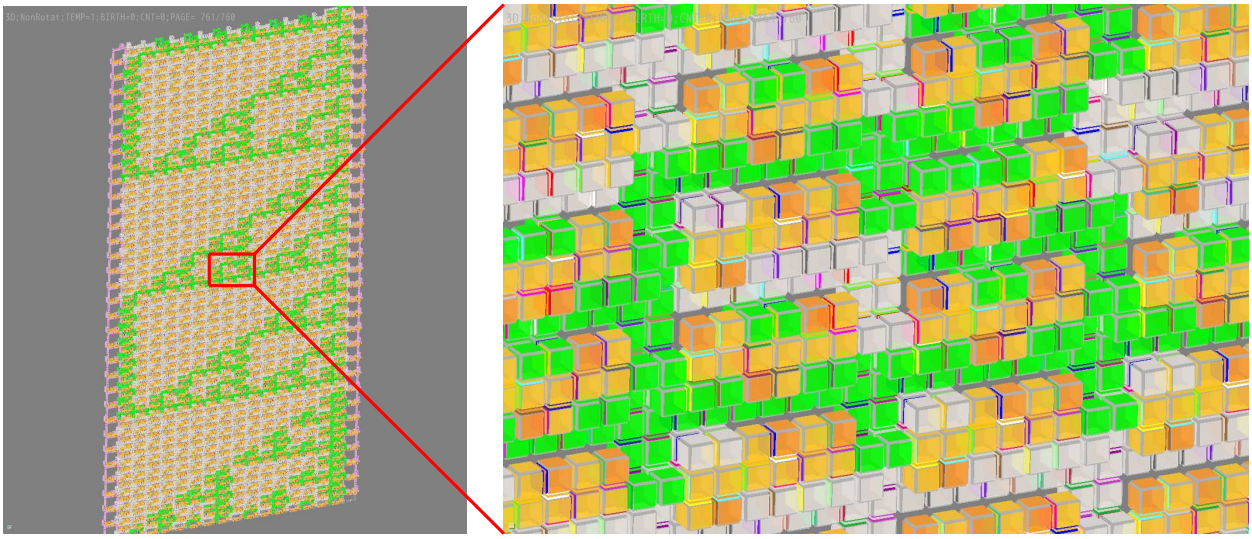


Figure 8. Snapshots of Sierpinski in 3D Zig-Zag Tile System

We are interested in experimentally comparing the fault tolerance of our temperature 1 probabilistic system with other fault tolerance schemes such as snaked proofreading systems.

We implement both systems with similar tile complexity constraints to see under what settings either system would begin to make errors. We have done preliminary simulations of the 2D systems in the kTAM with Caltech's Xgrow. Our test case was a simple parity checking tile system[10], and our metric was whether or not the correct parity was computed. Our preliminary tests used the input string "1000". The block size of the snaked system is 6×6, and the length of each bit(K) of probabilistic assembly tiles is 5. The glue strength of the probabilistic assembly tiles are changed to 2.

We used $G_{mc} = 13.6$, $G_{se} = 7.0$. Both tile systems outputted correct results. But as we increased the G_{se}, the performances of the two system diverged. The snaked system became unstable, facet errors occurred and the tile mismatches increased. In contrast, the probabilistic assembly tile set grew faster and the output of the system was still correct. We used the $\tau = \frac{G_{mc}}{G_{se}}$ ranging from 2 to 0.5, and the results show that the 2D probabilistic assembly system is more stable than the snaked system within the parameters we tested.

The comparison of the probabilistic and snaked systems is very preliminary because the two systems use different temperatures. We adjusted the glue strength of the probabilistic system to make it comparable with the snaked system. To more accurately compare the two systems, we need to investigate other methods and test sets, both for probabilistic and snaked systems.

Chapter III

Simulation Software

In this chapter, we first show the features of the simulator and tools we developed for our research, and then reveal the key algorithms of the software.

Design Specifications

The Supertile Simulation System is a simulator for self-assembly of DNA, where researchers can inspect the actions of different tile sets running in three types of models: abstract Tile Assembly Model (aTAM), kinetic Tile Assembly Model (kTAM) and two handed Tile Assembly Model(2hTAM). This software supports the simulation both in 2D and 3D, rendering in OpenGL[1]. It provides a convenient command line interface to support automatic simulations. Tile set creators and convertors are also provided, such as the squarecreator which create some types of square shape tile sets, and convzzg2s[2] which converts arbitrary zig-zag tile sets at temperature 2 to tile sets at temperature 1.

Core Objectives

1. The architecture of the simulator can support multiple self-assembly models, such as aTAM, kTAM, 2hTAM etc.

2. Supports 2D and 3D self-assembly, allows tile rotation or not.

[1]OpenGL-The Industry's Foundation for High Performance Graphics, http://www.opengl.org/

[2]This tool is a implementation of the algorithms described in the paper[21]

3. The core algorithms for 2hTAM. The algorithms should detect all of the posible binding sites of two supertiles. The binding sites are influenced by the geometrical shape of supertiles. The algorithms for 2hTAM's spliting supertile in higher temperature.

4. Supports saving result. The data file stores the tile types, glue type information, etc.

5. Supports data file formats of other simulators, such as TAS, xgrow, etc.

The Key Algorithms of the Simulation System

A X-Y coordinate system are used to record the position of each tile. The positive orientation of the Y axis is from bottom to up; the positive orientation of the X axis is from left to right. The value of axis start from 0.

An array were used to record all types and quantity of the supertiles created during the simulation. In each round, two types of supertile (could be same type) are picked out randomly and tested if they could mesh with each other. If it do, then the new supertile created will be stored in the array.

The tiles which are about to attach to the supertile have to be pass some tests. Two types of tests are very important in the simulation system. One is Equivalence Testing, and another one is Mesh Testing.

Equivalence Testing (ET) is used to test if two supetile is equal. The supertiles are placed at random orientation. The supertile can be rotated at most four times and compare with another supertile to check if the two supertils are equal. This type of testing is used in the statistic of the numbers of created supertiles.

Mesh Testing

Mesh Testing (MT) handle the test if two supertile could be engaged with each other at a given tempreture. We should consider the condition that one supertile should not enter the hollow in the center of other supertile.

In this paper The tested supertiles are named as T and B. X_T denotes the width of the supertile T rectangle area, and Y_T is the height. Accordingly, X_B denotes the width of the supertile B rectangle area, and Y_B is the height.

As we showed in the Figure 9, supertile B have a fix position in the X-Y coordinate plane, which is at the position (X_T, Y_T).

The orientation of one supertile is the rotatory times of the supertile from the origin state in the list table. The rotatory angle values are $0°, 90°, 180°$, and $270°$ respectively. The supertile T will be rotated four times and meshed with the supertile B.

Mesh Testing are described by two sub-testing, Sliding Testing and Plumb Testing. Both Sliding Testing and Plumb Testing are influenced by the geometry shape of supertiles.

Sliding Testing

Sliding Testing(ST) is the testing of combination probability between supertile T and B along the sides of two supertiles. Starting from two positions($(0, 0)$ and $(X_T + X_B, Y_T + Y_B)$) and along the outer edges of supertile B, the supertile T tries to mesh with supertile B.

At the start of Sliding Testing, the supertile T is placed at the edge of the supertile B, with only one tile side of the area adjoined. In each step of the ST, a Plumb Testing is applied in the testing. After the Plumb Testing, the supertile T will move to the next position at the same direction of Sliding Testing and do another Plumb Testing again. Each sides of supertile T will be applied one ST.

Plumb Testing

The Plumb Testing(PT) is used to find out all of the position that two supertiles can mesh with. The algorithm could be described by the Algorithm 2.

There're some other minor algorithms list below:

Rotate the supertile

This algorithm is used at the ET and MT. Supertiles have four positions after rotating in 2D.

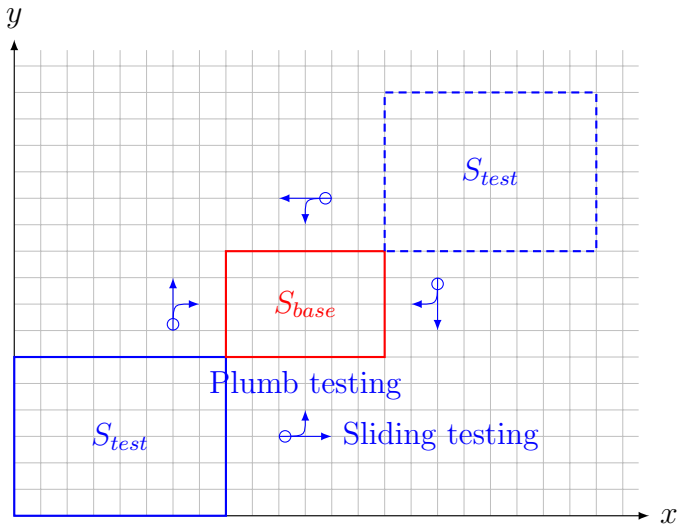


Figure 9. The slide test of supertiles S_{base} and S_{test}

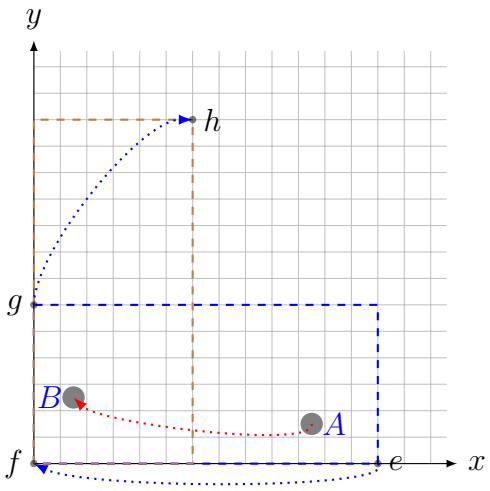


Figure 10. The rotation of the rectangle: from 0° to 90°; $A = (x_0, y_0) = (10, 1); B = (y_0, X_{max} - x_0 - 1) = (x_{90°}, y_{90°}) = (1, 2); (X_{max}, Y_{max}) = (13, 16)$

Algorithm 2: Plumb Testing

Input: (x_0, y_0), the start position
S_{base}, the base supertile
S_{test}, the test supertile
τ, the temperature of the system
BG_{pos}, the positions which are already visited
Output: L_{pos}, the set of postions which are suitable to combine two supertiles

