SELF-ASSEMBLY
FOR DISCREET, FAULT-TOLERANT, AND SCALABLE COMPUTATION
ON INTERNET-SIZED DISTRIBUTED NETWORKS

by

Yuriy Brun

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

May 2008

Dedication

I dedicate this work to my parents Yefim and Tatyana Brun
for their unconditional love and support.

Acknowledgements

Alice said "Would you tell me, please, which way I ought to go from here?"
"That depends a good deal on where you want to get to," said the Cat.
"I don't much care where —"said Alice.
"Then it doesn't matter which way you go," said the Cat.
"— so long as I get somewhere,"Alice added as an explanation.
"Oh, you're sure to do that," said the Cat, "if you only walk long enough."[1]

The beginnings of my adventure in research have not been unlike Alice's travels in Wonderland. When I started, I did not have much of a concept of where to go. I did, however, have a white rabbit to follow. I knew that systems that exist in nature are far more complex and dependable than systems that we, as humans, design and build. And as engineers are obsessed with building bigger and better systems, I knew that understanding what makes complex natural systems so robust and finding a way to use that knowledge was an interesting and powerful hare of a goal, but I knew little about how to pursue this white rabbit. Of course, I was far from the first to notice the power of nature and the potential it has to improve our engineering ways, but like many, I did not know just how to get my hands around and harness that power.

As I started graduate school, I did not have a Cheshire Cat to tell me which way to go, or even that it did not matter which way I went as long as I walked far enough. I did, however, have a series of characters who guided and helped me along the way, and I now wish to acknowledge and thank them.

First, I wish to thank the academic inspirations whose work and ideas have contributed most to my research. Prof. Erik Winfree originally proposed the tile assembly model, which, in the end, is the single most important piece of the foundation on which my work rests. Prof. Winfree's work, together with that of Dr. Paul W.K. Rothemund and Prof. Leonard Adleman (Len), formed the basis of the study of self-assembly. Self-assembly is a natural process of combining elements to form complex compounds. This trio introduced the formalism needed to study self-assembly as a mathematical model and posed the right questions to ask about such systems. Without inheriting their momentum, I am unlikely to have reached my current place in research.

Prof. Michael D. Ernst (Mike) was my first graduate advisor for my Masters work at MIT. He nurtured and supported my desire to do research, taught me the ropes, and was at least partially responsible for the continuation of my graduate studies. He even tried to convey to me that I must be at least a little crazy to be going into academia, but softened that news with the fact that all those I work with will share in my insanity.

Alice tried another question. "What sort of people live about here?"
"In that direction," the Cat said, waving its right paw round, "lives a Hatter; and

[1]This and the later quote in this chapter are from Lewis Carroll (Charles Lutwidge Dodgson). Alice's Adventures in Wonderland. Macmillan and Co., London, UK, 1865.

in that direction," waving the other paw, "lives a March Hare. Visit either you like: they're both mad."
"But I don't want to go among mad people," Alice remarked.
"Oh, you can't help that," said the Cat: "we're all mad here. I'm mad. You're mad."
"How do you know I'm mad?" said Alice.
"You must be,' said the Cat, "or you wouldn't have come here."

In fact, Mike was right. All characters I have encountered in my adventures in the Wonderland of research have been to some degree insane — a wonderful property without which I believe survival in academia to be impossible.

Len, my first Ph.D. advisor, deserves special thanks. He introduced me to a formal mathematical way of studying nature. It was that ability to view aspects of biology as formal mathematical concepts that has allowed me to extract certain properties of nature-made systems and inject them into the engineering process. In addition to arming me with the hammer of mathematical formalism and logic, Len supported and guided me through explorations of various areas within chemistry, biology, mathematics, and theoretical computer science for over two years. Without his help, I would have never been able to visualize the definitions of computation in self-assembling systems — the doorway to building biologically inspired software.

Throughout my years as a Ph.D. student, my primary companion has been Dustin Reishus. There have been numerous storms that we have weathered together, and I believe it is safe to say that neither of us would be nearing the completion of our Ph.D. journeys without the other. Dustin deserves my gratitude for too many of his actions to name here, but his constant support and motivation, relentless encouragement to improve my mathematical formalism, and constant (and I mean at all hours of the night) availability to have ideas bounced off him deserve special mention. I have become so dependent on Dustin, it is not even clear in my mind that I could go off and do research without at least a photo of him to speak to. These words are unlikely to convey the proper magnitude of gratitude that Dustin deserves; however, as his and my quests for Ph.D. degrees are only the start of our academic careers, I envision having plenty more opportunities to give him well-deserved praise.

Literature buffs know very well that every adventure worthy of being told must contain a conflict and a metaphorical low-point for the protagonist. The hero must feel despair and hopelessness before collecting himself and rising to the occasion. Interestingly, I have been told that no story of earning a Ph.D. is without numerous conflicts and the only difficulty in naming a low-point is selecting a single one from among the many candidates. However, I have several friends whose relatively problem-free graduate school experience contradicts this hypothesis. Perhaps even more interestingly, each of them has abandoned the world of academia, usually by force and not by choice. (Of course, why would someone who has experienced no major setbacks in a world leave that world by choice?) Meanwhile, those who hold the most terrifying, depressing, and shriek-causing stories of epic Ph.D. conquests seem to become the most successful academics. Having summarized the above anecdotal evidence, I am proud to say that my story has a defining point of utter vulnerability, misery, and gloom, together with a hard-fought-for happy ending. In the middle of my third year, my advisor and I came to the conclusion that his and my research were moving in different directions and that we should part ways. This was not a joyous decision for me, as not only were there no professors doing the work I had been doing in my proximity, there are perhaps only half a dozen such researchers in the entire world. Without boring the reader with the sad and perhaps embarrassing details of the time I spent in the proverbial gutter, cussing the mad nature of academia, as for the better part of the next year I was a self-guided and self-supported student, I will say that the person who emerged from that gutter was a stronger, more dedicated, and, dare

I say, wiser researcher. I am inclined to remark that without the support of the aforementioned Dustin, and the later-mentioned Alaina, as well as a few others, I may have abandoned academia altogether at this low-point.

With a half-completed project and my focus lacking, Prof. Nenad Medvidović (Neno) was the right man to rescue me from becoming another casualty of the cruel academic world. The gratitude I owe Neno cannot be put down into words, but let it be said that without him I fully expect to still be falling down a bottomless well. Neno's expertise in the realm of software architecture and his kindness and patience have all combined to help me not only focus my work, but also formulate what I consider a strong contribution to the field of software engineering. He has taught me how to present my research with impact and how to orient my work to make it important to many fields. In addition, I have learned more about the art of navigating academia from Neno than I have from anyone else, and expect to continue gaining knowledge of that kind deep into my career.

There are a few more characters who must be thanked for making my Ph.D. adventure in wonderland a success. I wish to thank David Woollard for his essential work in helping me formulate a software engineering thesis, and in general being a good friend. I also would like to thank my coworkers at the Laboratory for Molecular Science: Dr. Nickolas Chelyapov, Bilal Shaw, Manoj Gopalkrisnan, and Dr. Pablo Moisset de Espanés for their help in my DNA experiments and my coworkers at the Center for Systems and Software Engineering Prof. Sam Malek, Dr. Chris Mattmann, George Edwards, Chiyoung Seo, and Daniel Popescu for their support. I also thank several professors who have been on my various committees, in addition to those already mentioned: Prof. David Kempe, Prof. Krishna Nayak, Prof. Christoph von der Malsburg, and Prof. Clifford Neuman.

No person can survive the brutal rejections of academia without strong personal support. For more than two years now, I have received such support from Alaina Schlinker, who has stood by me no matter what and always received me with open arms, whether my latest endeavor in academia was a success or a failure. To her I owe that same gratitude a lost and wounded soldier owes his family at home: the unbroken and unremitting hope that everything will be alright in the end.

This brings me to the final set of people who were absolutely crucial to my academic career thus far: my family. From a very young age, my parents have encouraged me to pursue my interests in research in the fields of mathematics and sciences. Their struggles throughout their lives, including immigrating halfway around the world to provide me with opportunities I would never have had in Russia, have done nothing less than opened doors and laid out the carpet for me to walk on. Whatever credit I deserve for my research and hard work, they deserve at a minimum ten times that much credit. The support, help, encouragement, and love I have received from my father, Yefim Brun, my mother, Taytana Brun, and my sister, Dina Brun, have been unconditional and invaluable. Without them, I would never have been writing this sentence.

Thank you all for your help in making my dreams come true. I can only hope that helping me has been remotely as gratifying for you as receiving your help has been for me. I have sincere faith that I will spend the rest of my life paying your kindness back and look forward to making your aspirations as easy to achieve as you have made mine.

Table of Contents

List Of Figures

Abstract

When engineers compare biological and software systems, the former come out ahead in the majority of dimensions. For example, the human body is far more complex, better suited to deal with faulty components, more resistant to malicious agents such as viruses, and more adaptive to environmental changes than your favorite operating system. Thus it follows that we, the engineers, may be able to build better software systems than the ones we build today by borrowing technologies from nature and injecting them into our system design process.

In this dissertation, I present an architectural style and accompanying implementation support for building distributed software systems that allow large networks, such as the Internet, to solve computationally intensive problems. This architectural style, the tile style, is based on a nature's system of crystal growth, and thus inherits some of nature's dependability, fault and adversary tolerance, scalability, and security. The tile style allows one to distribute computation onto a large network in a way that guarantees that unless someone controls a large fraction of the network, they cannot learn the private data within the computation or force the computation to fail. These systems are highly scalable, capable of dealing with faulty and malicious nodes, and are discreet since every sufficiently small group of nodes knows neither the problem nor the data.

The tile style is based on a formal mathematical model of self-assembly. In order to leverage this model to build software, I define the notion of self-assembling computation and develop systems that compute functions such as adding, multiplying, factoring, and solving NP-complete problems *SubsetSum* and SAT. For each system, I prove its correctness, compute its probability of successful computation, and show that its running time and tileset size are asymptotically optimal.

I use the mathematical nature of the tile assembly model to present a formal mathematical analysis of the tile style, proving that software systems built using this style are discreet, fault- and adversary-tolerant, and scalable. I further implement a tile-style system and use it to distribute computation to empirically evaluate the tile style's utility.

Chapter 1

Introduction

Computer science has taken on the study of computation on several levels. Theoreticians try to devise models that can compute mathematical functions. Software engineers use previously defined models, most notably the traditional silicon computer, to design systems that manipulate data in controlled ways. Software architects attempt to generalize the structure of computer software systems to extract patterns and styles that can be applied to the design of future programs, allowing for the creation of increasingly more complex systems. Recently, chemists and molecular biologists have encountered a surprising notion: molecules and compounds can compute [3, 104]. In fact, molecules and compounds can compute every function that a traditional computer program can compute [8, 105].

The observation that simple objects, such as molecules and compounds, can compute, has led scientists to create a new area of research, called *self-assembly*, that studies the nature of simple components with simple interfaces coming together to exhibit complex behavior [4]. This field has produced a formal model of self-assembly known as the *tile assembly model* [106].

One of the driving forces behind exploring self-assembly systems in the realm of computation is that many self-assembly systems exhibit great potential for fault tolerance [14, 39, 83, 99, 107, 108]. In fact, self-assembly and other natural processes have many additional properties such as dependability, scalability, and security that surpass our most intricate human-designed systems. It follows that if we are able to extract certain properties from biological and other natural systems and inject them into our system design process, we may be able to build more complex, dependable, and efficient systems than we do today.

In this dissertation, I present a study of the natural process of self-assembly resulting in a series of asymptotically optimal (in terms of size and speed) tile assembly systems that compute functions such as addition, multiplication, factoring, and also solve NP-complete problems. I further develop a software architectural style, the tile style, for computing on large networks, such as the Internet. The tile style allows users to distribute computation over untrusted nodes in a discreet (without telling any small group of computers the problem being solved) and scalable manner, and be guaranteed the receipt of a correct answer with an arbitrarily high probability even when the underlying network is partially faulty or malicious.

1.1 Contributions

This dissertation brings forward contributions in two fields of study: self-assembly and software architecture. While these fields are typically considered quite distant, I introduce a methodology for using self-assembly systems to guide software architecture, bridging the two fields and uncovering beneficial cooperation between research in these areas.

1.1.1 Formal Study of Self-Assembly

In the area of the formal study of self-assembly, I will define deterministic and nondeterministic computation in the tile assembly model, as well as the notion of deciding sets. Then, I will present and analyze tile assembly model systems that compute mathematical functions, such as addition, multiplication, and factoring, and solve NP-complete problems. For each of those systems, I will formally prove that the system computes its intended function, analyze the running time and tileset size (two measures previously identified as important in tile systems [4]) and probability of successful computation (a measure I identify as important for nondeterministic computation).

I will pinpoint some techniques in designing computational tile systems that can be applied to future designs of such systems, as well as designs of other kinds of tile systems.

1.1.2 Software Architecture

In the area of software architecture, I will design a software architectural style, called the tile style, that allows the distribution of certain types of computation over a large insecure network in a discreet manner.

The tile style is particularly applicable to problems that are computationally intensive and easily parallelizable. Computationally intensive problems are ones that a single computer is unlikely to solve quickly, while easily parallelizable problems are ones that inherently yield a large number of parallel threads. For example, all NP-complete problems have both of those properties [98]. Further, the style is applicable to users who desire discreetness and have access to large but unreliable networks. By discreetness, I mean that the user does not want others to find out the input or the algorithm. By large but unreliable network, I mean a network, such as the Internet, that is partially or entirely outside of the user's control, and perhaps even hostile.

For example, someone who may want to use the tile style is a pharmaceutical company that has finished running a large clinical trial and has collected data that need to be analyzed. The data are sensitive and the company does not want that data to become public prematurely, but it wishes to use a large public network in order to process the data. What's more, not only are the data private, some of the analysis algorithms may be proprietary as well. The company needs a way to distribute the computation on an insecure network while making neither the data nor the algorithms public.

I will present the tile architectural style, and evaluate it theoretically and empirically. Some of the desired properties of systems built using the tile style I will prove mathematically. For example, I can guarantee that given a certain fraction of the nodes on the network being faulty or hostile, the probability of successful correct computation can be bounded arbitrarily closely to 1 at a relatively small cost in execution speed. I will also be able to prove that no small group of nodes will be able to learn the algorithm of the computation or the input to that algorithm. Also, I will present arguments for why the systems built using the tile style are scalable. Finally, I will design a software system based on the tile style to solve NP-complete problems and evaluate it empirically on a network to demonstrate the properties of discreetness, fault tolerance, and scalability.

1.2 Structure of This Dissertation

This dissertation is broken up into two parts. Part I discusses the theoretical work on self-assembly, including background and related work in Chapter 2, definitions of tile systems and computation in Chapter 3, and the designs and careful analysis of five computational tile systems that add,

multiply, factor, and solve two NP-complete problems in Chapter 4. This part answers some open questions in the field of self-assembly and can be read as a standalone document.

Part II discusses work on distributing computation on a large network, including background on how such distribution is related to self-assembly and work related to distributed systems and software engineering in Chapter 5, the definitions of the tile architectural style for building such distributed systems in Chapter 6, and the analysis and evaluation of tile-style systems in Chapter 7. This part depends heavily on the theoretical work in self-assembly discussed in Part I and leverages the tile systems from Chapter 4.

Chapter 8 summarizes my work and its contributions and presents several future directions for the research, some of which I plan to follow. It also briefly describes my other attempts at using self-assembly to solve interesting mathematical problems that may direct future research.

Part I

Theoretical Study of Self-Assembly

Chapter 2

Background and Related Work in Self-Assembly

Self-assembly is a crucial process by which systems in nature form on all scales. Atoms self-assemble to form molecules, molecules to form compounds, and stars and planets to form galaxies. One manifestation of self-assembly is crystal growth: molecules self-assembling to form crystals. Crystal growth is an interesting area of research for computer scientists because it has been shown that, in theory, under careful control, crystals can compute [105]. The field of DNA computation demonstrates that DNA can be used to compute [3], solving NP-complete problems such as the satisfiability problem [22, 23]. Winfree showed that DNA computation is Turing-universal [104]. While DNA computation suffers from relatively high error rates, the study of self-assembly shows how to utilize redundancy for error correction [14, 39, 83, 108]. Researchers have used DNA to assemble crystals with patterns of binary counters [12] and Sierpinski triangles [88].

In this chapter, I will discuss work related to a formal model of crystal growth called the tile assembly model. I will first give an intuitive explanation of the tile assembly model (more formal definitions will follow in Chapter 3) and then demonstrate why the model is Turing universal. I will then show some results on the complexities of systems that compute or assemble shapes and on fault tolerance of tile systems.

2.1 Intuitive Definitions of the Tile Assembly Model

Intuitively, the tile assembly model has *tiles*, or squares, that stick or do not stick together based on various *binding domains* on their four sides. Each tile has a binding domain on its north, east, south, and west sides, and may stick to another tile when the binding domain on the abutting sides of those tiles match and the total strength of all the binding domains on that tile exceeds the current temperature. The four binding domains, elements of a finite alphabet Σ, define the type of the tile. The strength of the binding domains are defined by the *strength function* g. The placement of some tiles on a 2-D grid is called a *configuration*, and a tile may *attach* in empty positions on the grid if the total strength of all the binding domains on that tile that match its neighbors exceeds the current *temperature* (a natural number). Finally, a *tile system* $\mathbb{S}$ is a triple $\langle T, g, \tau \rangle$, where T is a finite set of tiles, g is a strength function, and $\tau \in \mathbb{N}$ is the temperature, where $\mathbb{N} = \mathbb{Z}_{\geq 0}$.

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

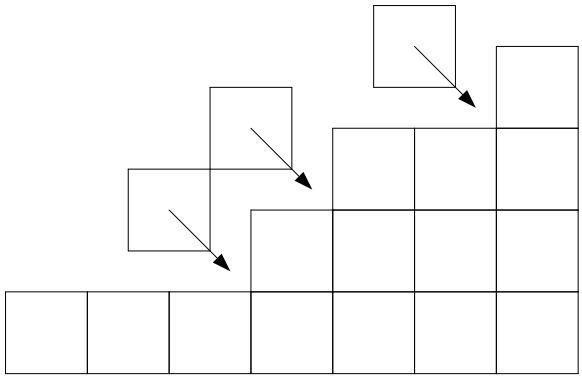


Figure 2.1: Temperature two systems: a set of special systems that operate at temperature 2, whose strength function is identically 1, and in which tiles attach only when their south and east neighbors are present.

2.2 Temperature Two Systems

Winfree brought researchers' attention to a special set of tile assembly systems that I call the *temperature two systems*. These are systems that operate at temperature 2, whose strength functions are identically 1, and that, on certain seeds, produce unique final configurations. I will discuss some properties of these systems in Section 4.1.1, but the essential property of importance is that every tile in these systems can only attach when its east and south neighbors are already present in an assembly. Figure 2.1 shows an example execution of such a system.

Winfree showed that temperature two system are Turing universal [105]. For that reason, some researchers have focused their attention on these systems.

2.3 Turing Universality

Winfree's constructions develop a way to convert a 1-D cellular automaton into a temperature two tile system that computes the same function. 1-D cellular automata are known to be Turning universal because they can emulate Turing machines. A 1-D cellular automaton is an infinite line of blocks with finite internal state that at every time step change their state based on their neighbors' states. It is fairly straightforward to see how a tile system could represent a 1-D cellular automaton in a single row, and compute the state of the automaton at the next time step in the row above. Continuing the process, tiles can emulate a universal 1-D cellular automaton.

I will present here a construction that converts a Turing machine to a tile system that computes the same function as the Turning machine. In essence, the tile system emulates the Turing machine by writing out all the instantaneous Turing machine descriptions [98] at every Turning machine step. While this construction is my own, the result is essentially identical to that shown by Winfree; however, it will give me a better sense of the size of the tile system necessary to emulate a Turning machine.

A Turning machine is a simple computing machine that has a head capable of reading a single element at a time on an infinite tape. Internal to the Turning machine is a finite state machine control with a single active state at each time. The machine can follow simple transition rules based on the symbol on the tape (the tape contains the symbols of some finite alphabet) under its head to transition to another control state, write a new symbol on the tape, and move the head to the right or to the left. I refer the reader to [98] for more details on Turning machines. The

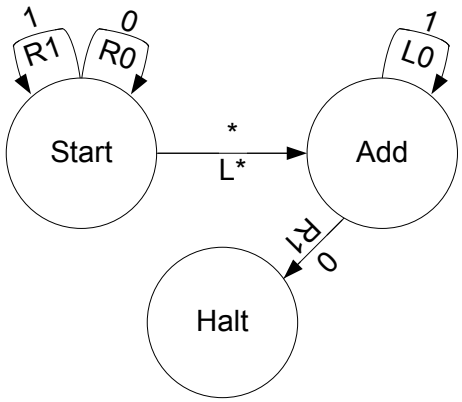


Figure 2.2: A Turning machine with three states. This Turing machine adds 1 to a number encoded in binary on a tape, and then halts. Each transition arrow is labeled above with the symbol the head must read on the tape to follow that transition arrow, and labeled below with either L or R, indicating moving the head left or right, respectively, and the symbol to write in old position on the tape.

significance of the Turing machine is that it can compute every computable function, and thus no more "powerful" computer exists (although faster or more efficient ones may exist). Figure 2.2 shows an example Turing machine. This Turing machine has three states (start, add, halt), and a three-character alphabet (0, 1, $\star$). When placed at the head of a tape with 0s and 1s written one after the other, the last followed by a $\star$, the Turing machine will scan down the tape until the $\star$, add 1 to the binary number encoded to the left of the $\star$, and halt.

A tile system can emulate a Turing machine. For example, Figure 2.3 shows how tiles emulate the Turing machine from Figure 2.2. The system starts with a single row seed (top left) and tiles add on top of that seed to form more rows (2, 3, 6, and 10 rows, respectively, clockwise from the top left). Note that the original row encodes the binary number $01011_2 = 11$ and the final top row encodes $01100_2 = 12 = 11+1$. The system computes the function $f(\alpha) = \alpha+1$ for some reasonable definition of "computes" (I will formally define the notion of computation in Chapter 3). Note that this system is similar to a temperature two system, but is not quite a temperature two system because the strength function is not identically 1. The binding domains on the north side of the tiles highlighted in yellow, the tiles that indicate the position of the head in each instantaneous Turing machine description, must have a strength of 2.

My constructions are interesting because they show that the tile assembly model is Turing universal, and further, for every Turing machine, including a universal Turing machine, there exists a tile system with $\Theta(1)$ tiles that emulates it. However, while the constant bound is important, in some sense these systems are quite large: the number of tile types in a system is $\Omega\left(|Q| \cdot |\Sigma|\right)$, where Q is the set of states of the finite control and Σ is the finite alphabet. Note that I am not being terribly formal with these quantities (e.g., a Turing machine really has two different alphabets, Σ and Γ), but my quantities give the proper idea of the asymptotic growth of the tileset. For example, the tile system from Figure 2.3 would have over 200 tile types, while theoretically, a tile system that counts in binary, and thus adds 1 on each row of an assembly, can use as few as 7 tile types [57].

The main advantage of Winfree's universal tile system constructions is that they are temperature two systems, whereas mine are not. This fact allows the application of many fault tolerance techniques to his constructions, as described in Section 2.6. On the other hand, given a Turing

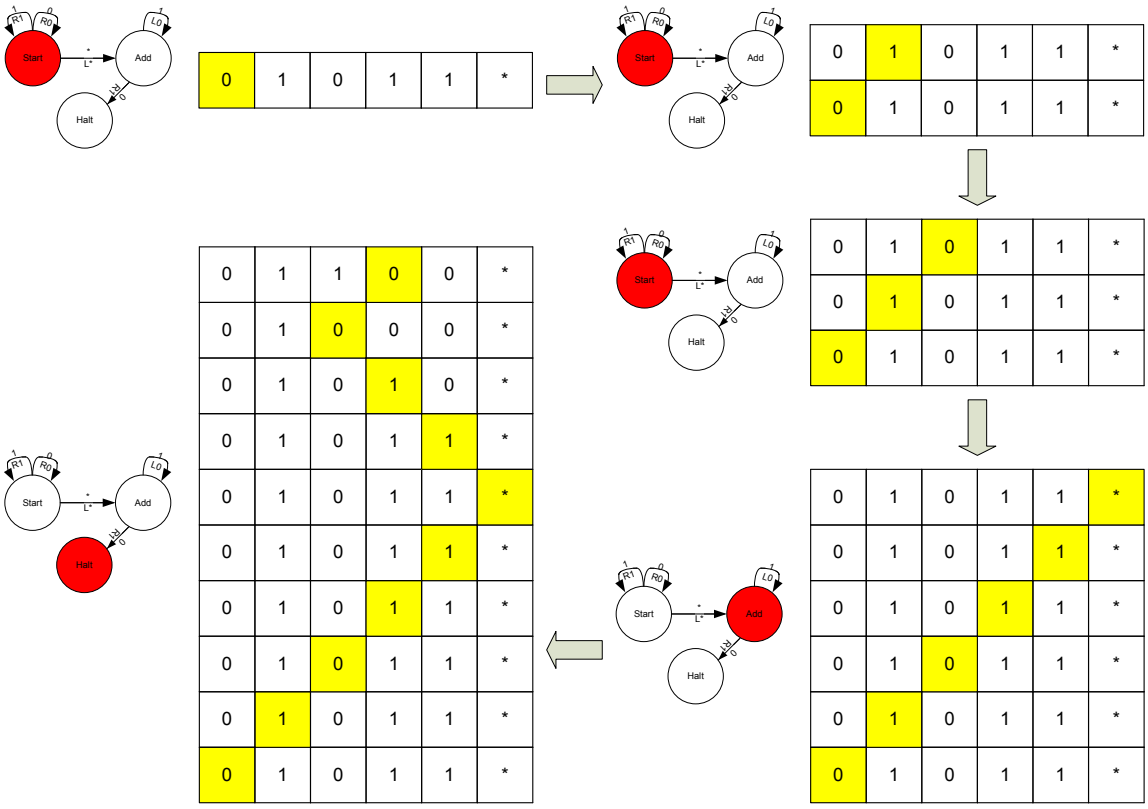


Figure 2.3: A tile system simulating the Turing machine from Figure 2.2 (in five steps, clockwise). Each row of tiles is a single instantaneous description of the Turing machine. The highlighted yellow tile indicates the position of the head. The diagram of the Turing machine to the left of the tiles have the current state highlighted in red, and is only shown for the reader — the tiles do not have a notion of the actual Turing machine. The system starts with a single row of tiles and grows rows upward. Note that no row has to complete before the row above it starts forming — a tile may attach as soon as two of its neighbors are present.

machine, Winfree's conversion of that Turing machine to a 1-D cellular automaton and then to a tile system will almost certainly result in a tile system with a larger tileset and one that constructs larger assemblies than my direct conversion from a Turing machine to a tile system.

Other ways of showing that tile systems are Turing universal exist. One such way [105] is to show that a tile system can simulate Wang tiles [103], which Robinson showed to be universal [85]. Meanwhile, Adleman et al. [8] used yet another approach to show that the tile assembly model at temperature 1 is also Turing universal.

To make computational tile systems using the constructions presented by Winfree, myself, or others is not unlike programming using Turning machines. The fact that you can solve all computable problems with a Turning machine is a powerful statement; however, programs written using Turing machines are in many ways inferior to ones written using higher-level languages such as Java. Higher-level languages allow for such concepts as abstraction, memory management, type-checking, and many others, which facilitate the construction of large and complex systems. Similarly, the statement that tile systems can emulate Turing machines should not be the end-all of the exploration of computational tile systems.

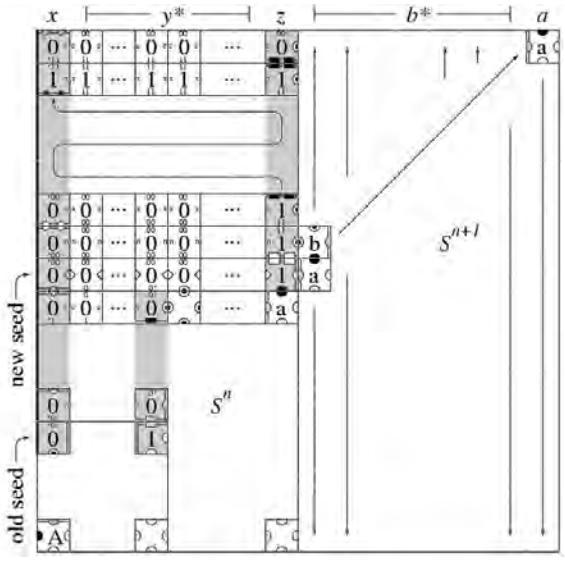


Figure 2.4: The schematic of an optimal assembly of an $n \times n$ square. This approach uses a theoretically optimal $\Theta\left(\frac{\log n}{\log \log n}\right)$ tile types to seed two binary counters, and a constant number of tiles to fill in those counters and the rest of the square (adopted from [89]).

2.4 Complexity of Tile Systems

In studying computer programs, or algorithms, computer scientists have identified several interesting measures: the time complexity of the algorithm, the space complexity of the algorithm, and the size of the algorithm. When considering tile systems that execute algorithms, there are related measures that are equally as important. For example, the size of a minimal tileset of a tile system that can compute a certain function is related to the size of a smallest program that computes that function and the minimum assembly time for that tile system is related to the time complexity of that program. Adleman has emphasized studying the number of steps it takes for an assembly to complete (assuming maximum parallelism) and the minimal number of tiles necessary to assemble a shape [4]. He answered these questions for n-long linear polymers. Adjusted to my definition of assembly time (from Chapter 3), the least number of steps required to build an n-long linear polymer is $\Theta(n)$ (Adleman's definition involves reversible tile assembly and considers tile concentrations and probabilities of attachment). The smallest tileset required for the task is of size $\Theta(n)$ tiles [5].

Adleman also proposed studying the complexity of tile systems that can uniquely produce $n \times n$ squares. A series of researchers proceeded to answer the questions: "what is a minimal tileset that can assemble such shapes?" and "what is the assembly time for these systems?" They showed that the minimal tileset is of size $\Theta\left(\frac{\log n}{\log \log n}\right)$ and the optimal assembly time is $\Theta(n)$ [6, 7, 89]. Figure 2.4 shows a diagram of the assembly of a square. A key issue related to assembling squares is the assembly of small binary counters. Figure 2.4 shows a square being built with a vertical counter. The counter starts with only $\Theta\left(\frac{\log n}{\log \log n}\right)$ unique tiles along an edge and counts out an $(n - \log n)$-long side of the square, before a constant number of tiles fill in the remaining space. The counter itself can also be created with only a constant number of tiles (and a $\Theta\left(\frac{\log n}{\log \log n}\right)$ tile seed). In fact, a counter can have as few as seven tile types [57]. Figure 2.5 shows the seven tile types necessary to count.

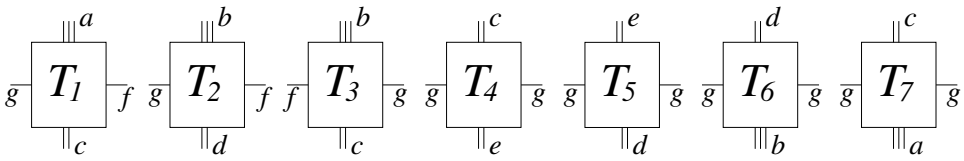


Figure 2.5: The seven tiles of a binary counter (adopted from [57]). The number of lines crossing the binding domains indicate the strength function value on those binding domains.

Soloveichik et al. [100] studied assembling all decidable shapes in the tile assembly model and found that the minimal set of tiles necessary to uniquely assemble a shape is directly related to the Kolmogorov complexity of that shape. Interestingly, they found that for the result to hold, scale must not be a factor. That is, the minimal set of tiles they found builds a given shape (e.g., square, a particular approximation of the map of the world, etc.) on some scale, but not on all scales. Thus they showed that smaller versions of the same shape might require larger sets of tiles to assemble. The reason the scale must not be a factor is that their constructions convert a Turning machine that can describe a shape (e.g., write out the coordinates of the pixels of the shape) by first emulating the Turing machine using tiles and then building the shape around that Turing machine assembly.

While my definition of the assembly time assumes maximum parallelism, some other definitions explore probability models of tiles attaching. Thus in my systems, constructions are in some ways analogous to traditional computer programs, and their running times are polynomially related to the running times of Turing machines. Baryshnikov et al. [13] began the study of fundamental limits on the time required for a self-assembly system to compute functions by considering models of molecular self-assembly and applying Markov models to show lower limits on assembly times. I do not go into great detail of this work because it does not apply directly to my systems, though I do mention it because it brings useful information to the table when considering implementations of the tile assembly model, especially implementations driven by diffusion of molecules or other such processes.

Researchers have also studied variations of the traditional tile assembly model. Aggarwal et al. [9] and Kao et al. [64] have shown that changing the temperature of assembly from a constant throughout the assembly process to a discrete function reduces the minimal tileset that can build an $n \times n$ square to a size $\Theta(1)$ tileset. And along with Dustin Reishus, I have shown that allowing the temperature to change only once is sufficient for creating tile systems that perform simple robotics tasks [34].

2.5 Other Computational Tile Systems

Some early attempts at computing nondeterministically using the tile assembly model include a proposal by Lagoudakis et al. [69] to solve the satisfiability problem. They informally define two systems that nondeterministically compute whether or not an n-variable Boolean formula is satisfiable using $\Theta(n^4)$ and $\Theta(n^2)$ distinct tiles, respectively. The former system encodes each clause as a separate tile, and the latter system encodes each pair of literals as a separate tile. In a DNA implementation of even the asymptotically smaller system, to solve a 50-variable satisfiability problem, one would need on the order of 2500 different DNA complexes, while current DNA self-assembly systems have on the order of 10 different complexes. In contrast, the systems I designed for solving NP-complete problems use $\Theta(1)$ distinct tile types (64 for satisfiability) and assemble in time linear in the input. The Lagoudakis et al. result was largely eclipsed by Winfree's universality

proof that showed that a tile system with a constant number of tiles could perform universal computation; although for small Boolean formulae, the Lagoudakis et al. approach may use fewer tiles.

Barish et al. [12] have demonstrated a DNA implementation of a tile system that copies an input and counts in binary. Similarly, Rothemund et al. [88] have demonstrated a DNA implementation of a tile system that computes the *xor* function, resulting in a Sierpinski triangle. These systems grow crystals using double-crossover complexes [55] as tiles.

Cook et al. [43] have explored using the tile assembly model to implement arbitrary circuits. Their model allows for tiles that contain gates, counters, and even more complex logic components, as opposed to the simple static tiles used in the traditional tile assembly model. While they speculate that the tile assembly model logic may be used to assemble logic components attached to DNA, my assemblies require no additional logic components and encode the computation themselves. It is likely that their approach will require fewer tile types and perhaps assemble faster, but at the disadvantage of having to not only assemble crystals but also attach components to those crystals and create connections among those components. Nevertheless, Rothemund's work with using DNA as a scaffold may be useful in attaching and assembling such components [87].