$L_{pos} \leftarrow \phi$;
$STK_{pos} \leftarrow (x_0, y_0)$; `/* Push the start position to position stack */`
while $STK_{pos} \neq \phi$ **do**
  Pop one item from the top of stack STK_{pos} to (x, y) ;
  if $(x, y) \notin BG_{pos}$ **then**
    `/* It's not test at position` (x, y) `*/`
    $BL_{pos} \leftarrow \phi$; `/* The positions which are pushed to stack */`
    Get the overlap region of the two supertiles, S_{base} and S_{test} ;
    cnt[north] $\leftarrow$ 0; cnt[east] $\leftarrow$ 0; cnt[south] $\leftarrow$ 0; cnt[west] $\leftarrow$ 0 ;
    `/* Check each tile of the supertile` S_{test} `in the overlap region */`
    for *each tile* t_0 *of* S_{test} *in overlap region* **do**
      strength $\leftarrow$ 0 ;
      for *each side of* t_0 $s \in$ {*north,east,south,east*} **do**
        if *exist tile* t_1 *of* S_{base} *near the side* s *of* t_0 **then**
          if *the glue type of* t_1 *at the side opposite(s) is the same of glue* t_0 *at side* s **then**
            strength $\leftarrow$ strength + glue strength ;
            `/* Add 1 to the tile number of` S_{base} `which is abut with the tile of` S_{test} `in this direction */`
            cnt[s] $\leftarrow$ cnt[s] + 1 ;
    if *strength* $\geq \tau$ **then**
      $L_{pos} \leftarrow L_{pos} \cup \{(x, y)\}$;
    for *each* $s \in$ { *north,east,south,east* } **do**
      (x_2, y_2) is the position at the side of t_0 ;
      if *cnt(s)=0 and* $(x_2, y_2) \notin BG_{pos}$ *and* $(x_2, y_2) \notin BL_{pos}$ **then**
        `/* there are no tiles of Base, the Test supertile can be moved to this direction to detect the mesh */`
        Push the position at the side of t_0 to stack STK_{pos} ;
        `/* Set the position` (x_2, y_2) `as local accessed */`
        $BL_{pos} \leftarrow BL_{pos} + \{(x_2, y_2)\}$;
    `/* Set the position` (x, y) `as accessed */`
    $BG_{pos} \leftarrow BG_{pos} + \{(x, y)\}$;
return L_{pos};

Algorithm 3: Mesh Testing

Input: (x_0, y_0), the start position
S_{base}, the base supertile
S_{test}, the test supertile
τ, the temperature of the system
BG_{pos}, the positions which are already visited
Output: L_{pos}, the set of postions which are suitable to combine two supertiles
nextsupertile $\leftarrow S$;
repeat
  Map the nextsupertile to graph ; `/* The tiles are regarded as the node of the graph. The same glue type between two tiles is regarded as the edge between two nodes, and the glue strenth is the value of edge */`
  Apply min-cut on the graph ;
  if *value of min-cut* $< \tau$ **then**
    if *the graph is maped from the item of array AS* **then**
      Update the item with one sub-supertiles and put another sub-supertiles to the end of array AS ;
    else
      Put the two sub-supertiles to AS ;
      nextsupertile $\leftarrow$ first item of array AS ;
until *Reach to the end of array AS* ;
return AS;

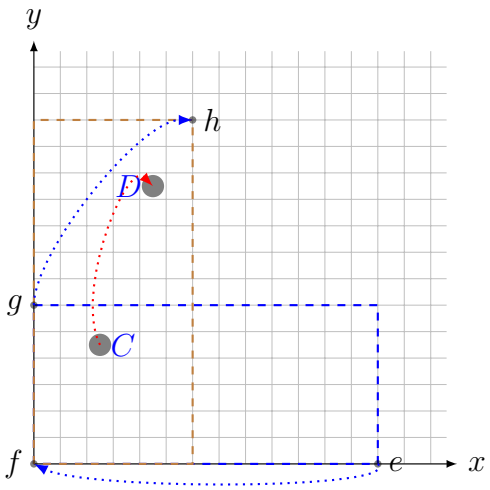


Figure 11. The rotation of the rectangle: from 180° to 270°; $C = (x_{180°}, y_{180°}) = (X_{max} - x_0 - 1, Y_{max} - y_0 - 1) = (2, 4)$; $D = (x_{270°}, y_{270°}) = (Y_{max} - y_0 - 1, x_0) = (4, 10)$;

We assume the width of supertile rectangle is X_{max}, the heigth is Y_{max}. The related position of one tile of the supertils is (x_0, y_0). After 90° of rotating, the position of the tile related to the new supertile will change to $(X_{90°}, Y_{90°}) = (y_0, X_{max} - 1 - x_0)$ and so on. The formulas are showed in Table 2.

The overlapped range of two supertile

When applying the Mesh Testing, the simulator only need to check the overlapped range of two supertiles in which no two tiles belong to different supertiles are overlaped.

As we showed in previous section, two supertiles, T and B, have the rectangle range (X_T, Y_T) and (X_B, Y_B) seperatly. To avoid negative values, the supertile B is placed at (X_T, Y_T). The supertile T can be placed (the top-left position (x_0, y_0) of the supertile T rectangle) at any position in the range of $([0, X_T + X_B), [0, Y_T + Y_B))$.

We noticed that the calculation of overlaped area of two supertiles is depend on which one is larger. For simplification, we'll discuss the overlaped range at the direction of X axis. We can use the same algoritm to apply to the direction of Y axis. See the Figure 12 and Figure 13 for the detail.

We use the absolute position to denote the position of the tiles. We can use the following formula to calculate the position of the tile related to each supertile:

$$x_t = P_{abs} - x_0$$

$$x_b = P_{abs} - X_T$$

Table 2. Rotation Formulas

Rotation	**New Position**	**Value**
0°	(x_0, y_0)	(x_0, y_0)
90°	$(X_{90°}, Y_{90°})$	$(y_0, X_{max} - 1 - x_0)$
180°	$(X_{180°}, Y_{180°})$	$(X_{max} - 1 - x_0, Y_{max} - 1 - y_0)$
270°	$(X_{270°}, Y_{270°})$	$(Y_{max} - 1 - y_0, x_0)$

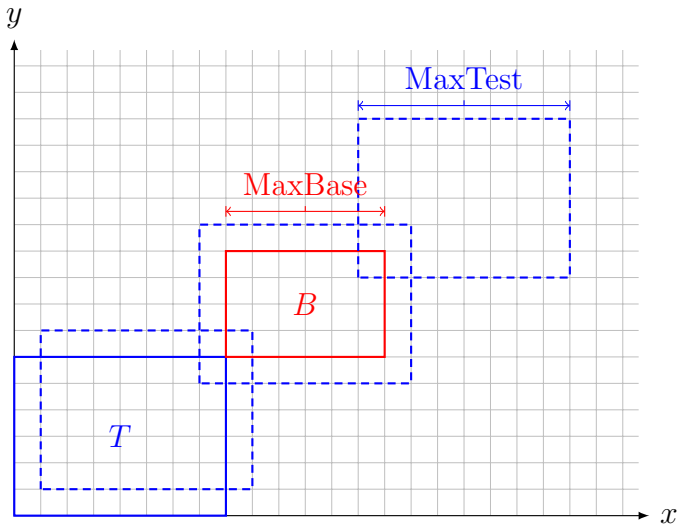


Figure 12. The overlap of the supertiles B and T (MaxBase $\leq$ MaxTest)

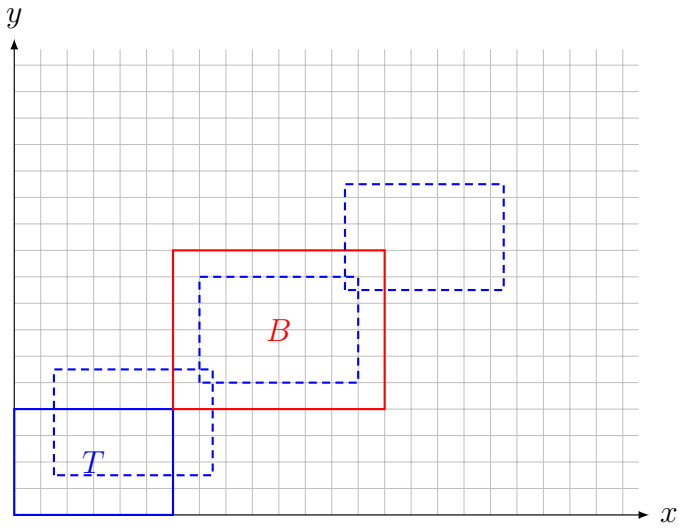


Figure 13. The overlap of the supertiles B and T (MaxBase $>$ MaxTest)

Table 3. The Absolute Position of the Overlaped Area

	Range of x_0	**Absolute Position(PX_{abs})**
Len_t<Len_b	$[0, X_B)$ $[X_B, X_T)$ $[X_T, X_T + X_B)$	$[X_T, X_T + x_0)$ $[X_T, X_T + X_B)$ $[x_0, X_T + X_B)$
Len_t>Len_b	$[0, X_T)$ $[X_T, X_B)$ $[X_B, X_T + X_B)$	$[X_T, X_T + x_0)$ $[x_0, X_T + x_0)$ $[x_0, X_T + X_B)$
	Range of y_0	**Absolute Position (PY_{abs})**
$Width_t$<$Width_b$	$[0, Y_B)$ $[Y_B, Y_T)$ $[Y_T, Y_T + Y_B)$	$[Y_T, Y_T + y_0)$ $[Y_T, Y_T + Y_B)$ $[y_0, Y_T + Y_B)$
$Width_t$>$Width_b$	$[0, Y_T)$ $[Y_T, Y_B)$ $[Y_B, Y_T + Y_B)$	$[Y_T, Y_T + y_0)$ $[y_0, Y_T + y_0)$ $[y_0, Y_T + Y_B)$

The Fragments of Assemblies

To support multi-temperature stage theory, we implementated the algorithm to split the supertiles when temperature increses. The base ideal of the algorithm is find out the 'weak' links between any two sub-supertiles. We have implemented the algorithm by different methods, and at last, we found that the min-cut algorithm is suitable to resolve the problem. The Algorithm 4 is applied in our simulator.

Conclusion

We implemented the DNA self-assembly simulation system and related utilities. The simulator's archetecture was designed intended to support various theroy modeles, and both for graphic interface and command user interface. It's very easy to extend other components to the software. The software meets the design requirements, and has been applied in our research work.