Some of the tile systems I discuss will require an exponential number of components. Experimental research [3,22] has shown that it is possible to work with an exponential number of components and even to solve NP-complete problems, for which we do not know of strictly polynomial-time algorithms.

2.6 Fault Tolerance

While the tile assembly model is a mathematical model of self-assembly, in its most basic form tiles only stick to assemblies when the sum of the strengths of the binding domains exceeds the temperature. That is to say, it assumes that one can design systems such that tiles stick only when their sides' binding domains match, and the tile system never makes the mistake of attaching an incorrect tile. In practice, in many implementations of the tile assembly model, e.g., DNA implementations, this assumption does not hold — tiles with mismatched binding domains and tiles whose binding domains' strengths fall short of the temperature manage to attach to assemblies. For this reason, Winfree extended the tile assembly model to include error rates and called the error-free model the *abstract* tile assembly model and the model with error rates the *kinetic* tile assembly model.

In the kinetic tile assembly model, every binding domain has a certain probability of being mistaken for a different binding domain. This probability is called the *error-rate*, and is commonly represented by the symbol ϵ. In the abstract tile assembly model, every tile that attaches to an assembly remains in the assembly forever; however, in the kinetic tile assembly model, every tile may detach from an assembly, and the probability of detachment is inversely proportional to the sum of the binding domain strengths with which that tile is attached.

A notion related to the difference between the abstract and kinetic tile assembly models is reversibility [5]. *Irreversible* tile systems do not allow tiles to fall off an assembly, just as the abstract tile assembly model. *Reversible* tile systems are more accurate models of real molecular biology and chemistry, and allow on and off rates for attachments and detachments, respectively. The rates between two tiles can depend on the strength of interaction between those tiles.

Using the kinetic tile assembly model, first Winfree and then others were able to design mechanisms to reduce the assembly error rates in tile systems. Most of the error correction mechanisms increase the number of tile types in a system by a constant factor, and lower the expected error

rate exponentially. That is, applying one such mechanism to a system with n tiles and an inherent error rate of ϵ results in a system with cn tiles, for some integer constant $c \geq 2$ and an error rate of $\epsilon^{\Theta(c)}$.

The error correction work to date was largely motivated by the fact that DNA implementations of the tile assembly model could not grow large crystals because of the inherent error rates and systems with built-in error correction may allow the creation of larger crystals. As I will show in Chapter 7, the same mechanisms can be used to correct errors in assemblies with no inherent binding domain mistakes, but with tiles that are "malicious" and attempt to break the computation. There, if an instantiation of a tile system has an ϵ fraction of all the tiles misbehave, the mechanisms will increase the number of tile types by a constant factor (or quadratically in one case) and exponentially lower the probability of an entire assembly failing.

The rest of this section will describe several error correction mechanisms. The error correction techniques I describe are all applicable to temperature two systems, which is the extent for which I will need them. Note that some techniques may also be applicable to other systems.

Most of the mechanisms I describe are transformations that convert tile systems into other tile systems that in some way perform the same tasks as the original systems, but are less likely to commit errors. My plan is to use these transformations as off-the-shelf mechanisms to convert systems I create that compute functions into more fault-tolerant systems that compute the same functions.

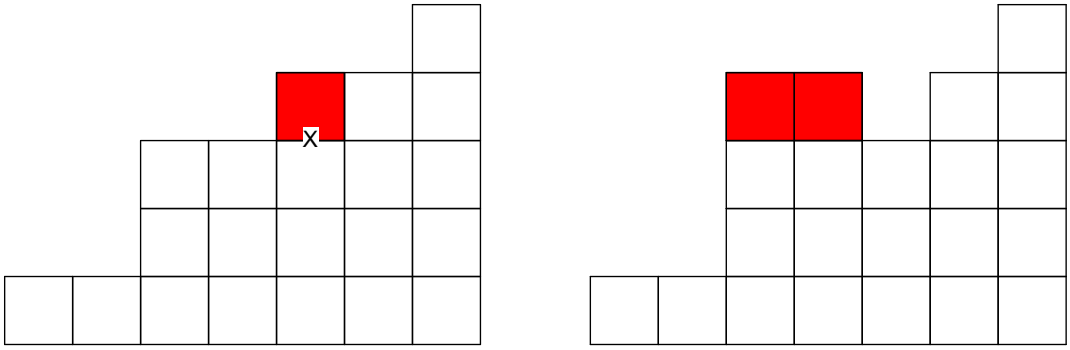
2.6.1 Types of Possible Errors

Winfree et al. [108] have identified two different types of errors that occur in temperature two systems: *growth* errors and *spontaneous nucleation* errors. Growth errors occur when incorrect tiles are incorporated into the growing structure (these happen in practice in DNA self-assembly with error rates ranging from 1% to 10%). Spontaneous nucleation errors occur spuriously and do not involve the seed tile. Spontaneous nucleation errors are essentially conglomerates of tiles that start growing by themselves, rather than from a seed. With DNA assembly, it is important to concern oneself with sponteneous nucleation errors because otherwise it may be very difficult to pick error-free assemblies from erroneous ones; however, in many implementations of self-assembly, the control over the seed is great enough that spontaneous nucleation errors do not pose a problem.

Chen et al. [39] noticed another type of growth error, the *facet nucleation* error. The facet nucleation errors occur when a tile attaches to a temperature two system assembly in a position that is not ready for a new attachment (e.g., the east and south neighbors of that position have not both attached). While normally, such a tile would have a high probability of detaching from the assembly because it can only be attached by at most one strength 1 binding domain, it is possible for a second tile to attach and "lock in" the original mistake tile, in a way such that both tiles are attached with strength 2 and are unlikely to detach. Because I will not be much concerned with spontaneous nucleation errors, I will refer to the facet nucleation errors as the nucleation errors.

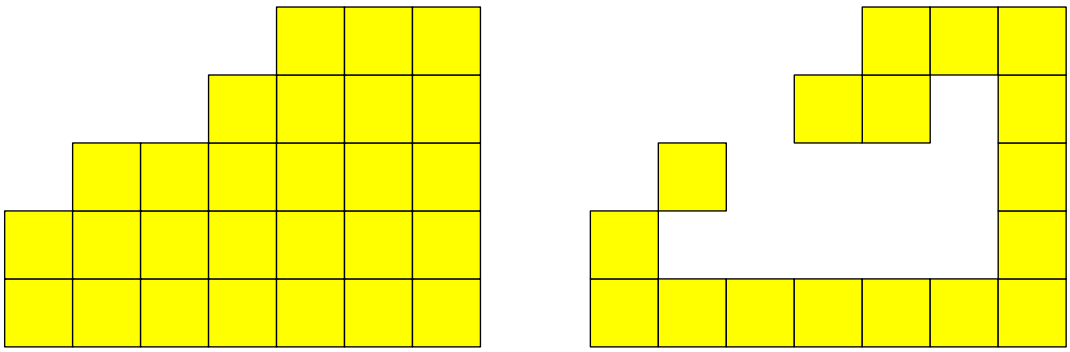
Figure 2.6(a) shows an example of a growth error (the X represents a mismatched binding domain) and Figure 2.6(b) shows an example of a facet nucleation error. The incorrectly attached tiles are highlighted in red. Note that with a nucleation error, there are no binding domain mismatches.

Finally, a growing crystal may experience an event of having a large collection of its tiles detached. These errors are different from growth and nucleation errors in that they affect a large number of tiles at once, and may occur in locations other than the edge of the assembly. I call these errors *shotgun* errors because they resemble the effect of a shotgun unloaded at the assembly. Figure 2.7 shows an example of a shotgun error.



(a) (b)

Figure 2.6: Two possible types of errors in a temperature two tile assembly system. A growth error (a) occurs when incorrect tiles are incorporated into the growing structure (the X represents a mismatched binding domain). A nucleation error, the kind defined by Chen et al., (b) occurs when a tile attaches in a position that is not ready for a new attachment (e.g., the east and south neighbors of that position have not both attached). The tiles that attach incorrectly are highlighted in red. Note that with a nucleation error, there are no binding domain mismatches.



(a) (b)

Figure 2.7: A shotgun error occurs in an assembly that has grown for some time (a) when a large number of tiles detach from the middle of the assembly (b).

2.6.2 Growth Error Correction

Winfree et al. [108] proposed a mechanism they called *proofreading* for correcting growth errors in temperature two systems. In reality, they proposed a number of mechanisms, one for each integer $k \geq 2$. I will first explain how the $k = 2$ mechanism works and then generalize for all k.

2.6.2.1 2×2 Proofreading

For $k = 2$, the mechanism is called 2×2 proofreading and consists of replacing every tile in the system with $2 \times 2 = 4$ tiles. Figure 2.8 shows an example of a tile's transformation to four tiles. The tile is represented by four new tiles, arranged in a 2×2 square, with unique internal binding domains. Each of the original tile's binding domains becomes represented by two new binding domains (e.g., binding domain a leads to binding domains $a1$ and $a2$), and the exposed binding domains of the 2×2 square are labeled with the new binding domains as shown in Figure 2.8.

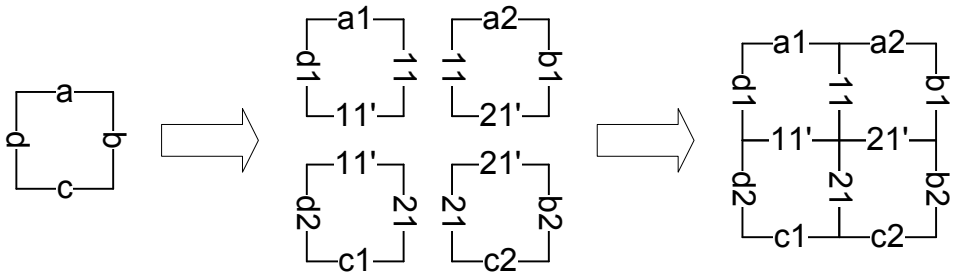


Figure 2.8: A 2×2 proofreading mechanism transforms each tile type in a system into four tile types, such that the four can attach uniquely to each other.

Note that the new 2×2 square contains essentially all the information the original tile contained, and some extra internal binding domains. It follows that if all tile types in a temperature two system are transformed as I described, the assemblies made by the new temperature two system will be four times larger (two times larger in each of the two dimensions) than the original system's assemblies, but will otherwise encode the same information.

As a reminder, growth errors occur when an incorrect tile, one whose east or south binding domain does not match its neighbors' binding domains, attaches to an assembly. Suppose that the probability of a tile attaching, resulting in a growth error, in a temperature two system is ϵ. Then the probability that some n tile assembly completes correctly is $(1-\epsilon)^n$, assuming independence of attachment errors. Assuming that the same per-tile error rate carries over to the system with four times as many tiles (the 2×2 proofreading system), the probability of the assembly representing the same information assembling correctly would be $(1-\epsilon)^{4n}$ if the errors were still independent. However, the errors are no longer independent. If an incorrect tile attaches in one of the four positions, the 2×2 representation of the tile will not be able to assemble, and the computation will not proceed until either the incorrect tile detaches, or a second growth error occurs in an adjacent position. Thus if one growth error were to survive until the final assembly is complete, a second error would have to occur at a nearby location, thus essentially making the error rate ϵ^2 per every 2×2 block. Thus the true probability of the 2×2 proofreading assembly assembling correctly is $\left(1-\epsilon^2\right)^n$. Thus at a linear cost in assembly size (and speed), the 2×2 proofreading mechanism squared the probability of a growth error.

2.6.2.2 $k \times k$ Proofreading

In the general case, a tile can be transformed into k^2 tiles by being represented by a $k \times k$ block, with unique internal binding domains, and k versions of each of the original exposed binding domains. Figure 2.9(a) shows an example with $k = 3$ and Figure 2.9(b) shows an example for an arbitrary k.

By the same argument as before, by transforming n tile types into $k^2 n$ tile types, the $k \times k$ proofreading mechanism linearly increases the size of the assemblies (an n tile assembly becomes a $k^2 n$ tile assembly), while raising the probability of a growth error to the power k. For the complete details and the formal proofs of the error rate improvements, I refer the reader to [108].

2.6.3 Nucleation Error Correction

Chen et al. [39] proposed a family of mechanisms they called *snake proofreading* for correcting nucleation errors in temperature two systems, one mechanism for each even integer $k \geq 4$. I will first explain how the $k = 4$ mechanism works and then generalize for all k.

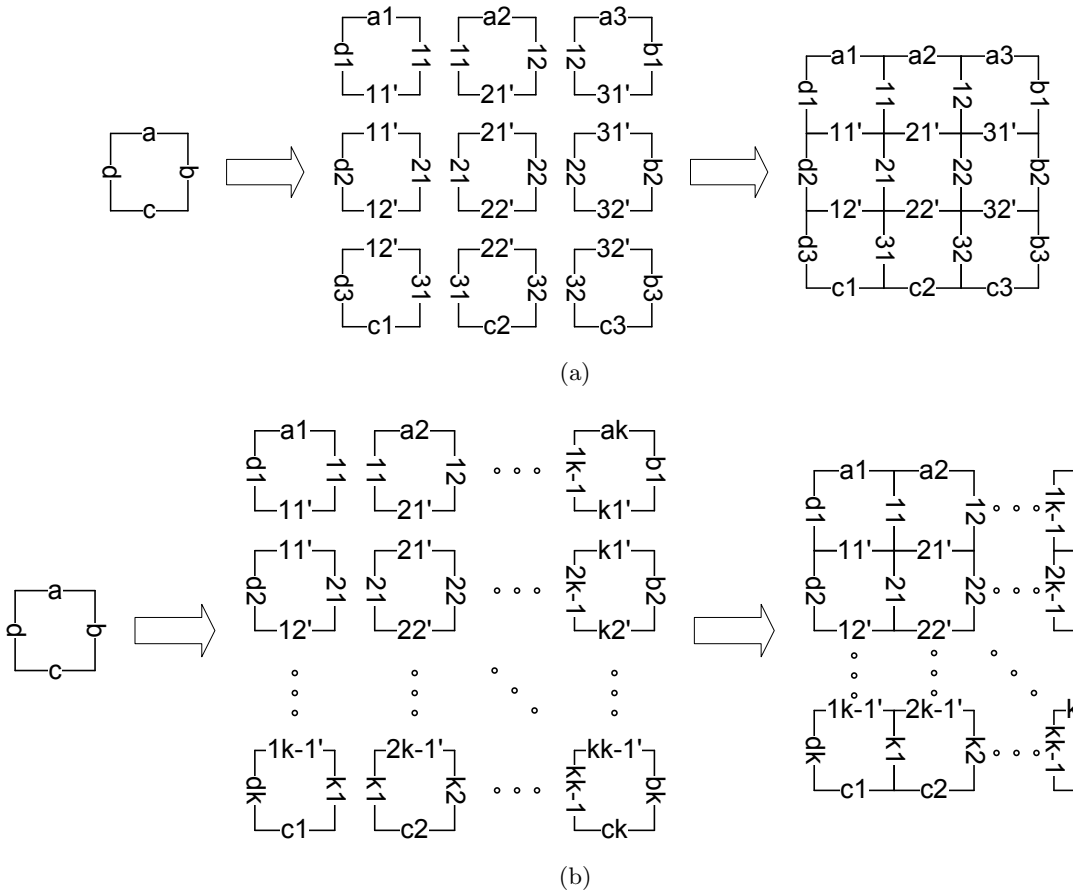


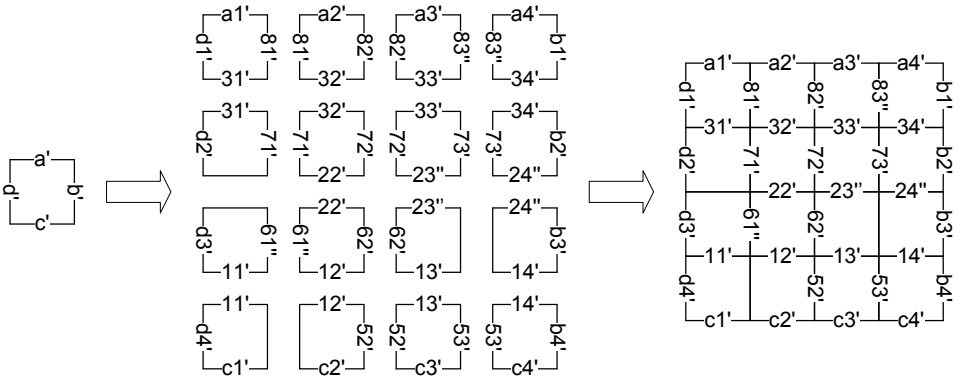
Figure 2.9: A 3×3 proofreading mechanism (a) transforms each tile type in a system into nine tile types, such that the nine can attach uniquely to each other. In general, a $k \times k$ proofreading mechanism (b) transforms each tile type in a system into k^2 tile types that can all uniquely attach to each other.

Unlike the Winfree et al. proofreading mechanism, the snake proofreading mechanism does not use a strength function that is identically 1. The snake proofreading mechanism ensures that the tiles of a $k \times k$ block attach in a specific order, which is designed to prevent nucleation errors.

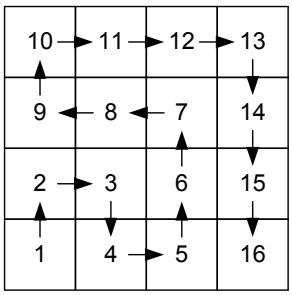
2.6.3.1 4×4 Snake Proofreading

A 4×4 snake proofreading mechanism is a transformation of every tile in the system into $4 \times 4 = 16$ tiles. The binding domains internal to the 4×4 block are unique to each tile type, and their strengths ensure a single possible order of attachment. Figure 2.10(a) shows the actual transformation of a tile with the strengths of the binding domains denoted by the number of '. Figure 2.10(b) shows the resulting order of the tile attachments.