Algorithm 4: Supertile Spliting

Input: S, the supertile
τ, the temperature of the system
Output: AS, the array of splited sub-supertiles
nextsupertile $\leftarrow S$;
repeat
    Map the nextsupertile to graph ; `/* The tiles are regarded as the node of the graph. The same glue type between two tiles is regarded as the edge between two nodes, and the glue strenth is the value of edge */`
    Apply min-cut on the graph ;
    if *value of min-cut* $< \tau$ **then**
        if *the graph is maped from the item of array AS* **then**
            Update the item with one sub-supertiles and put another sub-supertiles to the end of array AS ;
        else
            Put the two sub-supertiles to AS ;
            nextsupertile $\leftarrow$ first item of array AS ;
until *Reach to the end of array AS* ;
return AS;

Further Development

Inspired by the research we have done, we'll try to improve the software at following aspects:

1. Improve the speed of 2hTAM simulation.

2. Add more options to the GUI of simulator.

3. Develop more modules for the simulator to support multifarious theoretical models.

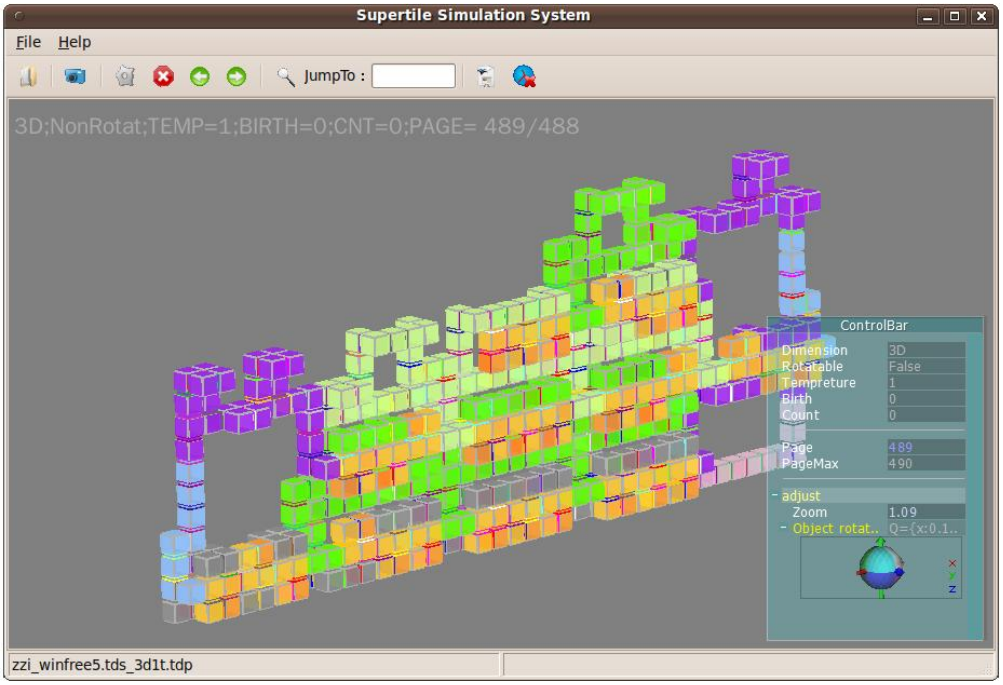


Figure 14. Scrrenshot of Simulator

References

[1] Nadrian C. Seeman. Nucleic-acid junctions and lattices. *Journal of Theoretical Biology*, 99:237–247, 1982. 1

[2] Hao Wang. Dominoes and the AEA case of the decision problem. In *Proceedings of the Symposium on Mathematical Theory of Automata (New York, 1962)*, pages 23–55. Polytechnic Press of Polytechnic Inst. of Brooklyn, Brooklyn, N.Y., 1963. 1

[3] Erik Winfree, Furong Liu, Lisa A. Wenzler, and Nadrian C. Seeman. Design and self-assembly of two-dimensional DNA crystals. *Nature*, 394(6693):539–44, 1998. 1

[4] Erik Winfree. Algorithmic self-assembly of dna, 1998. 2

[5] Paul Rothemund and Erik Winfree. The program-size complexity of self-assembled squares. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 459–468, 2000. 2

[6] Erik Winfree. Simulations of computing by self-assembly. 1998. 2, 3

[7] Erik Winfree and Renat Bekbolatov. Proofreading tile sets: Error correction for algorithmic self-assembly. In *DNA Computers 9 LNCS*, pages 126–144, 2004. 2, 3, 4

[8] Paul W.K. Rothemund, Nick Papadakis, and Erik Winfree. Algorithmic self-assembly of DNA Sierpinski triangles. *PLoS Biology*, 2(12):2041–2053, 2004. 3

[9] Erik Winfree. Self-healing tile sets. Technical report, Foundations of Nanoscience: Self-Assembled Architectures and Devices, 2005, 2005. 3

[10] Ho-Lin Chen Ashish, Ho lin Chen, and Ashish Goel. Error free self-assembly using error prone tiles. In *In DNA Based Computers 10*, pages 274–283, 2004. 3, 6, 19

[11] Rebecca Schulman and Erik Winfree. Programmable control of nucleation for algorithmic self-assembly. In *In DNA Based Computers 10, LNCS*, pages 319–328, 2005. 3

[12] D. Doty, M. Patiz, and S. Summers. Limitations of self-assembly at temperature 1. In *Proceedings of the 15th DIMACS Workshop on DNA Based Computers*, 2008. 4

[13] Ján Maňuch, Ladislav Stacho, and Christine Stoll. Two lower bounds for self-assemblies at temperature 1. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 808–809, New York, NY, USA, 2009. ACM. 4

[14] Harish Chandran, Nikhil Gopalkrishnan, and John Reif. The tile complexity of linear assemblies. In *ICALP '09: Proceedings of the 36th International Colloquium on Automata, Languages and Programming*, pages 235–253, Berlin, Heidelberg, 2009. Springer-Verlag. 4

[15] Ming-Yang Kao and Robert Schweller. Randomized self-assembly for approximate shapes. In *ICALP '08: Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part I*, pages 370–384, Berlin, Heidelberg, 2008. Springer-Verlag. 4

[16] David Doty. Randomized self-assembly for exact shapes. In *FOCS '09: Proceedings of the 2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 85–94, Washington, DC, USA, 2009. IEEE Computer Society. 4

[17] E. Demaine, M. Demaine, S. Fekete, M. Ishaque, E. Rafalin, R. Schweller, and D. Souvaine. Staged self-assembly: Nanomanufacture of arbitrary shapes with O(1) glues. In *Proceedings of the 13th International Meeting on DNA Computing*, 2007. 4

[18] Zachary Abel, Nadia Benbernou, Mirela Damian, Erik D. Demaine, Martin L. Demaine, Robin Flatland, Scott Kominers, and Robert Schweller. Shape replication through self-assembly and RNase enzymes. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2010)*, pages 1045–1064, Austin, Texas, January 17–19 2010. 4

[19] DNA and Natural Algorithms Group. Xgrow simulator. Homepage: http://www.dna.caltech.edu/Xgrow, 2009. 5, 18

[20] Matthew J. Patitz. ISU TAS. Homepage: http://www.cs.iastate.edu/~lnsa/software.html, 2008. 5

[21] Matthew Cook, Yunhui Fu, and Robert T. Schweller. Temperature 1 self-assembly: Deterministic assembly in 3d and probabilistic assembly in 2d. *CoRR*, abs/0912.0027, 2009. 7, 16, 20

[22] Michael Sipser. *Introduction to the Theory of Computation, Second Edition.* Course Technology, 2 edition, February 2005. 9

Appendix A

Appendix A

Convert Arbitrary Zig-Zag Tile Set from $\tau = 2$ to $\tau = 1$

Notation

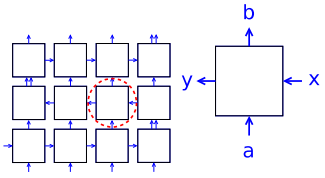
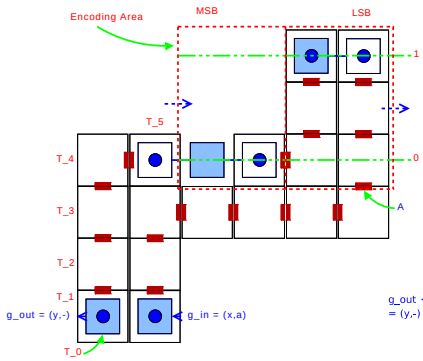
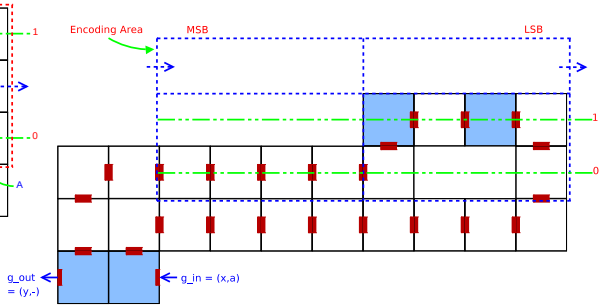
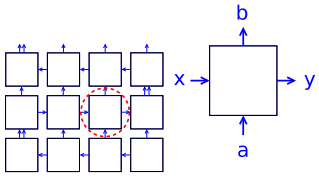
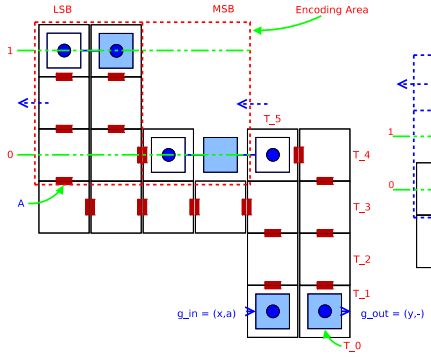
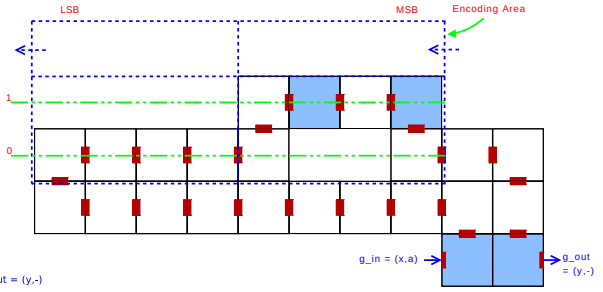
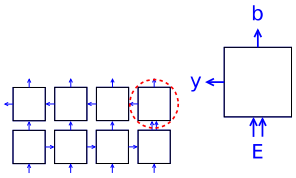
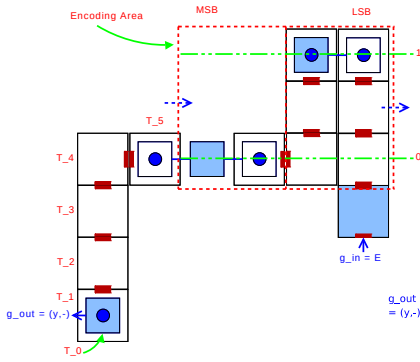
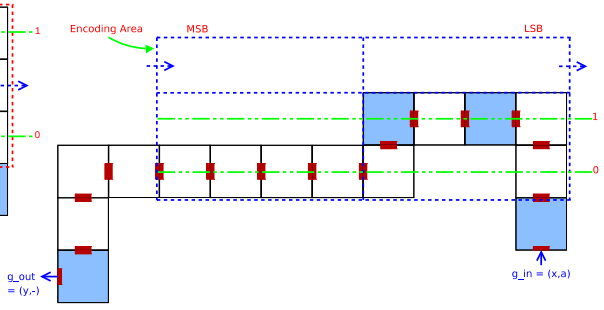
A tile t is a four sided Wang tile denoted by the quadruple (g_n, g_e, g_s, g_w), g_n, g_e, g_s, g_w denote the glue type of the four sides: North, East, South, and West.