Similarly to the proofreading mechanism, snake proofreading forces the system to either encounter two errors within a single 4×4 block, or the assembly halts until the incorrectly attached tiles detach. For the same reasons as with proofreading, at a linear cost in assembly size (and speed), the 4×4 snake proofreading mechanism squares the probability of a growth error.



(a)



(b)

Figure 2.10: A 4×4 snake proofreading mechanism transforms each tile into a 4×4 block of 16 tiles (a), ensuring that they attach in a particular order (b). In (a), the strength of each binding domain is denoted with the number of '.

2.6.3.2 $k \times k$ Snake Proofreading

In the general case, Chen et al. came up with the rules for the strengths of the labels of a $k \times k$ block of tiles that ensure a "linear assembly," such as the example in Figure 2.10(b). The $k \times k$ snake proofreading mechanism transforms a tile system with n tile types into another with k^2n tile types, linearly increases the size of the assemblies (an n tile assembly becomes a k^2n tile assembly), while raising the probability of a growth error to the power $\frac{k}{2}$. For the complete details and the formal proofs of the error rate improvements, I refer the reader to [39].

2.6.4 Compact Error Correction

The error correction mechanisms I have described so far increase the size of the final assembly and slow down the assembly time. A different class of error correction techniques, ones called *compact*, changes neither the size of the assembly nor its speed. I will describe two such mechanisms, one that works only on some special tile systems but does not increase the size of the tileset, and another that works on all temperature two systems.

2.6.4.1 Compact Snake Proofreading

Soloveichik et al. [99] observed that the snake proofreading mechanism transforms each tile type into k^2 tile types, and ensures that each one of them is placed precisely in its proper place within a $k \times k$ block. The power of the error correction comes from the tiles attaching in a linear order, and

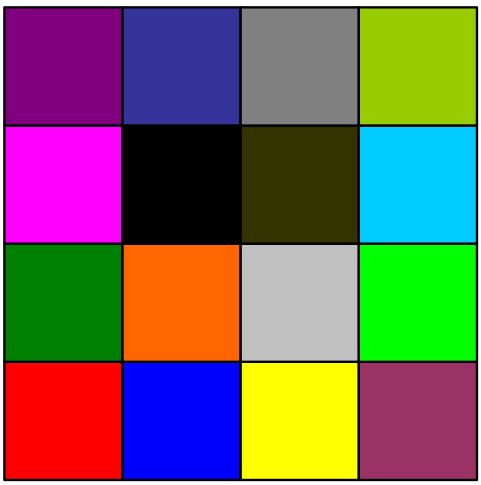


Figure 2.11: Some tile systems may require tiles to attach in such a way that each type of a tile (represented here by different colors) can only occur in a single unique position, and never in another position.

not from the increase in the number of tile types. This observation led them to explore a particular subset of tile systems, ones that limit the number of possible combinations of tiles in every $k \times k$ block to one. Suppose a system existed that required that a blue tile always attached to the east of every red tile, and a yellow tile to the east of every blue tile, and a purple tile to the east of every yellow tile, and a green tile to the north of every red tile, and so on, uniquely defining a 4×4 block of tiles, as shown in Figure 2.11. Then it is possible to change the strengths of the binding domains on the tiles to follow exactly the scheme described in Section 2.6.3.1 and Figure 2.10(b), thus achieving the benefits of the 4×4 snake proofreading mechanism without having to increase the tileset size.

A similar approach can be applied for all even k to emulate $k \times k$ proofreading mechanisms in a compact manner. Of course these mechanisms only apply to systems that have the special property of having unique $k \times k$ subassemblies within all their assemblies, thus greatly limiting the applicable systems. In particular, information rich systems, (e.g., a binary counter) do not have unique $k \times k$ subassemblies, and, furthermore, can have every possible $k \times k$ subassembly within them. Thus these mechanisms may be useful for some systems, but are not applicable in the general case. It is an interesting question as to whether one can characterize the set of functions that are computable by such systems, but it is safe to say that they are not applicable for almost all interesting functions.

2.6.4.2 Redundancy Proofreading

Reif et al. [83] proposed a technique for redundant computation within the tiles that results in compact error correction mechanisms. As before, there is a closely related mechanism for every positive integer $k \geq 2$, with k indicating the amount of redundancy introduced into the system.

For $k = 2$, every tile in the system will perform the computation of that tile as well as its immediate neighbors. Reif et al. achieve this by creating a new tile type for every pair of tile types in the original system. Thus if the original system had n distinct tile types, the 2×2 redundancy proofreading system will have n^2 tiles. Every tile, in its binding domains, encodes the tile that should attach at that binding domain and the tile that should attach to that tile. In essence, each

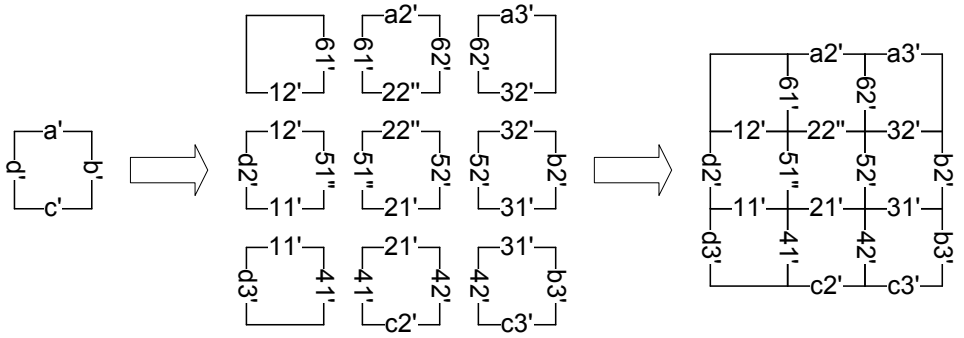


Figure 2.12: The 3×3 self-healing mechanism transforms every tile into a 3×3 block that prevents backwards (southeast) growth.

tile not only tries to ensure that its neighbor is correct, but also that that neighbor's neighbor is correct. The redundancy forces two errors to happen in close proximity in order for the final assembly to have an error, squaring the error rate (bringing an original error rate of ϵ to $6\epsilon^2$). I do not graphically demonstrate the transformation here, and refer the reader to [83] for a complete description of the 2×2 redundancy proofreading mechanism and its formal proof.

Generalizing the idea of redundancy proofreading, each k-tuple of tiles can be represented with a single tile, essentially forcing k levels or redundant checking, turning a system with n tile types into one with n^k tile types and lowering the error rate from ϵ to $\Theta\left(\epsilon^k\right)$.

2.6.5 Self-Healing

Winfree explored shotgun errors in tile assembly and found a mechanism that forces tile systems to "self-heal" such errors [107]. On one hand, shotgun errors are manageable, because the tiles that detached can reattach the exact same way they attached in the first place, thus tile systems *might* heal themselves; however, the problem is that they do not *have* to heal themselves. In other words, if the original assembly grew from the southeast to the northwest (as assemblies in temperature two systems do), a shotgun error can be repaired if tiles reattach in the error region from the southeast to the northwest. However, tiles may start attaching from the northwest to the southeast, filling in the error region incorrectly. Winfree's solution was to transform each tile type into a block of tiles that prevents backwards growth by using 0 strength binding domains. Figure 2.12 shows a 3×3 self-healing transformation.

Note that while I only showed the transformation necessary for temperature two systems, Winfree goes into some details for transforming other systems in a similar way. Winfree generalized the 3×3 mechanism to $k \times k$ mechanisms, for all odd $k \geq 3$, which can handle more and more types of tile systems, but I will not go into the details of those transformations here and refer the reader to [107] for both the formal proof of the 3×3 mechanism and the details of other mechanisms.

Chapter 3

Definitions: The Tile Assembly Model

The definitions in this chapter appear in my papers [24, 26–28].

The tile assembly model [89, 105, 106] is a formal model of crystal growth. It was designed to model self-assembly of molecules such as DNA. It is an extension of a model proposed by Wang [103]. The model was fully defined by Rothemund and Winfree [89], and the definitions here are similar to those, but I restate them here for completeness and to assist the reader.

Intuitively, the model has *tiles* or squares that stick or do not stick together based on various binding domains on their four sides. Each tile has a *binding domain* on its north, east, south, and west side, and may stick to another tile when the binding domains on the abutting sides of those tiles match and the total strength of all the binding domains on that tile exceeds the current temperature. The four binding domains define the *type* of the tile.

Formally, let Σ be a finite alphabet of binding domains such that $null \in \Sigma$. I will always assume $null \in \Sigma$ even when I do not specify so explicitly. A *tile* over a set of binding domains Σ is a 4-tuple $\langle \sigma_N, \sigma_E, \sigma_S, \sigma_W \rangle \in \Sigma^4$. A *position* is an element of $\mathbb{Z}^2$. The set of directions $D = \{N, E, S, W\}$ is the set of 4 functions from positions to positions, i.e., $\mathbb{Z}^2$ to $\mathbb{Z}^2$, such that for all positions (x, y), $N(x, y) = (x, y+1)$, $E(x, y) = (x+1, y)$, $S(x, y) = (x, y-1)$, $W(x, y) = (x-1, y)$. The positions (x, y) and (x', y') are neighbors iff $\exists d \in D$ such that $d(x, y) = (x', y')$. For a tile t, for $d \in D$, I will refer to $bd_d(t)$ as the binding domain of tile t on d's side. A special tile $empty = \langle null, null, null, null \rangle$ represents the absence of all other tiles.

A *strength function* $g\colon \Sigma \times \Sigma \to \mathbb{R}$, where g is commutative and $\forall \sigma \in \Sigma$, $g(null, \sigma) = 0$, denotes the strength of the binding domains. It is common to assume that $g(\sigma, \sigma') = 0 \iff \sigma \neq \sigma'$. This simplification of the model implies that the abutting binding domains of two tiles have to match to bind. Here, I will use $g = 1$ to mean $\forall \sigma \neq null$, $g(\sigma, \sigma) = 1$ and $\forall \sigma' \neq \sigma$, $g(\sigma, \sigma') = 0$.

Let T be a set of tiles containing the *empty* tile. A *configuration* of T is a function $A\colon \mathbb{Z} \times \mathbb{Z} \to T$. I write $(x, y) \in A$ iff $A(x, y) \neq empty$. A is finite iff there is only a finite number of distinct positions $(x, y) \in A$.

Finally, a *tile system* $\mathbb{S}$ is a triple $\langle T, g, \tau \rangle$, where T is a finite set of tiles containing *empty*, g is a strength function, and $\tau \in \mathbb{N}$ is the temperature, where $\mathbb{N} = \mathbb{Z}_{\geq 0}$.

If $\mathbb{S} = \langle T, g, \tau \rangle$ is a tile system and A is a configuration of some set of tiles $T' \subseteq \Sigma^4$ then a tile $t \in T$ can *attach* to A at position (x, y) and produce a new configuration A' iff:

- $(x, y) \notin A$, and
- $\sum_{d \in D} g(bd_d(t), bd_{d^{-1}}(A(d(x, y)))) \geq \tau$, and
- $\forall (u, v) \in \mathbb{Z}^2$, $(u, v) \neq (x, y) \Rightarrow A'(u, v) = A(u, v)$, and
- $A'(x, y) = t$.

That is, a tile can attach to a configuration only in empty positions and only if the total strength of the appropriate binding domains on the tiles in neighboring positions meets or exceeds the temperature τ. For example, if for all σ, $g(\sigma, \sigma) = 1$ and $\tau = 2$ then a tile t can attach only at positions with matching binding domains on the tiles in at least two adjacent positions.

Given a tile system $\mathbb{S} = \langle T, g, \tau \rangle$, a set of tiles Γ, and a *seed configuration* $S_0 \colon \mathbb{Z}^2 \to \Gamma$, if the above conditions are satisfied, one may attach tiles of T to S_0. Let $W_0 \subseteq \mathbb{Z}^2$ be the set of all positions where at least one tile from T can attach to S_0. For all $w \in W_0$ let U_w be the set of all tiles that can attach to S_0 at w. Let $\hat{S}_1$ be the set of all configurations S_1 such that for all positions $p \in S_0$, $S_1(p) = S_0(p)$ and for all positions $w \in W_0$, $S_1(w) \in U_w$ and for all positions $p \notin S_0 \cup W_0$, $S_1(p) = \textit{empty}$. For all $S_1 \in \hat{S}_1$, I say that $\mathbb{S}$ *produces* S_1 on S_0 in one step. If $A_0, A_1, \ldots, A_n$ are configurations such that for all $i \in \{1, 2, \ldots, n\}$, $\mathbb{S}$ produces A_i on A_{i-1} in one step, then I say that $\mathbb{S}$ produces A_n on A_0 in n steps. When the number of steps taken to produce a configuration is not important, I will simply say $\mathbb{S}$ produces a configuration A on a configuration A' if there exists $k \in \mathbb{N}$ such that $\mathbb{S}$ produces A on A' in k steps. If the only configuration produced by $\mathbb{S}$ on A is A itself, then A is said to be a *final configuration*. If there is only one final configuration A produced by $\mathbb{S}$ on S_0, then $\mathbb{S}$ is said to produce a *unique* final configuration on S_0. Finally, if A is a final configuration produced by $\mathbb{S}$ on S_0 and n is the least integer such that A is produced by $\mathbb{S}$ on S_0 in n steps, then n is the *assembly time* of $\mathbb{S}$ on S_0 to produce A.

I allow the codomain of S to be Γ, a set of tiles which may be different from T. The reason is that I will study systems that compute functions using minimal sets T; but the seed, which has to code for the input of the function, may contain more distinct tiles than there are in T. Therefore, I wish to keep the two sets separate. Note that, at any temperature τ, it takes $\Theta(n)$ distinct tiles to assemble an arbitrary n-bit input such that each tile codes for exactly one of the bits.

These definitions do not allow tiles to rotate; however, given this definition of a strength function, these systems are essentially equivalent to systems with rotating tiles.

I now describe how tile systems can compute functions.

3.1 Deterministic Computation

Intuitively, in order for a tile system to deterministically compute a function $f \colon \mathbb{N} \to \mathbb{N}$, for all $a \in \mathbb{N}$, for some seed that encodes a, the tile system should produce a unique final configuration that encodes $f(a)$. To allow configurations to encode numbers, I will designate each tile as a 1- or a 0-tile and define an ordering on the positions of the tiles. The i^{th} bit of a is 1 iff the tile in the i^{th} position of the seed configuration is a 1-tile (note that *empty* will almost always be a 0-tile). I will also require that the seed contains tiles coding for bits of a and no more than a constant number of extraneous tiles. Formally, let Γ and T be sets of tiles. Let Δ be the set of all possible seed configurations $S \colon \mathbb{Z}^2 \to \Gamma$, and let Ξ be the set of all possible finite configurations $C \colon \mathbb{Z}^2 \to \Gamma \cup T$. Let $v \colon \Gamma \cup T \to \{0, 1\}$ code each tile as a 1- or a 0-tile and let $o_s, o_f \colon \mathbb{N} \to \mathbb{Z}^2$ be two injections. Let the seed encoding function $e_s \colon \Delta \to \mathbb{N}$ map a seed S to a number such that $e_s(S) = \sum_{i=0}^{\infty} 2^i v(S(o_s(i)))$ if and only if for no more than a constant number of (x, y) not in the image of o_s, $(x, y) \in S$. Let the answer encoding function $e_f \colon \Xi \to \mathbb{N}$ map a configuration F to a number such that $e_f(F) = \sum_{i=0}^{\infty} 2^i v(F(o_f(i)))$.

Let $\mathbb{S} = \langle T, g, \tau \rangle$ be a tile system. I say that $\mathbb{S}$ computes a function $f \colon \mathbb{N} \to \mathbb{N}$ iff for all $a \in \mathbb{N}$ there exists a seed configuration S such that $\mathbb{S}$ produces a unique final configuration F on S and $e_s(S) = a$ and $e_f(F) = f(a)$.

I generalize the above definition for a larger set of functions. Let $\hat{m}$, $\hat{n} \in \mathbb{N}$ and $f \colon \mathbb{N}^{\hat{m}} \to \mathbb{N}^{\hat{n}}$ be a function. For all $0 \leq m < \hat{m}$ and $0 \leq n < \hat{n}$, let o_{s_m}, $o_{f_n} \colon \mathbb{N} \to \mathbb{Z}^2$ be injections.

Let the seed encoding functions $e_{s_m}\colon \Delta \to \mathbb{N}$ map a seed S to $\hat{m}$ numbers such that $e_{s_m}(S) = \sum_{i=0}^{\infty} 2^i v(S(o_{s_m}(i)))$ iff for no more than a constant number of (x, y) not in the union of the images of all o_{s_m}, $(x, y) \in S$. Let the answer encoding functions $e_{f_n}\colon \Xi \to \mathbb{N}$ each map a configuration F to a number such that $e_{f_n}(F) = \sum_{i=0}^{\infty} 2^i v(F(o_{f_n}(i)))$. Then I say that $\mathbb{S}$ computes the function f iff for all $\vec{a} \in \mathbb{N}^{\hat{m}} = \langle a_0, a_1, \cdots, a_{\hat{m}-1} \rangle$, there exists a seed configuration S such that $\mathbb{S}$ produces a unique final configuration F on S and $e_{s_m}(S) = a_m$ and $\langle e_{f_0}(F), e_{f_1}(F), \cdots, e_{f_{\hat{n}-1}} \rangle = f(\vec{a})$.