The Converting Table

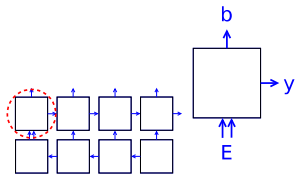
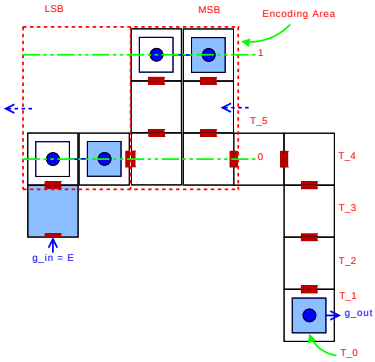
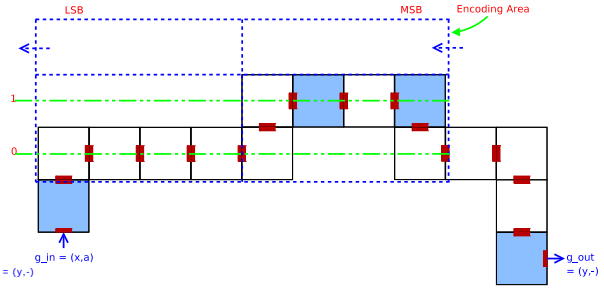
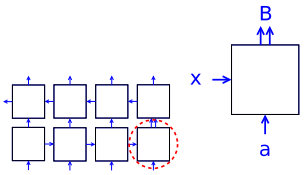
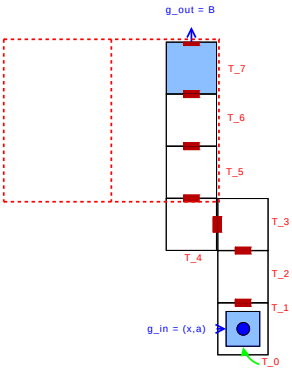
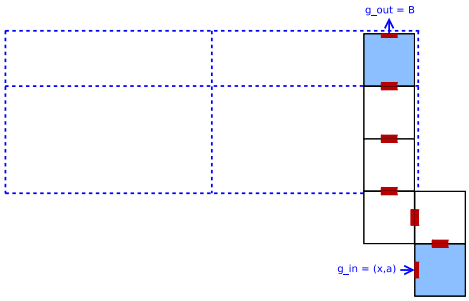
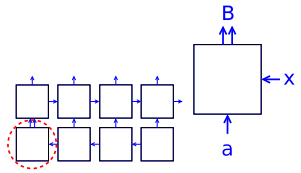
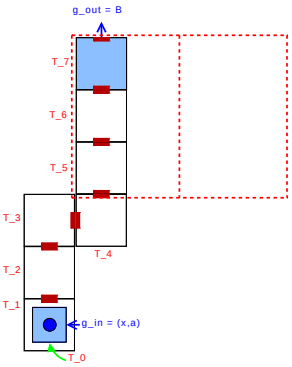
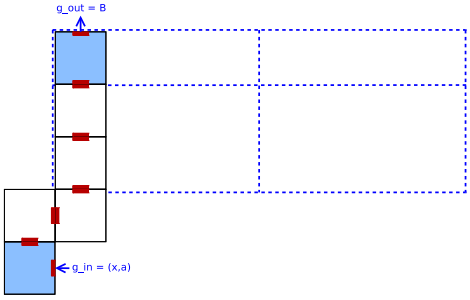
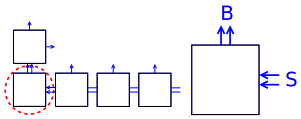
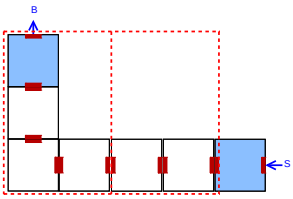
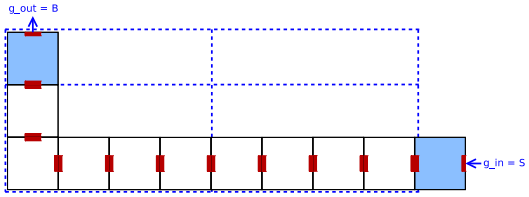
When the Zig-Zag tiles in 2D are categorized into some types, the tile types can be mapped to 3D and 2D probabilistic tile sets directly. Table 4 list all of the relationships between the 2D tiles and 3D tile sets.

The tiles (see the 3D figures in Table 4) are ***different*** from each other. The lines between two adjacent tiles denote the glues which strength are 1 and they are also ***different*** from each other except adjacent glues. Large squares denote the tiles in the plane $z = 0$, and small squares denote the tiles in the plane $z = 1$. The encoded code(e_n) showed in the figures in Table 4 is two bits long(maxbits=2). The parameter K of Probabilistic Zig-Zag is 2 in Table 4.

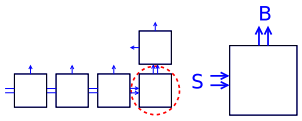
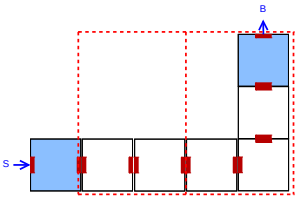
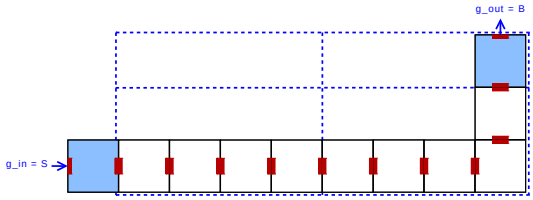
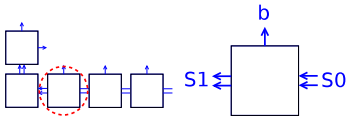
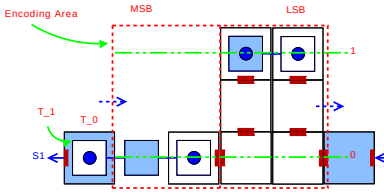
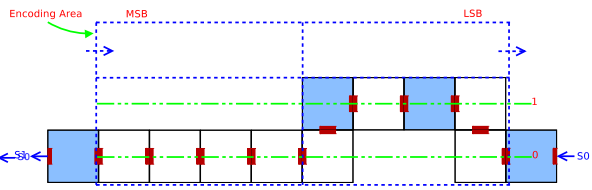
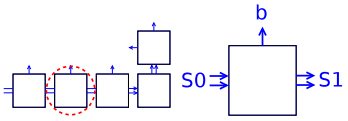
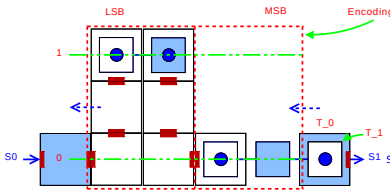
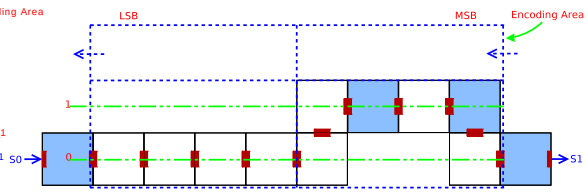
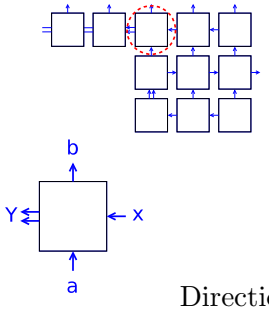
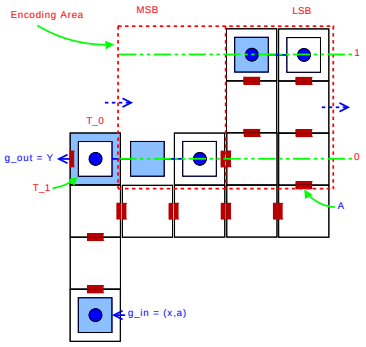
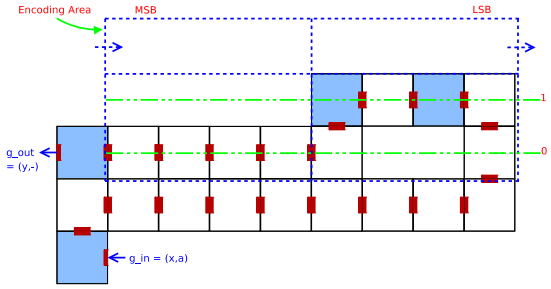
Table 4: Zig-Zag Tile Set Mapping

Zig-Zag in 2D $\tau = 2$	**Zig-Zag in 3D $\tau = 1$**	**Prob. Zig-Zag in 2D $\tau = 1$**
 Direction West, West 1 (DWW1)		
 Direction East, East 1 (DEE1)		
 Turn at East, West 1 (TEW1)		

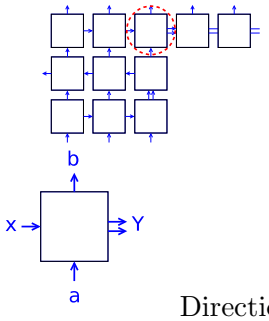
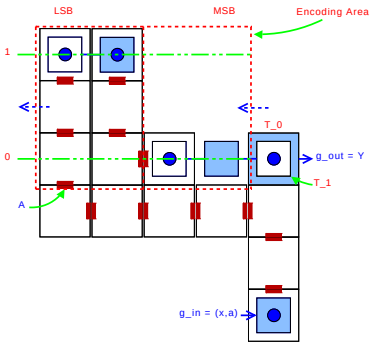
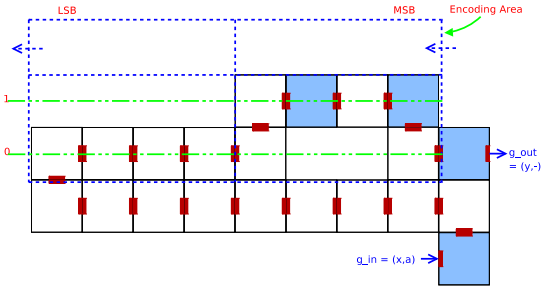
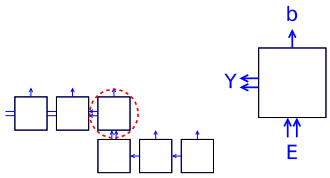
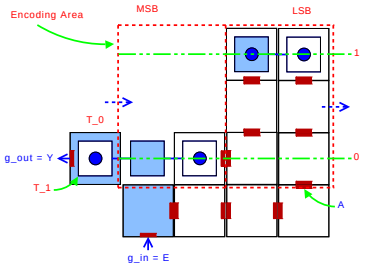
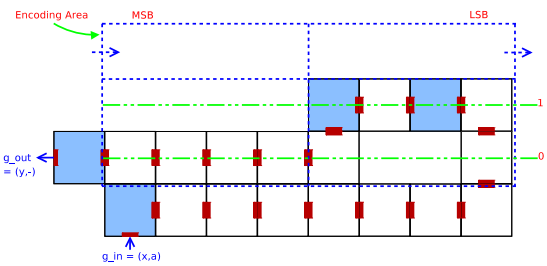
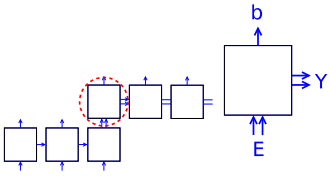
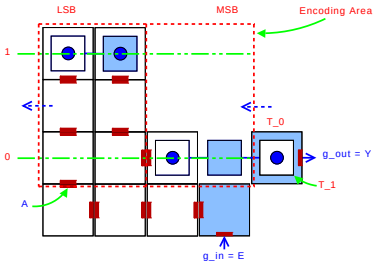
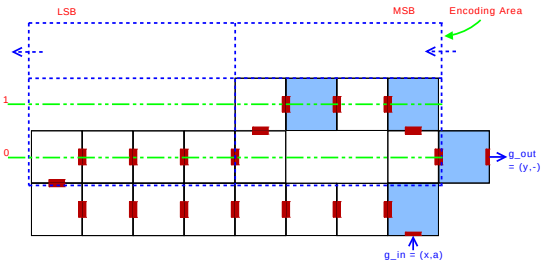
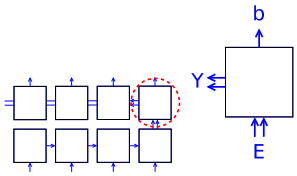
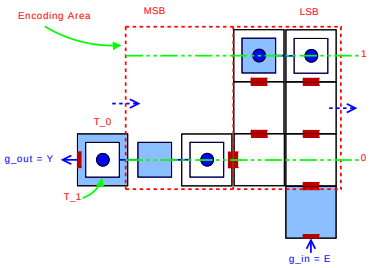
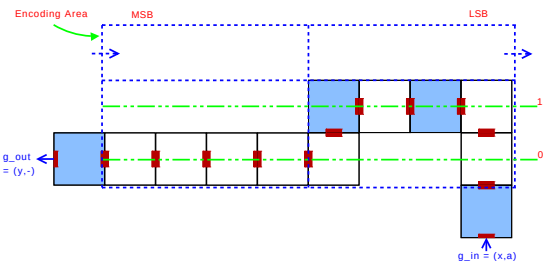
Continued on next page . . .

Zig-Zag in 2D $\tau = 2$	**Zig-Zag in 3D $\tau = 1$**	**Prob. Zig-Zag in 2D $\tau = 1$**
 Turn at West, East 1 (TWE1)		
 Direction East, North 2 (DEN2)		
 Direction West, North 2 (DWN2)		
 Side West, North 2 (SWN2)		

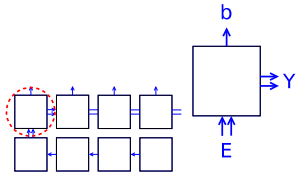
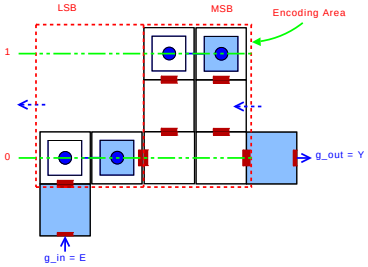
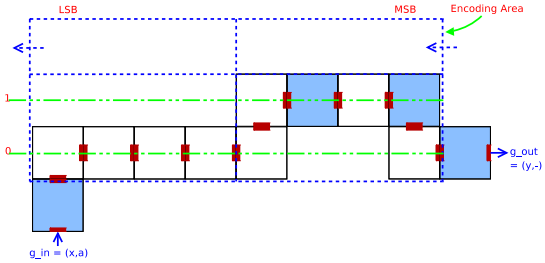
Continued on next page . . .

Zig-Zag in 2D $\tau = 2$	**Zig-Zag in 3D $\tau = 1$**	**Prob. Zig-Zag in 2D $\tau = 1$**
 Side East, North 2 (SEN2)		
 Fixed Location, Direction West (FW)		
 Fixed Location, Direction East (FE)		
 Direction West, West 2 (DWW2)		

Continued on next page . . .

Zig-Zag in 2D $\tau = 2$	Zig-Zag in 3D $\tau = 1$	Prob. Zig-Zag in 2D $\tau = 1$
 Direction East, East 2 (DEE2)		
 Turn at West, West 2 (TWW2)		
 Turn at East, East 2 (TEE2)		
 Turn at East, West 2 (TEW2)		

Continued on next page . . .

Zig-Zag in 2D $\tau = 2$	Zig-Zag in 3D $\tau = 1$	Prob. Zig-Zag in 2D $\tau = 1$
 Turn at West, East 2 (TWE2)		

Algorithms for Converting Tile Type

To converting arbitrary zig-zag tile set, the first step is to categorize the tiles into the sixteen types listed in Table 4. Then the zig-zag tiles in 2D can be converted directly to zig-zag tile set in 3D or probabilistic zig-zag tile set in 2D. We can recgonize all of the tile types in 2D by observing the positions the tiles be placed during the growth. In an other words, a simulator have to be used to get the types of tiles in 2D.

The tiles are categorized into sixteen types, it seems it will cost much effort to implement the converter by software. As we apply some tricks on the algorithms, the creating of tile sets converter would be simple. The tile set type DWW1, DWW2, TEW1, TEW2, TWW2 have similar structure, the differences between those types are the positions of the input and output glues. While the tile set type DEE1, DEE2, TWE1, TWE2, TEE2 also have the similar structure, they are the mirror of the tile set type DWW1, DWW2, TEW1, TEW2, TWW2. And SWN2 is the mirror of SEN2, DWN2 is mirror of DEN2.

Categorize Zig-Zag Tiles

A simulator is used to detect the tile types in this algorithm. The only restriction is that all of the tile types should be occured at least once in resonable steps during simulating. The algorithm will return right after all of the tile types are detected. The algorithm is listed

in Algorithm 5.

Zig-Zag in 3D

The algorithms to create the tile sets in 3D are listed in Algorithm 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17. The number of the binary bits used in the codes is denoted by maxbits.

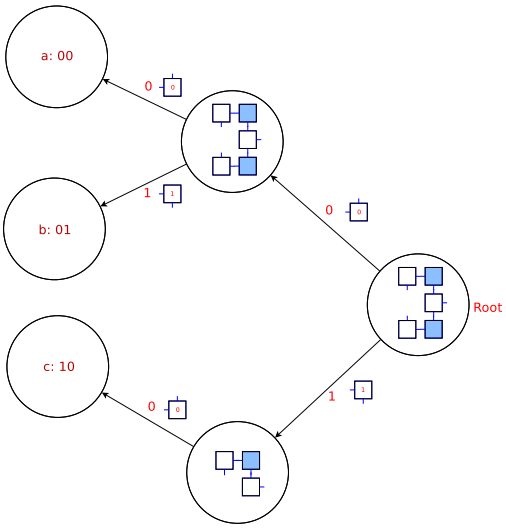


Figure 15. The logic binary tree for constructing decoding tile set (direction left).

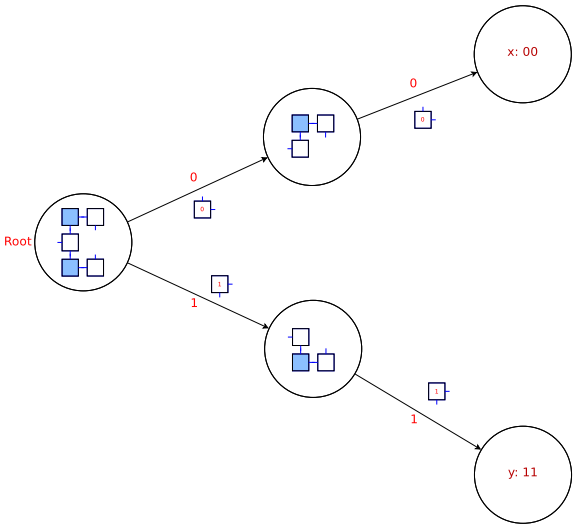


Figure 16. The logic binary tree for constructing decoding tile set (direction right).

Algorithm 5: CATEGORIZE-ZIG-ZAG-TILES()

Input: $T'_{2d,2t}$, Un-categorized tile set at temperature 2 in 2D
Output: $T_{2d,2t}$, Categorized tile set at temperature 2 in 2D
$T_{2d,2t} \leftarrow \phi$; numTypes $\leftarrow 0$
while *numTypes* $< |T'_{2d,2t}|$ **do**
  select tile type $t' \in T'_{2d,2t}$ which can attach at position *pos*
  record this step (t', *pos*) and append it to array *steps*
  if *the type of* t' *is detected* **then Continue**
  adjIdx $\leftarrow 0$; numTypes $\leftarrow$ numTypes + 1
  for $i \leftarrow (|steps| - 1)$ **to** 1 **do**
    if *pos is adjacent to steps*$[i]$*.pos and the glues at the sides are the same* **then**
      adjIdx $\leftarrow$ i; adjTile $\leftarrow$ *steps*$[i].t$
      dir $\leftarrow$ the side of t' that is adjacent to *steps*$[i].t$
      Goto end of this for loop
  if *adjIdx is not between* $(|steps| - 1)$ *and* 1 **then Continue**
  switch *the value of dir* **do**
    case *EAST*
      if $t'.g_e.strength > 1$ **then**
        if $t'.g_n.strength > 1$ **then** t'.type $\leftarrow$ SWN2; $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{t'\}$
        else if $t'.g_w.strength > 1$ **then** t'.type $\leftarrow$ FW; $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{t'\}$
      else
        if $t'.g_n.strength > 1$ **then** t'.type $\leftarrow$ DWN2; $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{t'\}$
        else if $t'.g_w.strength > 1$ **then** t'.type $\leftarrow$ DWW2; $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{t'\}$
        else t'.type $\leftarrow$ DWW1; $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{t'\}$
    case *SOUTH*
      switch *the type of adjTile* **do**
        case *DEN2 or SEN2*
          if $t'.g_e.strength > 1$ **then** t'.type $\leftarrow$ TEE2; $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{t'\}$
          else if $t'.g_w.strength > 1$ **then** t'.type $\leftarrow$ TEW2; $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{t'\}$
          else t'.type $\leftarrow$ TEW1; $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{t'\}$
        case *DWN2 or SWN2*
          if $t'.g_e.strength > 1$ **then** t'.type $\leftarrow$ TWE2; $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{t'\}$
          else if $t'.g_w.strength > 1$ **then** t'.type $\leftarrow$ TWW2; $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{t'\}$
          else t'.type $\leftarrow$ TWE1; $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{t'\}$
        otherwise Error, Ignored
    case *WEST*
      if $t'.g_w.strength > 1$ **then**
        if $t'.g_n.strength > 1$ **then** t'.type $\leftarrow$ SEN2; $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{t'\}$
        else t'.type $\leftarrow$ FE; $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{t'\}$
      else
        if $t'.g_n.strength > 1$ **then** t'.type $\leftarrow$ DEN2; $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{t'\}$
        else if $t'.g_e.strength > 1$ **then** t'.type $\leftarrow$ DEE2; $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{t'\}$
        else t'.type $\leftarrow$ DEE1; $T_{2d,2t} \leftarrow T_{2d,2t} \cup \{t'\}$
    otherwise Error, Ignored
return $T_{2d,2t}$