3.2 Nondeterministic Computation

In examining deterministic computation, in order for a system to compute a function, I required that the system produced unique final configurations. In some implementations of the tile assembly model systems, many assemblies happen in parallel. In fact, it is often almost impossible to create only a single assembly, and thus there is a parallelism that my previous definitions did not take advantage of. Here, I define the notion of nondeterministic computation in the tile assembly model.

I have defined a tile system to produce a unique final configuration on a seed if for all sequences of tile attachments, all possible final configurations are identical. If different sequences of tile attachments attach different tiles in the same position, the system is said to be *nondeterministic*. Intuitively, a system nondeterministically computes a function iff at least one of the possible sequences of tile attachments produces a final configuration that codes for the solution.

Since a nondeterministic computation may have unsuccessful sequences of attachments, it is important to distinguish those from the successful ones. Further, in many implementations of the tile assembly model that would simulate all the nondeterministic executions at once, it is useful to be able to identify which executions succeeded and which failed. For some problems, only an exponentially small fraction of the assemblies would represent a solution, and finding such an assembly would be difficult. For example, a DNA-based crystal-growing system would create millions of crystals, and only a few of them may represent the correct answer, while all others represent failed computations. Finding a successful computation by sampling the crystals at random would require time exponential in the input size. Thus it would be useful to attach a special identifier tile to the crystals that succeed so that the crystals may be filtered to find the solution quickly. It may also be possible to attach the special identifier tile to solid support so that the crystals representing successful computations may be extracted from the solution. I thus specify one of the tiles of a system as an *identifier* tile that only attaches to a configuration that represents a successful sequence of attachments.

Let $\hat{m}$, $\hat{n} \in \mathbb{N}$ and let $f\colon \mathbb{N}^{\hat{m}} \to \mathbb{N}^{\hat{n}}$ be a function. For all $0 \leq m < \hat{m}$ and $0 \leq n < \hat{n}$, let o_{s_m}, $o_{f_n}\colon \mathbb{N} \to \mathbb{Z}^2$ be injections. Let the seed encoding functions $e_{s_m}\colon \Delta \to \mathbb{N}$ map a seed S to $\hat{m}$ numbers such that $e_{s_m}(S) = \sum_{i=0}^{\infty} 2^i v(S(o_{s_m}(i)))$ iff for no more than a constant number of (x, y) not in the union of the images of all o_{s_m}, $(x, y) \in S$. Let the answer encoding functions $e_{f_n}\colon \Xi \to \mathbb{N}$ each map a configuration F to a number such that $e_{f_n}(F) = \sum_{i=0}^{\infty} 2^i v(F(o_{f_n}(i)))$. Let $\mathbb{S}$ be a tile system with T as its set of tiles, and let $r \in T$. Then I say that $\mathbb{S}$ *nondeterministically computes* a function f with identifier tile r iff for all $\vec{a} \in \mathbb{N}^{\hat{m}} = \langle a_0, a_1, \cdots, a_{\hat{m}-1} \rangle$ there exists a seed configuration S such that for all final configurations F that $\mathbb{S}$ produces on S, $r \in F(\mathbb{Z}^2)$ iff $\forall 0 \leq m < \hat{m}$, $e_{s_m}(S) = a_m$ and $\langle e_{f_0}(F), e_{f_1}(F), \cdots, e_{f_{\hat{n}-1}}(F) \rangle = f(\vec{a})$.

I was careful to pick the seed and answer encoding functions before the tile system because if you are allowed to tailor the encoding functions to specific systems, you can easily solve the halting problem by encoding all the complexity of the halting problem in the answer encoding function (design a system that on input n attaches a single 0- and a single 1-tile to the seed and the answer encoding function maps 0 to 1 if the n^{th} Turing machine halts on input n and to 0 otherwise). While

this formalism does not allow it, sometimes it may be convenient to give yourself some freedom in defining the answer encoding function and let it depend on certain aspects of the assembly process, but one has to be extremely careful stepping on that slippery slope. In Section 4.2, I will step on exactly that slope, but I will argue carefully that the system I present is equivalent to a slightly more complex one that fits the definitions exactly as presented here.

I have defined what it means for a tile system to compute a function f, whether it is deterministically or nondeterministically; however, I have not mentioned what happens if f is not defined on some arguments. Intuitively, the deterministic system should not produce a unique final configuration, and no nondeterministic assembly should contain the identifier tile. The above definitions formalize this intuitive idea.

3.3 Deciding Sets

Often, computer scientists talk about deciding subsets of the natural numbers instead of computing functions. Deciding a subset of the natural numbers is synonymous with computing a function that has value 1 on arguments that are in the set, and value 0 on arguments that are not in the set. I adapt the definition of nondeterministically computing functions to nondeterministically deciding subsets of natural numbers. (There is also a direct analog of deciding sets deterministically, which I do not bother to formally specify here.) Let $\mathbb{N} = \mathbb{Z}_{\geq 0}$. Since for all constants $n \in \mathbb{N}$, the cardinalities of $\mathbb{N}^n$ and $\mathbb{N}$ are the same, one can encode an element of $\mathbb{N}^n$ as an element of $\mathbb{N}$. Thus it makes sense to talk about deciding subsets of $\mathbb{N}^n$. The below-defined functions o_{s_m} can depend on the mapping of $\mathbb{N}^n \to \mathbb{N}$.

Let $v\colon \Gamma \cup T \to \{0,1\}$ code each tile as a 1- or a 0-tile. Let $\hat{m} \in \mathbb{N}$ and let $\Omega \subseteq \mathbb{N}^{\hat{m}}$. For all $0 \leq m < \hat{m}$, let $o_{s_m}\colon \mathbb{N} \to \mathbb{Z}^2$ be injections. Let the seed encoding functions $e_{s_m}\colon \Delta \to \mathbb{N}$ map a seed S to $\hat{m}$ numbers such that $e_{s_m}(S) = \sum_{i=0}^{\infty} 2^i v(S(o_{s_m}(i)))$ iff for no more than a constant number of (x,y) not in the union of the images of all o_{s_m}, $(x,y) \in S$. Let $\mathbb{S}$ be a tile system with T as its set of tiles, and let $r \in T$. Then I say that $\mathbb{S}$ *nondeterministically decides* a set Ω with identifier tile r iff for all $\vec{a} = \langle a_0, a_1, \cdots, a_{\hat{m}-1}\rangle \in \mathbb{N}^{\hat{m}}$ there exists a seed configuration S such that for all final configurations F that $\mathbb{S}$ produces on S, $r \in F(\mathbb{Z}^2)$ iff $\forall 0 \leq m < \hat{m}$, $e_{s_m}(S) = a_m$ and $\vec{a} \in \Omega$.

Chapter 4

Computational Tile Systems

In this chapter, I will examine tile assembly model systems that compute functions. Section 4.1 will present systems that add and multiply. Section 4.2 will present systems that factor positive integers. Sections 4.3 and 4.4 will present systems that solve well-known NP-complete problems: *SubsetSum* in Section 4.3 and *SAT* in Section 4.4. The work presented in this chapter appears in my papers [24, 26–28].

4.1 Arithmetic with Tiles: Addition and Multiplication

In this section, I will examine adders, systems that compute $f(a, b) = a+b$; and multipliers, systems that compute $f(a, b) = ab$. These systems appear in my paper [24].

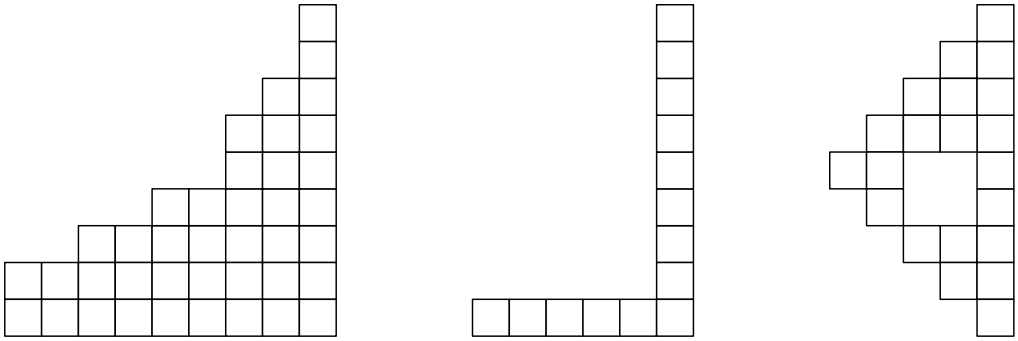
I require adder and multiplier systems to encode their inputs in binary. I could ask each input number to have a unique tile ($|\Gamma| = \Theta(2^n)$, for n-bit inputs) and a unique tile for every possible pair of inputs ($|T| = \Theta(2^n)$). Such a system would compute in $\Theta(1)$ time. However, in implementing such a system one would need exponentially many different types of tiles (e.g., molecules), which would make the implementation intractable. I choose, as is common in computer science, to look at systems that limit the number of components ($|\Gamma| + |T|$) to be polynomial in the size of the input. In most of the constructions presented here (all but one), $|\Gamma| = \Theta(1)$. I will present systems with small constants and show their running times are linear in the input size.

4.1.1 Temperature Two Systems and Tile System Configurations

In this section, I will prove three lemmas about temperature two systems that will later be useful in studying such systems.

Let A be a configuration. A is a *step-configuration* iff $\exists y_{min}, y_{max}, x_{max} \in \mathbb{Z}$ such that all of the following are true:

- $\forall x, y \in \mathbb{Z}$ such that $y < y_{min}$ or $y > y_{max}$, $(x, y) \notin A$, and
- $\forall x, y \in \mathbb{Z}$ such that $x > x_{max}$, $(x, y) \notin A$, and
- $\forall x, y \in \mathbb{Z}$ $\exists k_y \in \mathbb{N}$ such that:
  - $k_y \leq k_{y-1}$, and
  - $x \leq x_{max} - k_y \Rightarrow (x, y) \notin A$, and
  - $x_{max} - k_y < x \leq x_{max} \Rightarrow (x, y) \in A$.



(a) (b) (c)

Figure 4.1: A step-configuration (a), an L-configuration (b), and a configuration that is neither a step-configuration nor an L-configuration (c). A step-configuration has no holes and consists of a finite number of consecutive rows, such that each row has the same rightmost position and is shorter than the row below it. An L-configuration is a special case of a step-configuration and looks like the mirror image of the letter L.

Intuitively, a step-configuration has no holes and consists of a finite number of consecutive rows such that each row has the same rightmost position and is no longer than the row below it. A step-configuration can be desribed as a "solid hill." Figures 4.1(a) and 4.1(b) show examples of step-configurations and Figure 4.1(c) shows an example of a configuration that is *not* a step-configuration.

A *temperature two system* is a tile assembly model system $\mathbb{S} = \langle T, g, \tau \rangle$ such that $g = 1$, $\tau = 2$, and for all $t, t' \in T$, $\langle bd_E(t), bd_S(t) \rangle = \langle bd_E(t'), bd_S(t') \rangle \iff t = t'$. That is, a tile of a unique type may attach to the northwest of an assembly. In general, I will be interested in studying temperature two systems' behavior on step-configurations.

Let A be a configuration. A is an *L-configuration* iff $\exists x_{min}, x_{max}, y_{min}, y_{max} \in \mathbb{Z}$ such that $\forall x, y \in \mathbb{Z}$, $(x, y) \in A$ iff either

- $y_{min} \leq y \leq y_{max}$ and $x = x_{max}$, or
- $x_{min} \leq x \leq x_{max}$ and $y = y_{min}$.

Intuitively, an L-configuration looks like a mirror image of the letter L. Clearly, an L-configuration is a step-configuration. Figure 4.1(b) shows an example of an L-configuration.

Lemma 4.1.1 (Step configuration lemma) *Let $\mathbb{S} = \langle T, g, \tau \rangle$ be a tile system such that $g = 1$ and $\tau = 2$. Let Γ be a set of tiles and let $S \colon \mathbb{Z}^2 \to \Gamma$ be a seed step-configuration. All configurations produced by $\mathbb{S}$ on S are step-configurations.*

Proof: (By induction over tile addition.) The base case is trivial because the seed configuration is assumed to be a step-configuration. Now, assume that you have a step-configuration A and add the next tile to produce A'. Since all the binding domains are of strength 1, and $\tau = 2$, a tile may only attach if it is adjacent to τ or more tiles in A. Thus possible places to attach are "corners" of A, or places at the end of a non-empty row that have a longer row below it. That is, a tile may attach at (x, y) iff $(x, y) \notin A$ and $(x + 1, y) \in A$ and $(x, y - 1) \in A$. Therefore, $k_{y-1} > k_y$ and since only one tile is added, after the addition $k_{y-1} \geq k_y$. Therefore A' is a step-configuration. ■

Lemma 4.1.2 (Unique final configuration lemma) *Let $\mathbb{S} = \langle T, g, \tau \rangle$ be a tile system such that $g = 1$ and $\tau = 2$. Let Γ be a set of tiles and let $S \colon \mathbb{Z}^2 \to \Gamma$ be a seed step-configuration. If $\forall t \in T$ the pair $(bd_s(t), bd_e(t))$ is unique then $\mathbb{S}$ produces a unique final configuration on S.*

Proof: First of all, because $\tau = 2$ and $g = 1$, a tile may only attach if it has two neighbors, thus the configuration will never grow larger than the bounding box of S, so there will be a final configuration. Second, by Lemma 4.1.1, whenever a tile attaches, it will attach to a step-configuration, and thus the only positions at which it can attach will have the south and east neighbors available for binding. Since $\tau = 2$ both of them have to be available, and since the pair $(bd_S(t), bd_E(t))$ is unique for each tile t, only one type of tile may attach at every given (x, y) position. Therefore, the final configuration is unique. ■

Corollary 4.1.3 (Unique final configuration corollary) *Let $\mathbb{S} = \langle T, g, \tau \rangle$ be a tile system such that $g = 1$ and $\tau = 2$. Let Γ be a set of tiles and let $S \colon \mathbb{Z}^2 \to \Gamma$ be a seed L-configuration but with the corner southeast tile missing and let the binding domains neighboring that corner be null. Then if $\forall t \in T$, the pair $(bd_s(t), bd_e(t))$ is unique then $\mathbb{S}$ produces a unique final configuration on S.*

Proof: Since the binding domains neighboring the southeast corner are *null*, no tile can attach there. By Lemma 4.1.2, the rest of the seed configuration produces a unique final configuration. Thus $\mathbb{S}$ produces a unique final configuration on S. ■

Lemma 4.1.4 (Assembly time lemma) *Let $\mathbb{S} = \langle T, g, \tau \rangle$ be a tile system such that $g = 1$ and $\tau = 2$. Let Γ be a set of tiles, let $S \colon \mathbb{Z}^2 \to \Gamma$ be a seed L-configuration, and let x_{max} and y_{max} be the lengths of the bottom row and rightmost column of S, respectively. If $\forall t \in T$ the pair $(bd_s(t), bd_e(t))$ is unique then $\mathbb{S}$ produces a unique final configuration on S. If that final configuration is the complete $x_{max} \times y_{max}$ rectangle, then the assembly time is $\Theta(x_{max} + y_{max})$.*

Proof: $\mathbb{S}$ produces a unique final configuration on S by Lemma 4.1.2. Let that configuration be F. Assume F is the complete x_{max} by y_{max} rectangle. Without loss of generality, assume $x_{max} \geq y_{max}$. It is clear that, at first, only a single tile may attach to S, in the corner position. After that tile attaches, there are two new corner positions that may have a tile attach. Similarly, for the first y_{max} time steps, during the i^{th} time step exactly i tiles may attach. After that, for the next $x_{max} - y_{max}$ time steps, exactly y_{max} tiles may attach per step. Finally, for the last y_{max} steps, on the j^{th} time step, exactly $y_{max} - j$ tiles may attach. The resulting configuration is the $x_{max} \times y_{max}$ rectangle, and the assembly time is $y_{max} + x_{max}$ steps. ■

Corollary 4.1.5 (Assembly time corollary) *Let $\mathbb{S} = \langle T, g, \tau \rangle$ be a tile system such that $g = 1$ and $\tau = 2$. Let Γ be a set of tiles, let $S \colon \mathbb{Z}^2 \to \Gamma$ be a seed L-configuration but with a single tile missing: the corner tile and let the binding domains neighboring that corner be null, and let x_{max} and y_{max} be the lengths of the bottom row and rightmost column of S. If $\forall t \in T$ the pair $(bd_s(t), bd_e(t))$ is unique, then $\mathbb{S}$ produces a unique final configuration on S. If that final configuration is the (almost) complete $x_{max} \times y_{max}$ rectangle (missing only the same single corner as S), then the assembly time is $\Theta(x_{max} + y_{max})$.*

Proof: By Corollary 4.1.3, $\mathbb{S}$ produces a unique final configuration on S. By Lemma 4.1.4, the assembly time is $\Theta(x_{max} + y_{max})$. ■

Let $a = \sum_i 2^i a_i$ where $a_i \in \{0, 1\}$. I will often say $bit_i(a)$ or the i^{th} bit of a to mean a_i, so the indexing of the bits starts at 0. I will also say $cbit_i(a, b)$ or the i^{th} carry bit of $a + b$ to mean 1 iff the sum of a_{i-1}, b_{i-1}, and $cbit_{i-1}(a, b)$ is greater or equal to 2. Formally, $cbit_0(a, b) = 0$ and $cbit_i(a, b) = bit_1(a_{i-1} + b_{i-1} + cbit_{i-1}(a, b))$.