Algorithm 6: TILESET-CONVERT-ALL-CATEGORIES()

Input: $T_{2d,2t}$, Categorized tile set at temperature 2 in 2D
Output: $T_{3d,1t}$, Tile set at temperature 1 in 3D
/*

$$T_{2d,2t} = T_{DWW1} \cup T_{DWW2} \cup T_{TEW1} \cup T_{TEW2} \cup T_{TWW2}$$
$$\cup T_{DEE1} \cup T_{DEE2} \cup T_{TWE1} \cup T_{TWE2} \cup T_{TEE2}$$
$$\cup T_{FE} \cup T_{FW} \cup T_{SWN2} \cup T_{SEN2} \cup T_{DWN2} \cup T_{DEN2}$$

*/

$T_{3d,1t} \leftarrow \phi$; /* The tile set in 3D temperature 1 */
$G_{ns} \leftarrow \phi$; /* The glues at the north and south sides of the tile */
$S_{2d,2t} \leftarrow \{s_i | s_i =$ Strength of all of the glues of t_j, $t_j \in T_{2d,2t}\}$;
foreach $t_i \in T_{2d,2t}$ **do**
  if $1 = s_{t_i.g_n}$ **then**
    $G_{ns} \leftarrow G_{ns} \cup \{t_i.g_n\}$;
  if $1 = s_{t_i.g_s}$ **then**
    $G_{ns} \leftarrow G_{ns} \cup \{t_i.g_s\}$;
Encode the glues in set G_{ns} by binary codes $e_i | i \in G_{ns}$;
$\mathcal{E} \leftarrow \{e_i | i \in G_{ns}\}$; /* $\mathcal{E}$ contains all of the code of glue $\in G_{ns}$. */
maxbits $\leftarrow \lceil \log |G_{ns}| \rceil$;
$T_{3d,1t} \leftarrow T_{3d,1t} \cup$ GENERATE-DECODE-TILE-SET($T_{dirleft} \cup T_{dirright}, S_{2d,2t}, \mathcal{E}$, maxbits) ;
$T_{3d,1t} \leftarrow T_{3d,1t} \cup$ GENERATE-ENCODE-TILE-SET($T_{dirleft} \cup T_{dirright}, S_{2d,2t}, \mathcal{E}$, maxbits) ;
return $T_{3d,1t}$;

Algorithm 7: GENERATE-DECODE-TILE-SET()

Input: $T_{2d,2t}$, Categorized tile set at temperature 2 in 2D
$S_{2d,2t}$, all of the strength of glue in $T_{2d,2t}$
$\mathcal{E}$, all of the code of glue $\in G_{ns}$
maxbits, the number of binary bits to encode all of the glues $\in G_{ns}$
Output: $T'_{3d,1t}$, Tile set at temperature 1 in 3D
$T'_{3d,1t} \leftarrow \phi$;
$G_{2w} \leftarrow \phi$; `/* The glue set of input from east to west */`
$G_{2e} \leftarrow \phi$; `/* The glue set of input from west to east */`
foreach $t_i \in T_{DWW1} \cup T_{DWW2} \cup T_{DWN2}$ **do**
  $G_{2w} \leftarrow G_{2w} \cup \{t_i.g_e\}$;
foreach $t_i \in T_{DEE1} \cup T_{DEE2} \cup T_{DEN2}$ **do**
  $G_{2e} \leftarrow G_{2e} \cup \{t_i.g_w\}$;
foreach $g_i \in G_{2w}$ **do**
  $G_{w,i} \leftarrow \phi$;
  foreach $t_j \in T_{DWW1} \cup T_{DWW2} \cup T_{DWN2}$ **do**
    if $t_j.g_e = g_i$ *and* $s_{t_j.g_s} < 2$ **then**
      `/*` $s_{t_j.g_s}$ `is the strength of` g_s `of tile` t_j, $s_{t_j.g_s} \in S_{2d,2t}$ `*/`
      $G_{w,i} \leftarrow G_{w,i} \cup \{t_j.g_s\}$
  $T'_{3d,1t} = T'_{3d,1t} \cup$ GENERATE-DECODE-TILE-TO-WEST$(g_i, G_{w,i}, \mathcal{E}, maxbits)$;
foreach $g_i \in G_{2e}$ **do**
  $G_{e,i} = \phi$;
  foreach $t_j \in T_{DEE1} \cup T_{DEE2} \cup T_{DEN2}$ **do**
    if $t_j.g_w = g_i$ *and* $s_{t_j.g_s} < 2$ **then**
      `/*` $s_{t_j.g_s}$ `is the strength of` g_s `of tile` t_j, $s_{t_j.g_s} \in S_{2d,2t}$ `*/`
      $G_{e,i} \leftarrow G_{e,i} \cup \{t_j.g_s\}$
  $T'_{3d,1t} \leftarrow T'_{3d,1t} \cup$ GENERATE-DECODE-TILE-TO-EAST$(g_i, G_{e,i}, \mathcal{E}, maxbits)$;
return $T'_{3d,1t}$;

Algorithm 8: GENERATE-DECODE-TILE-TO-WEST()

Input: g_{in}, the input glue of the current tile set
$G_{w,i}$, the glue set to be the output glues of current tile set
$\mathcal{E}$, all of the code of glue $\in G_{ns}$
maxbits, the number of binary bits to encode all of the glues $\in G_{ns}$
Output: $T'_{3d,1t}$, Tile set at temperature 1 in 3D
$T'_{3d,1t} \leftarrow \phi$;
`/* Contruct a binary tree according to the encoding code of each items in` $G_{w,i}$. `*/`
foreach $g_i \in G_{w,i}$ **do**
    curnode ← root;
    foreach *bit of* e_{g_i} *from MSB to LSB* **do**
        if *bit = 1* **then**
            if *curnode have no right child* **then**
                create right child of the curnode;
            curnode ← curnode.right_child;
        else
            if *curnode have no left child* **then**
                create left child of the curnode;
            curnode ← curnode.left_child;
Traversal the tree by pre-order algorithm: Part of the tiles set will be generated and saved to $T'_{3d,1t}$ in each visitation. The input glue of the tile set is g_{in}. See Figure 15;
return $T'_{3d,1t}$;

Algorithm 9: GENERATE-DECODE-TILE-TO-EAST()

Input: g_{in}, the input glue of the current tile set
$G_{e,i}$, the glue set to be the output glues of current tile set
$\mathcal{E}$, all of the code of glue $\in G_{ns}$
maxbits, the number of binary bits to encode all of the glues $\in G_{ns}$
Output: $T'_{3d,1t}$, Tile set at temperature 1 in 3D
$T'_{3d,1t} \leftarrow \phi$;
`/* Contruct a binary tree according to the encoding of each items in` $G_{e,i}$`. */`
foreach $g_i \in G_{e,i}$ **do**
    curnode ← root;
    foreach *bit of* e_{g_i} *from MSB to LSB* **do**
        if *bit = 1* **then**
            if *curnode have no right child* **then**
                create right child of the curnode;
            curnode ← curnode.right_child;
        else
            if *curnode have no left child* **then**
                create left child of the curnode;
            curnode ← curnode.left_child;
Traversal the tree by pre-order algorithm: Part of the tiles set will be generated and saved to $T'_{3d,1t}$ in each visitation. The input glue of the tile set is g_{in}. See Figure 16;
return $T'_{3d,1t}$;

Algorithm 10: GENERATE-ENCODE-TILE-SET()

Input: $T_{2d,2t}$, Categorized tile set at temperature 2 in 2D
$\mathcal{E}$, all of the code of glue $\in G_{ns}$
maxbits, the number of binary bits to encode all of the glues $\in G_{ns}$
Output: $T'_{3d,1t}$, Tile set at temperature 1 in 3D
$T'_{3d,1t} = \phi$;
$T'_{3d,1t} \leftarrow T'_{3d,1t} \cup$ GENERATE-ENCODE-TILE-SET-TO-EAST$(T_{2d,2t}, \mathcal{E}, maxbits)$;
$T'_{3d,1t} \leftarrow T'_{3d,1t} \cup$ GENERATE-ENCODE-TILE-SET-TO-WEST$(T_{2d,2t}, \mathcal{E}, maxbits)$;
$T'_{3d,1t} \leftarrow T'_{3d,1t} \cup$ GENERATE-ENCODE-TILE-SET-OTHERS$(T_{2d,2t}, \mathcal{E}, maxbits)$;
return $T'_{3d,1t}$;

Algorithm 11: GENERATE-ENCODE-TILE-SET-TO-WEST()

Input: $T_{2d,2t}$, Categorized tile set at temperature 2 in 2D
$\mathcal{E}$, all of the code of glue $\in G_{ns}$
maxbits, the number of binary bits to encode all of the glues $\in G_{ns}$
Output: $T'_{3d,1t}$, Tile set at temperature 1 in 3D
$T'_{3d,1t} = \phi$;
foreach $t_i \in T_{DWW1}$ **do**
    $g_{in} \leftarrow (t_i.g_e, t_i.g_s)$;
    $g_{out} \leftarrow (t_i.g_w, -)$;
    $T'_{3d,1t} \leftarrow$
    $T'_{3d,1t} \cup$ GENERATE-ENCODE-TILE-TO-WEST$(DWW1, g_{in}, g_{out}, e_{t_i.g_n}, maxbits)$;
    /* $e_{t_i.g_n}$ is the encoding code of glue g_n of tile t_i */
foreach $t_i \in T_{TEW1}$ **do**
    $g_{in} \leftarrow t_i.g_s$;
    $g_{out} \leftarrow (t_i.g_w, -)$;
    $T'_{3d,1t} \leftarrow$
    $T'_{3d,1t} \cup$ GENERATE-ENCODE-TILE-TO-WEST$(TEW1, g_{in}, g_{out}, e_{t_i.g_n}, maxbits)$;
foreach $t_i \in T_{DWW2}$ **do**
    $g_{in} \leftarrow (t_i.g_e, t_i.g_s)$;
    $g_{out} \leftarrow t_i.g_w$;
    $T'_{3d,1t} \leftarrow$
    $T'_{3d,1t} \cup$ GENERATE-ENCODE-TILE-TO-WEST$(DWW2, g_{in}, g_{out}, e_{t_i.g_n}, maxbits)$;
foreach $t_i \in T_{TWW2}$ **do**
    $g_{in} \leftarrow t_i.g_s$;
    $g_{out} \leftarrow t_i.g_w$;
    $T'_{3d,1t} \leftarrow$
    $T'_{3d,1t} \cup$ GENERATE-ENCODE-TILE-TO-WEST$(TWW2, g_{in}, g_{out}, e_{t_i.g_n}, maxbits)$;
foreach $t_i \in T_{TEW2}$ **do**
    $g_{in} \leftarrow t_i.g_s$;
    $g_{out} \leftarrow t_i.g_w$;
    $T'_{3d,1t} \leftarrow$
    $T'_{3d,1t} \cup$ GENERATE-ENCODE-TILE-TO-WEST$(TEW2, g_{in}, g_{out}, e_{t_i.g_n}, maxbits)$;
return $T'_{3d,1t}$;

Algorithm 12: GENERATE-ENCODE-TILE-SET-TO-EAST()

Input: $T_{2d,2t}$, Categorized tile set at temperature 2 in 2D
$\mathcal{E}$, all of the code of glue $\in G_{ns}$
maxbits, the number of binary bits to encode all of the glues $\in G_{ns}$
Output: $T'_{3d,1t}$, Tile set at temperature 1 in 3D
$T'_{3d,1t} = \phi$;
foreach $t_i \in T_{DEE1}$ **do**
    $g_{in} \leftarrow (t_i.g_w, t_i.g_s)$;
    $g_{out} \leftarrow (t_i.g_e, -)$;
    $T'_{3d,1t} \leftarrow T'_{3d,1t} \cup$ GENERATE-ENCODE-TILE-TO-EAST$(DEE1, g_{in}, g_{out}, e_{t_i.g_n}, maxbits)$
    ; `/* ` $e_{t_i.g_n}$ `is the encoding code of glue` g_n `of tile` t_i `*/`
foreach $t_i \in T_{TWE1}$ **do**
    $g_{in} \leftarrow t_i.g_s$;
    $g_{out} \leftarrow (t_i.g_e, -)$;
    $T'_{3d,1t} \leftarrow T'_{3d,1t} \cup$ GENERATE-ENCODE-TILE-TO-EAST$(TWE1, g_{in}, g_{out}, e_{t_i.g_n}, maxbits)$;
foreach $t_i \in T_{DEE2}$ **do**
    $g_{in} \leftarrow (t_i.g_w, t_i.g_s)$;
    $g_{out} \leftarrow t_i.g_e$;
    $T'_{3d,1t} \leftarrow T'_{3d,1t} \cup$ GENERATE-ENCODE-TILE-TO-EAST$(DEE2, g_{in}, g_{out}, e_{t_i.g_n}, maxbits)$;
foreach $t_i \in T_{TEE2}$ **do**
    $g_{in} \leftarrow t_i.g_s$;
    $g_{out} \leftarrow t_i.g_e$;
    $T'_{3d,1t} \leftarrow T'_{3d,1t} \cup$ GENERATE-ENCODE-TILE-TO-EAST$(TEE2, g_{in}, g_{out}, e_{t_i.g_n}, maxbits)$;
foreach $t_i \in T_{TWE2}$ **do**
    $g_{in} \leftarrow t_i.g_s$;
    $g_{out} \leftarrow t_i.g_e$;
    $T'_{3d,1t} \leftarrow T'_{3d,1t} \cup$ GENERATE-ENCODE-TILE-TO-EAST$(TWE2, g_{in}, g_{out}, e_{t_i.g_n}, maxbits)$;
return $T'_{3d,1t}$;

Algorithm 13: GENERATE-ENCODE-TILE-SET-OTHERS()

Input: $T_{2d,2t}$, Categorized tile set at temperature 2 in 2D
$\mathcal{E}$, all of the code of glue $\in G_{ns}$
maxbits, the number of binary bits to encode all of the glues $\in G_{ns}$
Output: $T'_{3d,1t}$, Tile set at temperature 1 in 3D
$T'_{3d,1t} = \phi$;
foreach $t_i \in T_{DWN2}$ **do**
  $g_{in} \leftarrow (t_i.g_e, t_i.g_s)$;
  $g_{out} \leftarrow t_i.g_n$;
  $T'_{3d,1t} \leftarrow T'_{3d,1t} \cup$ GENERATE-CONNECTION-TILE$(DWN2, g_{in}, g_{out}, maxbits)$;
foreach $t_i \in T_{DEN2}$ **do**
  $g_{in} \leftarrow (t_i.g_w, t_i.g_s)$;
  $g_{out} \leftarrow t_i.g_n$;
  $T'_{3d,1t} \leftarrow T'_{3d,1t} \cup$ GENERATE-CONNECTION-TILE$(DEN2, g_{in}, g_{out}, maxbits)$;
foreach $t_i \in T_{SWN2}$ **do**
  $g_{in} \leftarrow t_i.g_e$;
  $g_{out} \leftarrow t_i.g_n$;
  $T'_{3d,1t} \leftarrow T'_{3d,1t} \cup$ GENERATE-CONNECTION-TILE$(SWN2, g_{in}, g_{out}, maxbits)$;
foreach $t_i \in T_{SEN2}$ **do**
  $g_{in} \leftarrow t_i.g_w$;
  $g_{out} \leftarrow t_i.g_n$;
  $T'_{3d,1t} \leftarrow T'_{3d,1t} \cup$ GENERATE-CONNECTION-TILE$(SEN2, g_{in}, g_{out}, maxbits)$;
foreach $t_i \in T_{FW}$ **do**
  $g_{in} \leftarrow t_i.g_e$;
  $g_{out} \leftarrow t_i.g_w$;
  $T'_{3d,1t} \leftarrow T'_{3d,1t} \cup$ GENERATE-ENCODE-TILE-FIXED$(FW, g_{in}, g_{out}, e_{t_i.g_n}, maxbits)$;
  /* $e_{t_i.g_n}$ is the encoding code of glue g_n of tile t_i */
foreach $t_i \in T_{FE}$ **do**
  $g_{in} \leftarrow t_i.g_w$;
  $g_{out} \leftarrow t_i.g_e$;
  $T'_{3d,1t} \leftarrow T'_{3d,1t} \cup$ GENERATE-ENCODE-TILE-FIXED$(FE, g_{in}, g_{out}, e_{t_i.g_n}, maxbits)$;
return $T'_{3d,1t}$;

Algorithm 14: GENERATE-ENCODE-TILE-TO-WEST()

Input: tiletype, the type of the tile
g_{in}, the input glue
g_{out}, the output glue
e_n, the code of the north glue
maxbits, the number of binary bits to encode all of the glues $\in G_{ns}$
Output: $T'_{3d,1t}$, Tile set at temperature 1 in 3D
$T'_{3d,1t} \leftarrow \phi$;
/* Generate ***distinct*** tiles as showed in Table 4. */
if *tiletype = DWW1* **then**
    Generate the tiles $t'_0, t'_1, t'_2, t'_3, t'_4, t'_5$ and put it to $T'_{3d,1t}$;
    /* The $t'_0, t'_1, t'_2, t'_3, t'_4, t'_5$ are denoted separately by t_0, t_1, t_2, t_3, t_4, and t_5 in the figure of DWW1 in the Table 4. The glue is g_{out} at the west of the tile t'_0. */
else if *tiletype = DWW2* **then**
    Generate the tiles t'_0, t'_1 and put it to $T'_{3d,1t}$;
else if *tiletype = TEW1* **then**
    Generate the tiles $t'_0, t'_1, t'_2, t'_3, t'_4, t'_5$ and put it to $T'_{3d,1t}$;
else if *tiletype = TEW2 or tiletype = TWW2* **then**
    Generate the tiles t'_0, t'_1 and put it to $T'_{3d,1t}$;
Generate the tiles in the *Encoding Area* and put it to $T'_{3d,1t}$;
/* The positions of the tiles are depend on the e_n; Each bit of the e_n are encoded by two tiles in the plane $z = 1$; The tiles will place at the position '1'(the dotted line denoted by '1' in the figures if the bit is 1, while the tiles will place at position '0' if the bit is 0; The encoding of the most significant bit (MSB) of the e_n is place at the left side of the ***Encoding Area***, and the least siginificant bit(LSB) of the e_n is place at the right side of the ***Encoding Area***. All of the encoded tiles are connected by the tiles in the plane $z = 0$. */
if *tiletype = DWW1 or tiletype = DWW2 or tiletype = TWW2* **then**
    Generate the tiles between glue A and glue g_{in}, put those tiles to $T'_{3d,1t}$;
else if *tiletype = TEW1 or tiletype = TEW2* **then**
    Generate the tile with glue g_{in}, put it to $T'_{3d,1t}$;
return $T'_{3d,1t}$;

Algorithm 15: GENERATE-ENCODE-TILE-TO-EAST()

Input: tiletype, the type of the tile
g_{in}, the input glue
g_{out}, the output glue
e_n, the code of the north glue
maxbits, the number of binary bits to encode all of the glues $\in G_{ns}$
Output: $T'_{3d,1t}$, Tile set at temperature 1 in 3D
$T'_{3d,1t} \leftarrow \phi$;
if *tiletype = DEE1* **then**
    Generate the tiles $t'_0, t'_1, t'_2, t'_3, t'_4, t'_5$ and put it to $T'_{3d,1t}$;
    /* The $t'_0, t'_1, t'_2, t'_3, t'_4, t'_5$ are denoted separately by t_0, t_1, t_2, t_3, t_4, and t_5 in the figures of Table 4. The glue is g_{out} at the east of the tile t'_0. */
else if *tiletype = DEE2* **then**
    Generate the tiles t'_0, t'_1 and put it to $T'_{3d,1t}$;
else if *tiletype = TWE1* **then**
    Generate the tiles $t'_0, t'_1, t'_2, t'_3, t'_4, t'_5$ and put it to $T'_{3d,1t}$;
else if *tiletype = TWE2 or tiletype = TEE2* **then**
    Generate the tiles t'_0, t'_1 and put it to $T'_{3d,1t}$;
Generate the tiles in the *Encoding Area* and put it to $T'_{3d,1t}$;
if *tiletype = DEE1 or tiletype = DEE2 or tiletype = TEE2* **then**
    Generate the tiles between glue A and glue g_{in}, put those tiles to $T'_{3d,1t}$;
else if *tiletype = TWE1 or TWE2* **then**
    Generate the tile with glue g_{in}, put it to $T'_{3d,1t}$;
return $T'_{3d,1t}$;