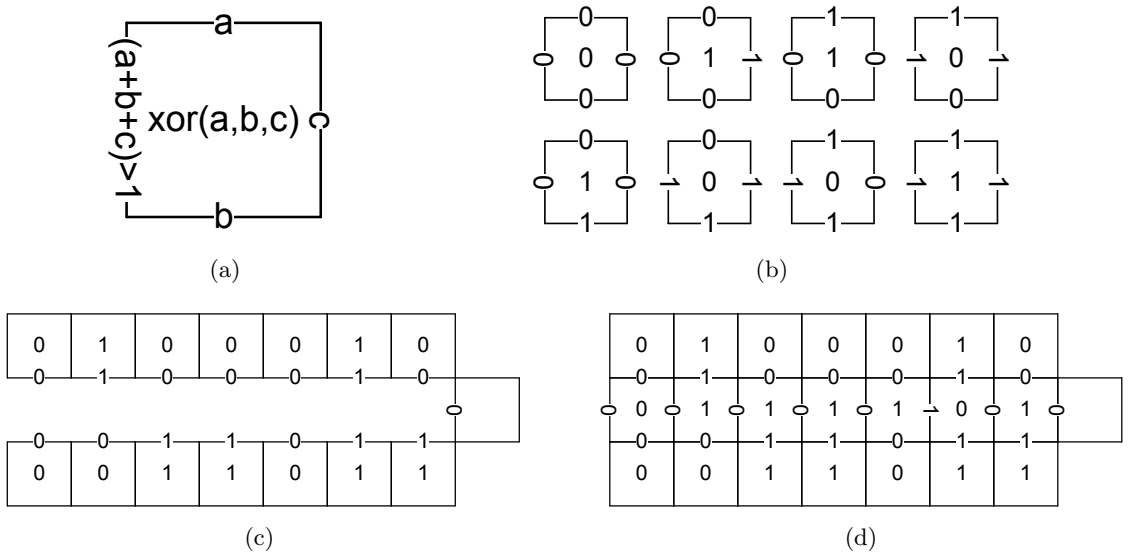


Figure 4.2: Tile system $\mathbb{S}_{+8}$ computes the function $f(a, b) = a + b$. The tiles have 3 input sides (east, north, and south) and 1 output side (west) (a). Each tile is labeled with a 1 or a 0 to assist the reader with reading the encoding v_8. There are eight tiles in the system (b). Given a sample input of $b = 100010_2 = 34$ and $a = 11011_2 = 27$ (c), with 34 on the top row and 27 on the bottom row, the system fills the row in the middle with $a + b = 111101_2 = 61$ to produce the unique final configuration (d). Note that the least significant digit is on the right. Also note that the inputs are padded with extra 0 tiles in the most significant bit places because the sum of two n-bit numbers may be as large as $n + 1$ bits.

4.1.2 Adder Tile Systems

In this section, I will first discuss a small tile system that computes $f(x, y) = x + y$ using $\Theta(1)$ distinct tiles, I will second show a system that uses $\Theta(n)$ tiles but presents important ideas, and finally, I will introduce a new concept to simplify that system to use only $\Theta(1)$ tiles.

For some tile systems and some seeds, tiles will attach only when certain neighbors are present. I call the sides of the tile that attaches to these neighbors the "input" sides. I call the other sides the "output" sides. For example, given a tile system $\langle T, 1, 2\rangle$ and a step seed configuration, tiles will only attach to a configuration at a position if the east and south neighbors of that position are in the configuration. The east and south sides of these tiles are the input sides and the west and north sides are the output sides. The input sides determine which tile attaches to a configuration now, and the output sides determine which tiles may attach later.

4.1.2.1 Eight-Tile Adder

Theorem 4.1.6 *Let $\Sigma_8 = \{0, 1\}$, $g_8 = 1$, $\tau_8 = 3$, and T_8 be a set of tiles over Σ_8 as described in Figure 4.2(b). Then $\mathbb{S}_{+8} = \langle T_8, g_8, \tau_8\rangle$ computes the function $f(a, b) = a + b$.*

Intuitively, $\mathbb{S}_{+8}$ has eight tiles with the east, north, and south sides as the input sides and the west side as the output side, outputting 1 iff the sum of the inputs is at least 2. Note that I do not formally define the tiles of T because it is easiest to read their descriptions from Figure 4.2(b).

To encode the answer, the type of the tile is determined by the sum of the inputs, modulo 2. Figure 4.2(a) shows the concepts behind the tiles and Figure 4.2(b) shows the eight tiles for all

possible 0 and 1 binding domains for the three input sides. The 1 or 0 in the middle of the tile t is that tile's $v_8(t)$ value.

Figure 4.2(c) shows a sample seed configuration, which encodes two numbers in binary: $100010_2 = 34$, $11011_2 = 27$. The number 34 is encoded in the top row and the number 27 is encoded in the bottom row. There are 5 tiles in Γ_8, the 1 and 0 tiles for each of the two inputs, and the single starter tile on the right side. Note that at the start, only one tile may attach to this configuration because $\tau_8 = 3$. Figure 4.2(d) shows the final configuration for the example of $34 + 27$, with the solution encoded in the center row. The row reads 111101_2, which is $61 = 34 + 27$.

Because the sum of two n-bit numbers may be as large as $n + 1$ bits, each of the two inputs needs to be padded to be $n + 1$ bits long with extra 0 tiles.

Proof: (Theorem 4.1.6) Consider the tile system $\mathbb{S}_{+8}$. Let a and b be the numbers to add and let n_a and n_b be the sizes, in bits, of a and b, respectively. Let $n = \max(n_a, n_b)$. For all $i \in \mathbb{N}$, let $a_i, b_i \in \{0, 1\}$ be such that $a = \sum_i 2^i a_i$ and $b = \sum_i 2^i b_i$.

Let $\Gamma_8 = \{\alpha_0 = \langle 0, \mathit{null}, \mathit{null}, \mathit{null}\rangle, \alpha_1 = \langle 1, \mathit{null}, \mathit{null}, \mathit{null}\rangle, \beta_0 = \langle \mathit{null}, \mathit{null}, 0, \mathit{null}\rangle, \beta_1 = \langle \mathit{null}, \mathit{null}, 1, \mathit{null}\rangle, \gamma = \langle \mathit{null}, \mathit{null}, \mathit{null}, 0\rangle\}$. Let the seed configuration $S\colon \mathbb{Z}^2 \to \Gamma_8$ be such that

- $S(1, 0) = \gamma$,
- $\forall i = 0, \ldots, n$, $S(-i, -1) = \alpha_{a_i}$,
- $\forall i = 0, \ldots, n$, $S(-i, 1) = \beta_{b_i}$, and
- for all other $(x, y) \in \mathbb{Z}^2$, $S(x, y) = \mathit{empty}$.

It is clear that there is only a single position where a tile may attach to S. It is also clear that after that tile attaches there will only be a single position where a tile may attach. By induction, because $\forall t \in T_8$ the triplet $\langle bd_S(t), bd_E(t), bd_N(t)\rangle$ is unique, and because $\tau_8 = 3$, it follows that $\mathbb{S}$ produces a unique final configuration on S. Let that configuration be F.

For all $0 \leq i \leq n$, S and F agree on $(-i, -1)$ and $(-i, 1)$. Therefore, $bd_N(F(-i, -1)) = a_i$ and $bd_S(F(-i, 1)) = b_i$. I will now show, by induction, that $v(F(-i, 0)) = bit_i(a + b)$ and $bd_W(F(-i, 0)) = cbit_{i+1}(a, b)$.

From the definition of S, $bd_W(F(1, 0))$ is $0 = cbit_0(a, b)$. Now I assume that $bd_W(F(-(i - 1), 0)) = cbit_i(a, b)$ and show that $bd_W(F(-i, 0)) = cbit_{i+1}(a, b)$ and $v(F(-i, 0)) = bit_i(a + b)$. Let $t = F(-i, 0)$. The value $v(t)$ is $xor(bd_N(t), bd_E(t), bd_S(t))$. Since t binds with strength 3, $bd_N(t) = bd_S(F(-i, 1))$, $bd_E(t) = bd_W(F(-(i - 1), 0))$, $bd_S(t) = bd_N(F(-i, -1))$, so $v(t)$ is the xor of a_i, b_i, and $cbit_i(a, b)$. The i^{th} bit of $a + b$ is exactly that xor. The binding domain $bd_W(t)$ is defined as 1 if at least two of a_i, b_i, and $cbit_i(a, b)$ are 1 and 0 otherwise. That is exactly the definition of $cbit_{i+1}(a, b)$.

Thus, for all $1 \leq i \leq n$ $v(F(-i, 0)) = bit_i(a + b)$. It is clear that those are the only tiles of T in F. For all $j \in \mathbb{N}$, let $o_{s_1}(j) = (-j, -1)$, $o_{s_2}(j) = (-j, 1)$, and $o_f(j) = (-j, 0)$. Because $a + b$ is no longer then $n + 1$ bits, it follows that $\mathbb{S}_{+8}$ computes $f(a, b) = a + b$. ■

The logic of the system is identical to a series of 1-bit full adders. Each solution tile takes in a bit from each of the inputs on the north and south sides and a carry bit from the previous solution tile on the east side, and outputs the next carry bit on the west side. Because $\tau_8 = 3$, only a tile with three neighbors may attach at any time, and therefore, no tile may attach until its right neighbor has. Thus the assembly time for this system is n steps to add two n-bit numbers. Note that $|\Gamma_8| = 5$ and $|T_8| = 8$.

Note that if for some i, $a_i = b_i = 0$, then $cbit_{i+1} = 0$ regardless of the value of $cbit_i$. Similarly, if $a_i = b_i = 1$, then $cbit_{i+1} = 1$ regardless of the value of $cbit_i$. Thus, in theory, computation could

start in parallel at all such i, and it seems feasible to design a tile system to take advantage of this property to greatly speed up the assembly time for most inputs.

4.1.2.2 L-Configuration Adder

I now present an adder tile system that uses $\Theta(n)$ tile types to compute, but builds on L-shape seed configurations. This adder will use the 2 sides of the L-configuration to encode inputs, and produce its output on the top row of an almost complete rectangle. Therefore, systems could chain computations together, using the output of this computation as an input to another computation.

Theorem 4.1.7 *For all $n \in \mathbb{N}$, let $\Sigma_n = \{0,1\} \cup \{\#i | i = 0,1,\ldots,n\}$, $g_n = 1$, $\tau_n = 2$, and T_n be a set of tiles over Σ_n as described in Figure 4.3(b). Then $\mathbb{S}_{+n} = \langle T_n, g_n, \tau_n \rangle$ computes the function $f(a,b) = a + b$ for all $a, b < 2^n$.*

Let n_a and n_b be the sizes, in bits, of a and b, respectively, and let $n = \max(n_a, n_b)$. Intuitively, $\mathbb{S}_{+n}$ adds a and b using $\Theta(n)$ distinct tile types. The first input number, a, is coded on the bottom row and the second input number, b, on the rightmost column of an L-configuration. The adder adds one bit of b to a in each row. The i^{th} bit has to be added in the i^{th} column, and the system uses the $\Theta(n)$ tiles to count down to the correct position. Each tile has two input sides (south and east) and two output sides (north and west). The north side is the value of the tile and the west side is the carry bit. The tile descriptions are easiest to read from Figure 4.3(b).

To encode the answer, the type of the tile is determined by the sum of the inputs, modulo 2. Figure 4.3(a) shows the concepts behind the tiles, and Figure 4.3(b) shows the actual tiles (but only some of the counting tiles) for $n = 99$. The 1 or 0 in the middle of the tile t is that tile's $v(t)$ value.

Figure 4.3(c) shows a sample seed configuration, which encodes two numbers in binary: $1010111_2 = 87$, $101101_2 = 45$. The number 87 is encoded in the bottom row and the number 45 is encoded in the column. There are up to $n+3$ tiles in Γ_n, the 1 and 0 tiles for the row input, the 0 tile for the column input, and up to n distinct tiles for the 1-bits of the column input. Note that at the start, only one tile may attach to this configuration because $\tau_n = 2$ and there is only a single corner. Since no two tile types in T_{+n} have identical south-east binding domain pairs, by Corollary 4.1.3, $\mathbb{S}_{+n}$ produces a unique final configuration. Figure 4.3(d) shows the final configuration for the example of $87+45$, with the solution encoded in the top row. The row reads 10000100_2, which is $132 = 87+35$.

Again, because the sum of two n-bit numbers may be as large as $n+1$ bits, the row input needs to be padded with a single extra 0 tile as its most significant bit. The column input does not need this padding.

Each row adds a single bit of the column input at the correct location. Theorem 4.1.7 follows from the logic of the tiles (I do not bother to formally prove it here because the purpose of this system is to display ideas useful in designing later systems). By Corollary 4.1.5, the assembly time for adding two n-bit numbers is $\Theta(n)$ steps. Note that $|\Gamma_n| = n + 3$ and $|T_n| = 2n + 6$.

4.1.2.3 L-Configuration Constant-Size Tileset Adder

I now modify the adder that acts on L-configurations to only use $\Theta(1)$ tiles to compute by invoking an idea of sandwiching tiles, which will be important in constructing a multiplier system. As before, this adder will produce its output on the top row of a rectangle and allow the chaining of computations.

Theorem 4.1.8 *Let $\Sigma_{16} = \{0, 1, \#0, \#1, {}^\star 0, {}^\star 1\}$, $g_{16} = 1$, $\tau_{16} = 2$, and T_{16} be a set of tiles over Σ_{16} as described in Figure 4.4(b). Then $\mathbb{S}_{+16} = \langle T_{16}, g_{16}, \tau_{16} \rangle$ computes the function $f(a,b) = a+b$.*

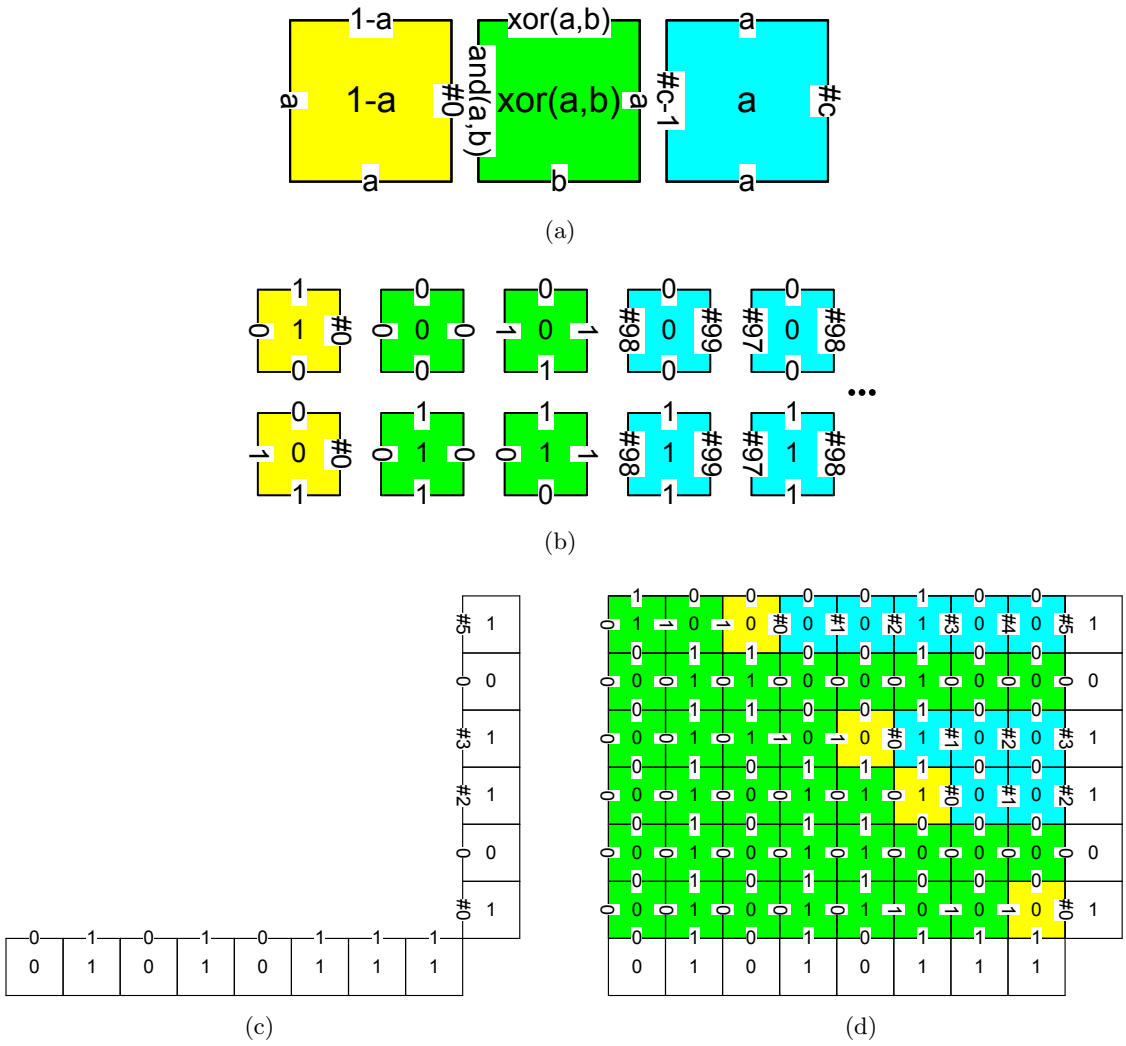


Figure 4.3: Tile system $\mathbb{S}_{+n}$ computes the function $f(a,b) = a+b$ and uses $\Theta(n)$ distinct tile types. The tiles have 2 input sides (east and south) and 2 output sides (west and north). The north side is the value of the bit, and the west side is the carry bit (a). Each tile is labeled with a 1 or a 0 to assist the reader with reading the encoding v. There are $2n+6$ tiles in the system (b). Given a sample input of $a = 1010111_2 = 87$, and $b = 101101_2 = 45$ (c), with 87 on bottom row and 45 on the rightmost column, the system fills the rectangle in the middle by adding a single bit of b in each row. The top row reads the solution: $a+b = 10000100_2 = 132$. Note that the least significant digit is on the right and that the color is purely to help the reader track tiles; the tiles themselves have no sense of color. Also note that the row input a has an extra 0 tile in the most significant bit place because the sum of two n-bit numbers may be as large as $n+1$ bits.