Algorithm 16: GENERATE-CONNECTION-TILE()

Input: tiletype, the type of the tile
g_{in}, the input glue
g_{out}, the output glue
e_n, the code of the north glue
maxbits, the number of binary bits to encode all of the glues $\in G_{ns}$
Output: $T'_{3d,1t}$, Tile set at temperature 1 in 3D
$T'_{3d,1t} \leftarrow \phi$;
if *tiletype = SWN2 or tiletype = SEN2* **then**
    The number of tiles to be generated at the bottom of dotted box in the figure is $(maxbits \times 2)$.
else if *tiletype = DWN2 or tiletype = DEN2* **then**
    Generate the tiles showed as the figures in the Table 4, with the input glue g_{in} and output glue g_{out}. Put all of the tiles to $T'_{3d,1t}$.
return $T'_{3d,1t}$;

Algorithm 17: GENERATE-ENCODE-TILE-FIXED()

Input: tiletype, the type of the tile
g_{in}, the input glue
g_{out}, the output glue
e_n, the code of the north glue
maxbits, the number of binary bits to encode all of the glues $\in G_{ns}$
Output: $T'_{3d,1t}$, Tile set at temperature 1 in 3D
$T'_{3d,1t} \leftarrow \phi$;
Generate the tiles with the input glue g_{in};
Generate the tiles in the *Encoding Area* and put it to $T'_{3d,1t}$;
Generate the tiles with the output glue g_{out};
return $T'_{3d,1t}$;

Probabilistic Zig-Zag in 2D

The algorithms for converting the zig-zag from temperature $\tau = 2$ to temperature $\tau = 1$ are similar to that in 3D. Using the same algorithms to encode all of the glues at the north or south of tiles with strength 1 (See Algorithm 6, 7).

The decoding tile sets are different from the zig-zag in 3D. The parameter K is introduced in the probabilistic zig-zag tile set. Figure 17, 18 shows the tile set for detecting one bit of the code by using $K(= 4)$ groups of the detect tile set. Figure 19 shows one of the complete decoding tile sets which have similar function as showed in Figure 15.

The *Encoding Area* is a bit different from that in 3D. The length for each bit of the code in the *Encoding Area* is depend on the parameter K. The length of the *Encoding Area* will be $2K \times maxbits$. The mapping between the zig-zag and probabilistic zig-zag for each of the tile types is showed in Table 4, the implemention use the similar algorithms as that used in 3D.

The success ratio of constructing zig-zag structure depends on the parameter K, but we noticed that it also depends on the number of zero in encoding code, because there exist false positive in detecting the zero bits of the encoding codes. We can select the codes which contain many one bits for the encoding code to reduce the error ratio.

The algorithms for probabilistic zig-zag in 2D are omited.

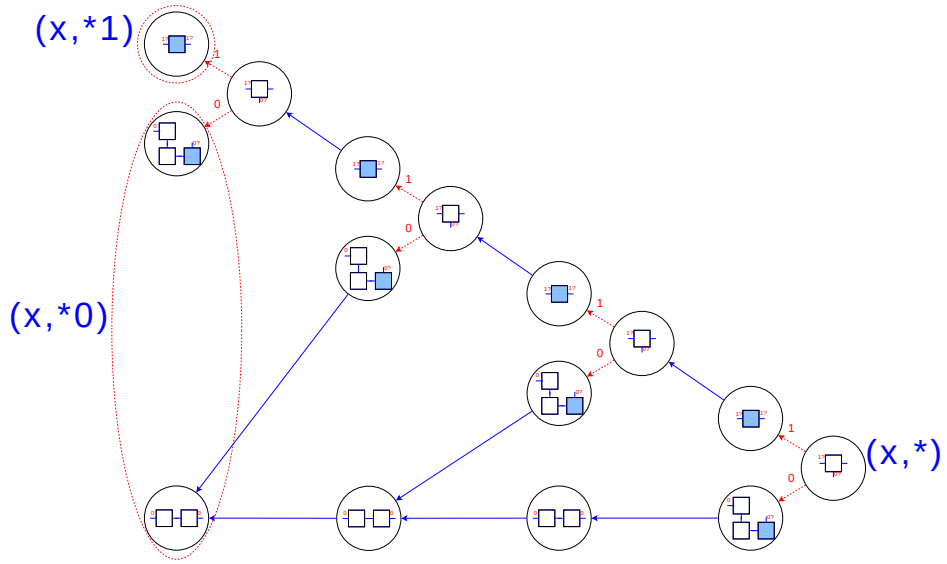


Figure 17. The tile set to decode one bit of the code (Direction left, K=4).

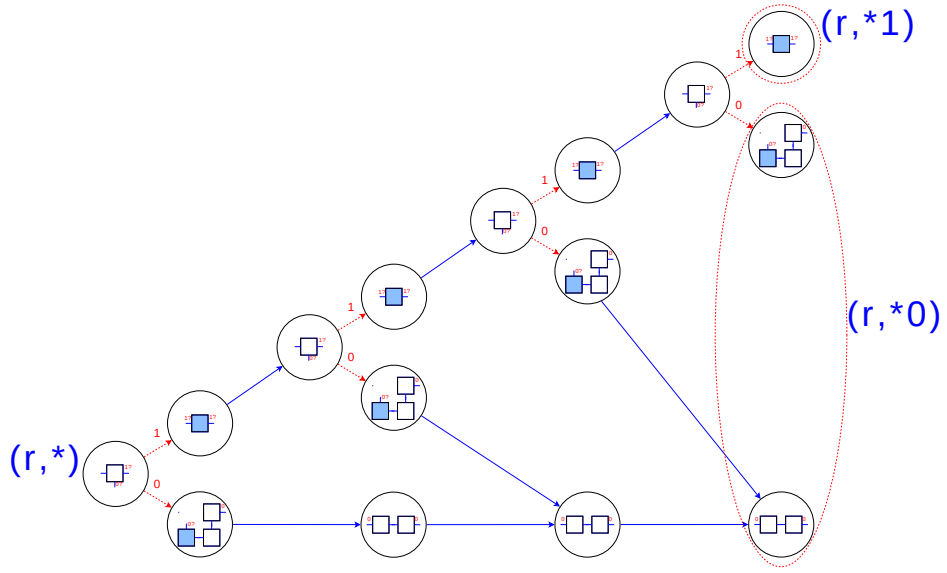


Figure 18. The tile set to decode one bit of the code (Direction right, K=4).

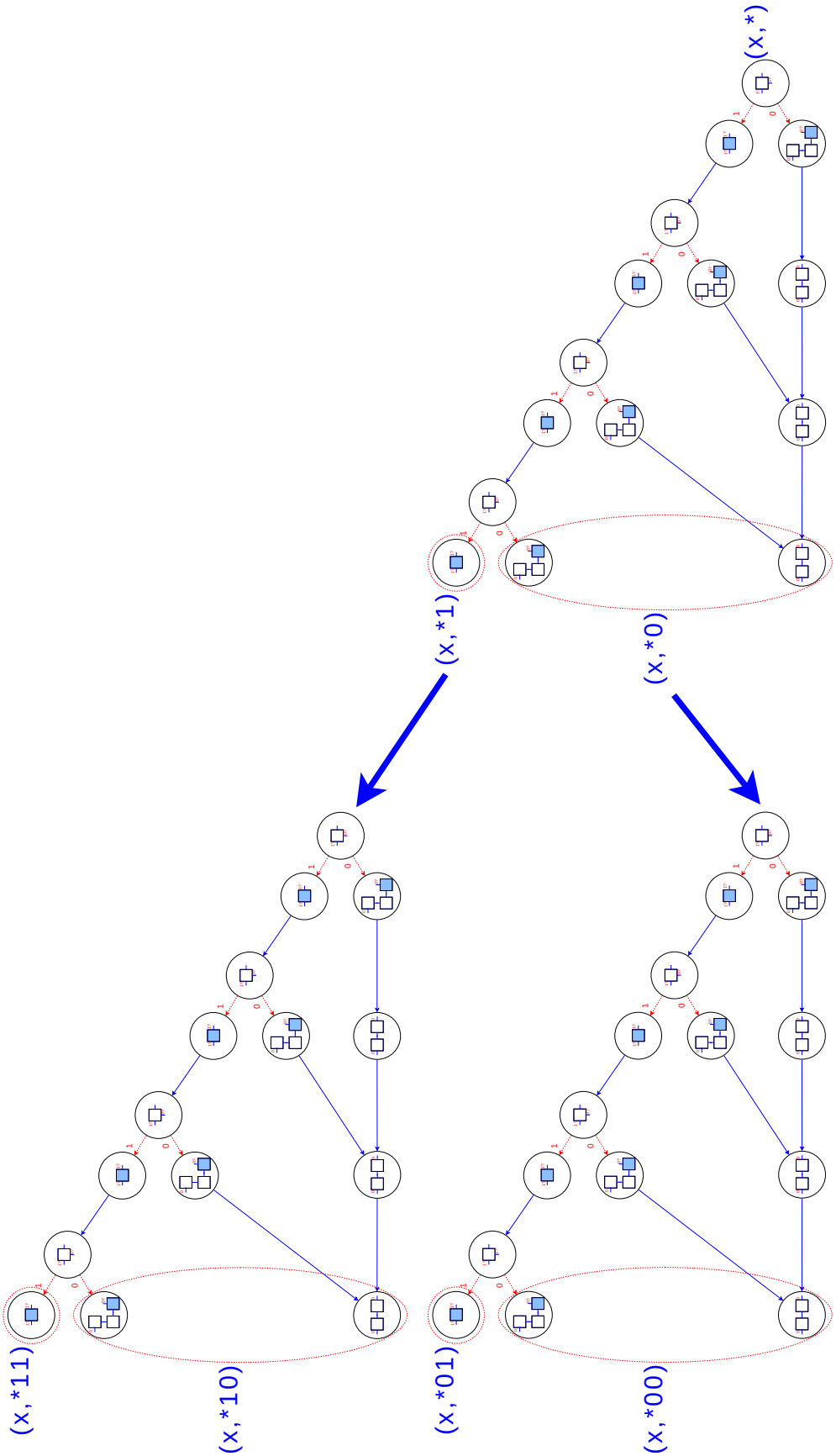


Figure 19. The logic binary tree for constructing one of decoding tile set (direction left).

Biographical Sketch

Yunhui Fu received the Bachelor of Science degree in Engineering from the Central South University in 2000 and he worked as an engineer for several electronics companies in China before moving to Texas, USA to persue for Master program. He was accepted and started graduate studies in August, 2008. His professional interests include design and analysis of algorithms, distributed systems.

Permanent Address: APT 1001, 1607 W. Schunior Street
Edinburg, TX 78541