Intuitively, $\mathbb{S}_{+16}$ adds two n-bit numbers using 16 tiles. While the previous adder used i tiles to count down to the i^{th} position, this tile system will use a constant number of tiles to find that position. It is easy to imagine how to create a tile system that finds the i^{th} position in the i^{th} row (or just builds a diagonal line). The key to $\mathbb{S}_{+16}$ is to make one system compute both, the diagonal line and the sum, by using a technique related to tile sandwiching. A sandwich tile [8] is a tile which is essentially made up of two tiles. Suppose a set of tiles A over Σ_A constructs a diagonal line and a set of tiles B over Σ_B ensures that red tiles only attach to blue tiles on the east-west sides. Then one can create a set of sandwich tiles C over $\Sigma_A \times \Sigma_B$ which will make striped diagonal lines. C is defined such that if $a = \langle \sigma_{a,N}, \sigma_{a,E}, \sigma_{a,S}, \sigma_{a,W} \rangle \in A$ and $b = \langle \sigma_{b,N}, \sigma_{b,E}, \sigma_{b,S}, \sigma_{b,W} \rangle \in B$, then the sandwich tile of a and b is $c = \langle (\sigma_{a,N}, \sigma_{b,N}), (\sigma_{a,E}, \sigma_{b,E}), (\sigma_{a,S}, \sigma_{b,S}), (\sigma_{a,W}, \sigma_{b,W}) \rangle \in C \subseteq (\Sigma_A \times \Sigma_B)^4$. In other words, if c is the sandwich of two tiles a and b, and c' is the sandwich tile of a' and b', then c binds to c' iff a binds to a', and b binds to b'. Note that $|C| = |A| \cdot |B|$.

$\mathbb{S}_{+16}$ sandwiches tile types that build a diagonal line with the tile types from $\mathbb{S}_{+n}$. First of all, note that a system with 2 tile types can form an infinite diagonal line. Though I do not formally describe such a system, I will call the set of 2 tiles that build a diagonal line $T_{diagonal}$. (The reader may examine the yellow and magenta tiles in Figure 4.4, which under the right conditions would build a diagonal line, although they do more than just that.)

Remember that $\mathbb{S}_{+n}$ had three different kinds of tiles: yellow tiles that added the i^{th} bit to the running sum, green tiles that added the number below to a possible carry bit, and blue tiles that counted down to the i^{th} position and also propagated information up. Suppose I change this set of tiles so that the blue tiles no longer count down to the i^{th} position, but the tile in that position is always yellow. Then I would need 4 blue tiles and 4 yellow tiles (and the same 4 green tiles as before) for a total of 12 tiles. I have now almost completely designed T_{16}; however, I have not explained how to get the yellow tiles to form the diagonal line. To that end, I sandwich the 4 yellow tiles with the 2 tiles in $T_{diagonal}$ to create 8 tiles, represented by yellow and magenta in Figure 4.4(b). The union of green, blue, yellow, and magenta tiles makes up T_{16}.

In other words, in addition to the green, blue, and yellow tiles, $\mathbb{S}_{+16}$ also has magenta tiles that perform two jobs: propagating the information up just as the blue tiles and guiding the yellow diagonal line. Just as before, the first input number is coded on the bottom row and the second input number on the rightmost column of an L-configuration. The adder adds one bit of the column number to the row number, per row. The i^{th} bit has to be added at the i^{th} position, and the system uses the yellow diagonal line to compute that position. Each tile has 2 input sides (south and east) and two output sides (north and west). The north side is the value of the tile and the west side is the carry bit. The tile descriptions are easiest to read from Figure 4.4(b).

For answer-encoding purposes, the type of the tile is determined by the sum of the inputs, modulo 2. Figure 4.4(a) shows the concepts behind the tiles, and Figure 4.4(b) shows the actual 16 tiles. The 1 or 0 in the middle of the tile t is that tile's $v(t)$ value.

Figure 4.4(c) shows a sample seed configuration, which encodes two numbers in binary: $1010111_2 = 87$, $101101_2 = 45$, just as before. The number 87 is encoded in the bottom row and the number 45 is encoded in the rightmost column. There are 4 tiles in Γ_{16}, the 1 and 0 tiles for each of the inputs. Note that at the start, only one tile may attach to this configuration because $\tau_{16} = 2$ and there is only a single corner. Since no two tile types in T_{+16} have identical south-east binding domain pairs, by Corollary 4.1.3, $\mathbb{S}_{+16}$ produces a unique final configuration. Figure 4.4(d) shows the final configuration for the example of 87+45, with the solution encoded in the top row. The row reads 10000100_2 which is $132 = 87 + 35$. Note that in addition to the sum, the final construction "computes" a diagonal yellow (and also a magenta) line. Just as before, because the sum of two n-bit numbers may be as large as $n + 1$ bits, the row input needs to be padded with a single extra 0 tile as its most significant bit. The column input does not need this padding.

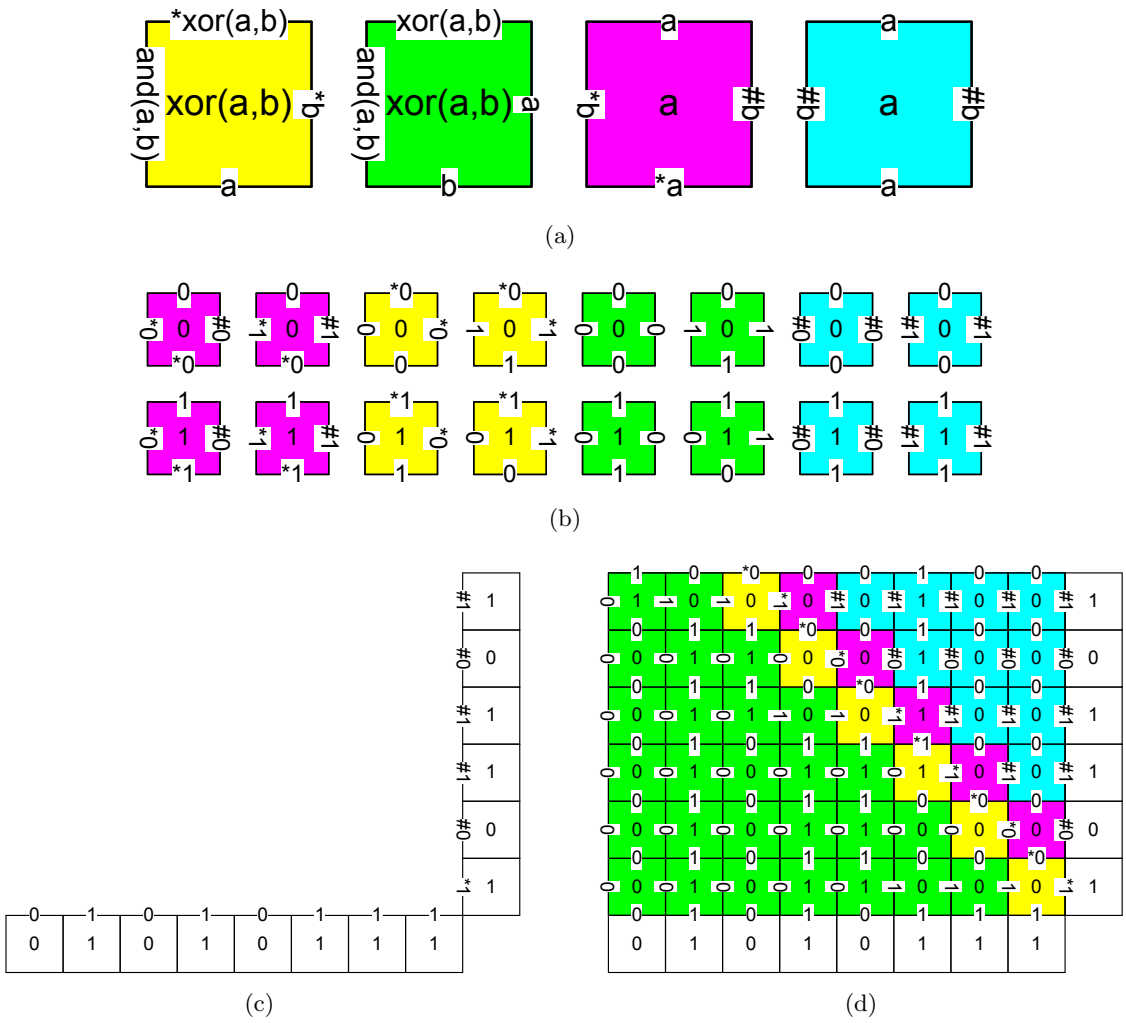


Figure 4.4: Tile system $\mathbb{S}_{+16}$ computes the function $f(a,b) = a + b$ and uses 16 distinct tile types. The tiles have 2 input sides (east and south) and 2 output sides (west and north). The north side is the value of the bit, and the west side is the carry bit (a). Each tile is labeled with a 1 or a 0 to assist the reader with reading the encoding v_{16}. There are 16 tiles in the system (b). Given a sample input of $a = 1010111_2 = 87$, and $b = 1011101_2 = 45$ (c), with 87 on bottom row and 45 on the rightmost column, the system fills the rectangle in the middle by adding a single bit of y in each row. The top row reads the solution: $a + b = 10000100_2 = 132$. Note that the least significant digit is on the right and that the color is purely to help the reader track tiles; the tiles themselves have no sense of color. Also note that the row input a has an extra 0 tile in the most significant bit place because the sum of two n-bit numbers may be as large as $n + 1$ bits.

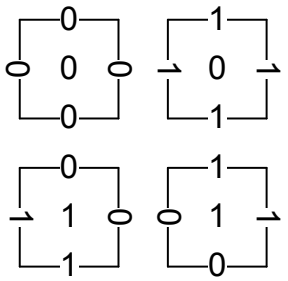


Figure 4.5: A set of tiles that perform a left-shift on a row of 0s and 1s. In binary, a left-shift is a multiplication by 2.

Each row adds a single bit of the column input at the correct location. Theorem 4.1.8 follows from the logic of the tiles (I do not bother to formally prove it here because the purpose of this system is to display ideas useful in designing the next system, and I will prove that system's correctness formally). By Corollary 4.1.5, the assembly time for adding two n-bit numbers is $\Theta(n)$ steps. Note that $|\Gamma_{16}| = 4$ and $|T_{16}| = 16$.

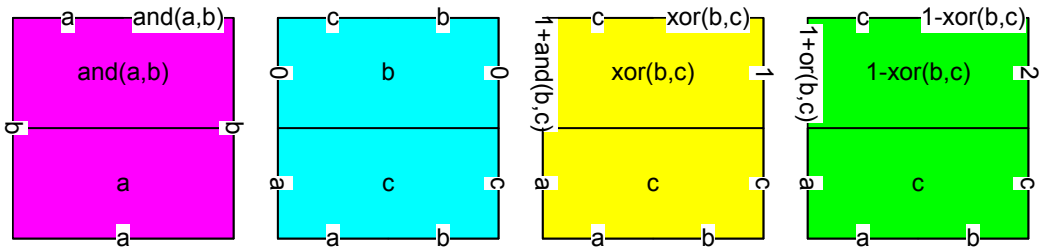
4.1.3 Multiplier Tile System

I have described two adder systems that start with an L-configuration and add a single bit per row to arrive at a row representing the sum of two numbers. I have also described briefly how to combine the functionality of two tile systems to produce a single tile system with properties of each of them. I will now illustrate how to combine these ideas to create a multiplier tile system. As with the last two adders, the multiplier will produce its output on the top row of a rectangle and allow the chaining of computations.

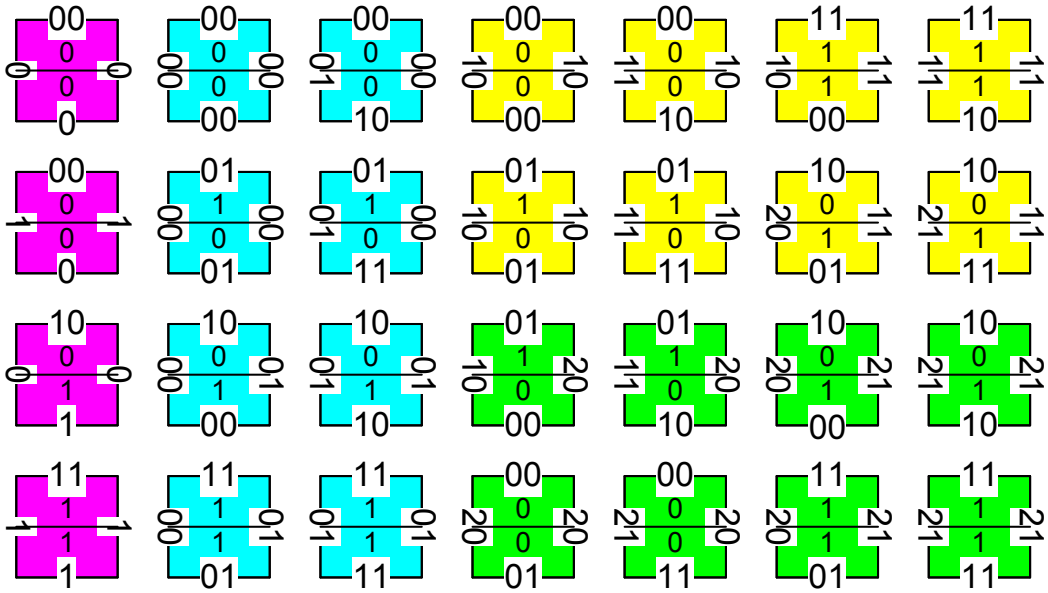
Observe that if $b = \sum_i b_i 2^i$, for $b_i \in \{0, 1\}$, that is, the i^{th} bit of b is b_i, then the product ab can be written as $ab = \sum_i b_i a 2^i$. That is, one can multiply a by each of the powers of 2 in b, and then add up the products. Multiplying by 2 in binary is simple — it is a left-shift. Figure 4.5 shows 4 tiles that would shift a row of 1 and 0 tiles to the left (I do not bother to define a formal system to do this, but just show the tiles). The tiles simply "flow" the information to the left and display the information received from the right. What's left is to add the rows representing the appropriate powers of 2 to construct the product of two numbers. The system should, therefore, add up to n numbers, as opposed to just 2. It is feasible to imagine such a system that on each row adds a new number to a running total, and arrives at the sum of n numbers on the n^{th} row. Thus I have informally described two sets of tiles: one that performs a left-shift and one that adds a number to the running total on each row. Sandwiching these two sets of tiles produces a set of tiles that sums up products of some input number a and powers of 2, i.e., computes $\sum_i a 2^i$. What is left is to add minor control to only sum up the appropriate powers of 2.

Theorem 4.1.9 *Let $\Sigma_\times = \{0, 1, 00, 01, 10, 11, 20, 21\}$, $g_\times = 1$, $\tau_\times = 2$, and $T_\times$ be a set of tiles over $\Sigma_\times$ as described in Figure 4.6(b). Then $\mathbb{S}_\times = \langle T_\times, g_\times, \tau_\times \rangle$ computes the function $f(a, b) = ab$.*

Figure 4.6 shows the tiles of $\mathbb{S}_\times$. Figure 4.7 shows an example execution of $\mathbb{S}_\times$ on $a = 87$ and $b = 45$. The input a, as before, is encoded in the bottom row and the input b is encoded in the rightmost column. There are 4 special magenta tiles which deal with the input and fill in the bottom row of the computation. These tiles are necessary because the least significant bit of the column input is 2^0 and thus requires no left-shift. The blue tiles code rows that have a 0 in the



(a)



(b)

Figure 4.6: Tile system $\mathbb{S}_\times$ computes the function $f(a, b) = ab$ and uses 28 distinct tile types. The tiles have 2 input sides (east and south) and 2 output sides (west and north) (a). There are 24 computation tiles and 4 special magenta tiles to start the computation (b). Note that the colors are solely for the reader; the tiles themselves have no sense of color.

input b. These tiles perform a left-shift, but do not add the current power of 2 to the running sum. The green and yellow tiles fill in the rows that have a 1 in the input b. Yellow tiles indicate that the incoming carry bit is 0, and green tiles indicate that the bit is 1. Each tile, in addition to its binding domains, is labeled with two pieces of information: the lower half of the tile on the i^{th} row displays the appropriate bit of $2^i a$ and the upper half of the tile displays the appropriate bit of the running sum so far. The top half of the top row displays the solution. In the example, the solution is $00111101001011_2 = 3915 = ab$.

There are 6 tiles in $\Gamma_\times$, the 1 and 0 tiles for each of the inputs and special 0 and 1 tiles for the least significant bit of b. Note that at the start, only one tile may attach to this configuration because $\tau_\times = 2$ and there is only a single corner.

Before, the inputs to the adder systems had to be padded by a single 0 bit. Because the product of two n-bit numbers may be as large as $2n$ bits, the row input to the multiplier system needs to be padded with n extra 0 tiles. The column input does not need this padding.

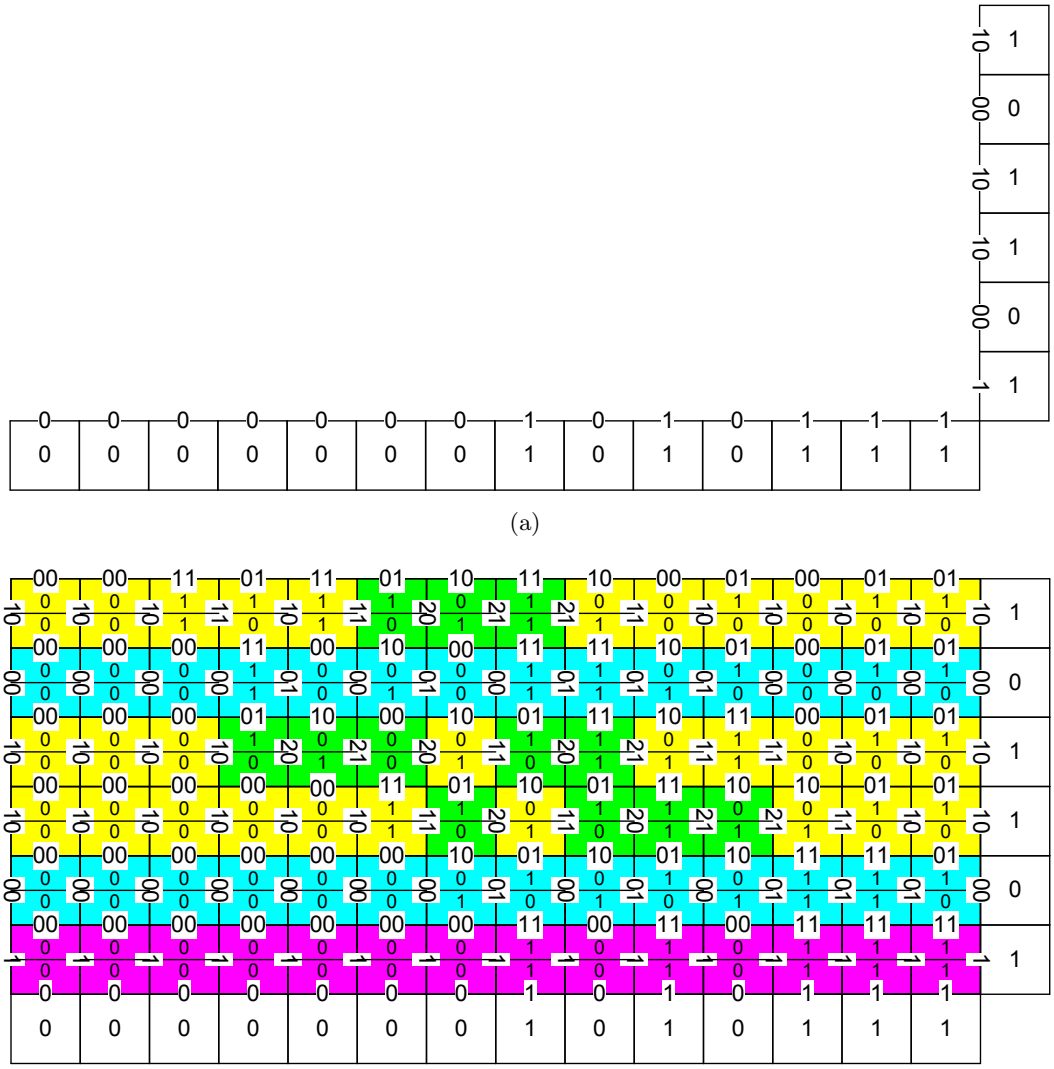


Figure 4.7: An example of $\mathbb{S}_\times$. Given a sample input of $a = 1010111_2 = 87$, and $b = 101101_2 = 45$ (a), with 87 on bottom row and 45 on the rightmost column, the system fills the rectangle (b). The i^{th} row has two pieces of information: on the lower half of each tile is the value of the appropriate bit of $2^i a$ and on the upper half is the running sum of the products of a and powers of 2 in b smaller or equals to i. The upper portion of the top row reads the solution: $ab = 00111101001011_2 = 3915 = 87 \cdot 45$. Note that the least significant digit is on the right and that the color is purely to help the reader track tiles; the tiles themselves have no sense of color. Also note that the row input a has n_b extra 0 tiles in the most significant bit places because the product of two n-bit numbers may be as large as $2n$ bits.

Proof: (Theorem 4.1.9) Consider the tile system $\mathbb{S}_\times$. Let a and b be the numbers to multiply and let n_a and n_b be the sizes, in bits, of a and b, respectively. For all $i \in \mathbb{N}$ let $a_i, b_i \in \{0,1\}$ be such that $a = \sum_i 2^i a_i$ and $b = \sum_i 2^i b_i$.

Let $\Gamma_\times = \{\alpha_0 = \langle 0, \mathit{null}, \mathit{null}, \mathit{null}\rangle, \alpha_1 = \langle 1, \mathit{null}, \mathit{null}, \mathit{null}\rangle, \beta_0 = \langle \mathit{null}, \mathit{null}, \mathit{null}, 00\rangle, \beta_1 = \langle \mathit{null}, \mathit{null}, \mathit{null}, 10\rangle, {}^\star\beta_0 = \langle \mathit{null}, \mathit{null}, \mathit{null}, 0\rangle, {}^\star\beta_1 = \langle \mathit{null}, \mathit{null}, \mathit{null}, 1\rangle$. Then the seed configuration $S\colon \mathbb{Z}^2 \to \Gamma_\times$ is such that

- $\forall i = 0, \ldots, n_a + n_b - 1$, $S(-i, -1) = \alpha_{a_i}$,
- $S(1,0) = {}^\star\beta_{b_0}$,
- $\forall i = 1, \ldots, n_b - 1$, $S(1, i) = \beta_{b_i}$, and
- for all other $(x, y) \in \mathbb{Z}^2$, $S(x, y) = \mathit{empty}$.

Since no two tile types in $T_\times$ have identical south-east binding domain pairs, by the unique final configuration corollary (Corollary 4.1.3), $\mathbb{S}_\times$ produces a unique final configuration on S. Let that configuration be F.

For those tiles with 2-bit binding domains bd, let $l(bd)$ be the first bit and $r(bd)$ be second bit of the binding domain. For all $t \in T$, let $v(t) = r(bd_N(t))$. I will show that:

1. $\forall 0 \leq i < n_a + n_b$, $l(bd_N(F(-i, 0))) = a_i$ and $r(bd_N(F(-i, 0))) = a_i$ if $b_0 = 1$ and 0 otherwise.
2. $\forall 0 \leq i < n_a + n_b$, $\forall 1 \leq j \leq n_b - 1$, all the following are true:
   - $l(bd_N(F(-i, j)))$ is the i^{th} bit of $a2^j$
   - $r(bd_N(F(-i, j))) = bit_i(a(b \bmod 2^{j+1}))$
   - $l(bd_W(F(-i, j))) = \begin{cases} 0 & \text{if } b_j = 0 \\ 1 + cbit_{i+1}(a(b \bmod 2^j), a2^j) & \text{otherwise} \end{cases}$
   - $r(bd_W(F(-i, j))) = bit_i(a2^{j-1})$

Proof of (1) (by induction): For all $0 \leq i < n_a + n_b$, S and F agree on $(-i, -1)$ and $(1, 0)$. Therefore, $bd_W(F(1, 0)) = b_0$. I assume that $bd_W(F(-(i-1), 0)) = b_{i-1}$ and I will show that $bd_W(F(-i, 0)) = b_i$, $l(bd_N(F(-i, 0))) = a_i$, and $r(bd_N(F(-i, 0))) = a_i$ if $b_0 = 1$ and 0 otherwise. Note that only magenta tiles may attach in this row because they are the only ones with a single bit east binding domain. For all such tiles t', $bd_W(t') = bd_E(t')$, so $bd_W(t') = b_0$. Also, $l(bd_N(t')) = bd_S(t')$, and since $F(-i, -1) = S(-i, -1)$ and $bd_N(S(-i, -1)) = a_i$, it follows that $l(bd_N(t')) = a_i$. Finally, $r(bd_N(F(-i, 0))) = 1$ iff $bd_S(t') = 1$ and $bd_E(t') = 1$, so $r(bd_N(F(-i, 0))) = a_i$ if $b_0 = 1$ and 0 otherwise.

Proof of (2) (by induction): Base cases: $\forall 0 \leq i < n_a + n_b$, $l(bd_N(F(-i, 0))) = a_i$ and $r(bd_N(F(-i, 0))) = a_i$ if $b_0 = 1$ and 0 otherwise (by (1)); $\forall 0 < j < n_b$, $l(bd_W(F(0, j))) = b_j$ and $r(bd_W(F(0, j))) = 0$. Inductive hypothesis: I assume that

- $l(bd_N(F(-i, j-1))) = bit_i(a2^{j-1})$
- $r(bd_N(F(-i, j-1))) = bit_i(a(b \bmod 2^j))$
- $l(bd_W(F(-(i-1), j))) = \begin{cases} 0 & \text{if } b_j = 0 \\ 1 + cbit_i(a(b \bmod 2^j), a2^j) & \text{otherwise} \end{cases}$
- $r(bd_W(F(-(i-1), j))) = bit_{i-1}(a2^{j-1})$.

and show that

(i) $l(bd_N(F(-i,j))) = bit_i(a2^j)$,

(ii) $r(bd_N(F(-i,j))) = bit_i(a(b \bmod 2^{j+1}))$,

(iii) $l(bd_W(F(-i,j))) = \begin{cases} 0 & \text{if } b_j = 0 \\ 1 + cbit_{i+1}(a(b \bmod 2^j), a2^j) & \text{otherwise} \end{cases}$

(iv) $r(bd_W(F(-i,j))) = bit_i(a2^{j-1})$.

Let $t = F(-i,j)$. Note that t cannot be magenta because for all $j \geq 1$ only non-magenta tiles may attach.

(i): For all non-magenta tiles in position $(-i,j)$:

$$\begin{aligned} l(bd_N(F(-i,j))) &= r(bd_E(F(-i,j))) \\ &= r(bd_W(F(-(i-1),j))) \\ &= bit_{i-1}(a2^{j-1}) \\ &= bit_i(a2^j). \end{aligned}$$

(ii and iii): There are three possible cases:

1. ($b_j = 0$): only a blue tile may attach in position $(-i,j)$.

   (ii):
   $$\begin{aligned} r(bd_N(F(-i,j))) &= r(bd_S(F(-i,j))) \\ &= r(bd_N(F(-i,j-1))) \\ &= bit_i(a(b \bmod 2^j)) \\ &= bit_i(a(b \bmod 2^{j+1})), \text{ since } b_j = 0. \end{aligned}$$

   (iii): $l(bd_W(F(-i,j))) = 0$ for all blue tiles, which is correct because $b_j = 0$.

2. ($b_j = 1$ and $cbit_i(a(b \bmod 2^j), a2^j) = 0$): only a yellow tile may attach in position $(-i,j)$ because $l(bd_W(F(-(i-1),j))) = 1$.

   (ii):
   $$\begin{aligned} r(bd_N(F(-i,j))) &= xor(r(bd_S(F(-i,j))), r(bd_E(F(-i,j)))) \\ &= xor(r(bd_N(F(-i,j-1))), r(bd_W(F(-(i-1),j)))) \\ &= xor(bit_i(a(b \bmod 2^j)), bit_{i-1}(a2^{j-1})) \\ &= xor(bit_i(a(b \bmod 2^j)), bit_i(a2^j)) \\ &= bit_i(a(b \bmod 2^{j+1})), \text{ because the incoming } cbit = 0. \end{aligned}$$

   (iii): for all yellow tiles,
   $$\begin{aligned} l(bd_W(F(-i,j))) &= 1 + and(r(bd_S(F(-i,j))), r(bd_E(F(-i,j)))) \\ &= 1 + and(r(bd_N(F(-i,j-1))), r(bd_W(F(-(i-1),j)))) \\ &= 1 + and(bit_i(a(b \bmod 2^j)), bit_{i-1}(a2^{j-1})) \\ &= 1 + and(bit_i(a(b \bmod 2^j)), bit_i(a2^j)) \\ &= 1 + cbit_{i+1}(a(b \bmod 2^j), a2^j). \end{aligned}$$

3. ($b_j = 1$ and $cbit_i(a(b \bmod 2^j), a2^j) = 1$): only a green tile may attach in position $(-i,j)$ because $l(bd_W(F(-(i-1),j))) = 2$.

   (ii):
   $$\begin{aligned} r(bd_N(F(-i,j))) &= 1 - xor(r(bd_S(F(-i,j))), r(bd_E(F(-i,j)))) \\ &= 1 - xor(r(bd_N(F(-i,j-1))), r(bd_W(F(-(i-1),j)))) \\ &= 1 - xor(bit_i(a(b \bmod 2^j)), bit_{i-1}(a2^{j-1})) \\ &= 1 - xor(bit_i(a(b \bmod 2^j)), bit_i(a2^j)) \\ &= bit_i(a(b \bmod 2^{j+1})), \text{ because the incoming } cbit = 1. \end{aligned}$$

(iii): for all green tiles,

$$\begin{aligned} l(bd_W(F(-i,j))) &= 1 + or(r(bd_S(F(-i,j))), r(bd_E(F(-i,j)))) \\ &= 1 + or(r(bd_N(F(-i,j-1))), r(bd_W(F(-(i-1),j)))) \\ &= 1 + or(bit_i(a(b \bmod 2^j)), bit_{i-1}(a2^{j-1})) \\ &= 1 + or(bit_i(a(b \bmod 2^j)), bit_i(a2^j)) \\ &= 1 + cbit_i(a(b \bmod 2^j), a2^j). \end{aligned}$$

(iv): For all non-magenta tiles in position $(-i, j)$:

$$\begin{aligned} r(bd_W(F(-i,j))) &= l(bd_S(F(-i,j))) \\ &= l(bd_N(F(-i,j-1))) \\ &= bit_i(a2^{j-1}). \end{aligned}$$

Thus, for all $0 \leq i < n_a + n_b$, $v(F(-i, n_b - 1)) = bit_i(a(b \bmod 2^{n_b}) = bit_i(ab)$. For all $j \in \mathbb{N}$, let $o_{s_1}(j) = (-j, -1)$, $o_{s_2}(j) = (1, j)$, and $o_f(j) = (-j, n_b)$. Because ab is no larger then $n_a + n_b$ bits, it follows that $\mathbb{S}_\times$ computes $f(a, b) = ab$. ■

By the assembly time corollary (Corollary 4.1.5), the assembly time for this system is $\Theta(n_a+n_b)$. Note that $|\Gamma_\times| = 6$ and $|T_\times| = 28$.

4.2 Factoring with Tiles

In this section, I will examine systems that factor positive integers; that is, systems that nondeterministically compute, for all $\zeta \geq 2$, $f(\zeta) = \langle \alpha, \beta \rangle$, such that $\alpha, \beta \geq 2$ and $\alpha\beta = \zeta$. I refer to such systems as factoring tile systems. These systems appear in my paper [26].

I will describe two factoring tile systems. To that end, I will introduce three tile systems, building up factoring functionality one step at a time. I will then combine those systems to create a system that factors at temperature four, and then discuss simplifying this factoring system to work at temperature three. All the systems use $\Theta(1)$ distinct tiles. The factoring systems, and the proofs of their correctness, are based in part on the multiplier system (one that deterministically computes $f(\alpha, \beta) = \alpha\beta$) from Section 4.1. Intuitively, this system will nondeterministically pick two numbers, multiply them, and then compare the result to the input. If the result and the input match, the assembly will include an identifier tile.

The factoring tile system will use a set of tiles T_4. I will define this set in three disjoint subsets: $T_4 = T_{\text{factors}} \cup T_\times \cup T_\checkmark$. The tiles in T_{factors} nondeterministically "guess" two factors; $T_\times$ is identical to $T_\times$ from Section 4.1 and multiply the two factors; and the tiles in $T_\checkmark$ ensure the computation is complete and compare the product of the factors to the input.

Whenever considering a number $\alpha \in \mathbb{N}$, I will refer to the size of α, in bits, as n_α. I will further refer to the i^{th} bit of α as α_i; that is, for all $i \in \mathbb{N}$, $\alpha_i \in \{0, 1\}$ such that $\sum_i \alpha_i 2^i = \alpha$. The least significant bit of α is α_0. Finally, I define $\lambda_\alpha \in \mathbb{N}_{\geq 1}$ to be the smallest positive integer such that $\alpha_{\lambda_\alpha} = 1$. For example, let $\alpha = 1000101_2$, then $n_\alpha = 7$, and $\lambda_\alpha = 2$ because the smallest positive power of 2 in 1000101_2 is 2^2. Of course, the same definitions extend to β, ζ and other variables.

The tile systems I will describe will use four types of tiles to encode the input number the system is factoring. Let the set of those tiles be $\Gamma_4 = \{\gamma_L = \langle null, null, s, null \rangle$, $\gamma_0 = \langle null, null, 0^\star, null \rangle$, $\gamma_1 = \langle null, null, 1^\star, null \rangle$, $\gamma_s = \langle null, null, |o|, s \rangle\}$. Figure 4.8 shows a graphical representation of Γ_4.

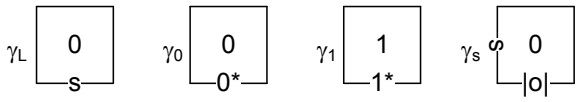


Figure 4.8: There are 4 tiles in Γ_4. The value in the middle of each tile t represents that tile's $v(t)$ value and each tile's name is written on its left.

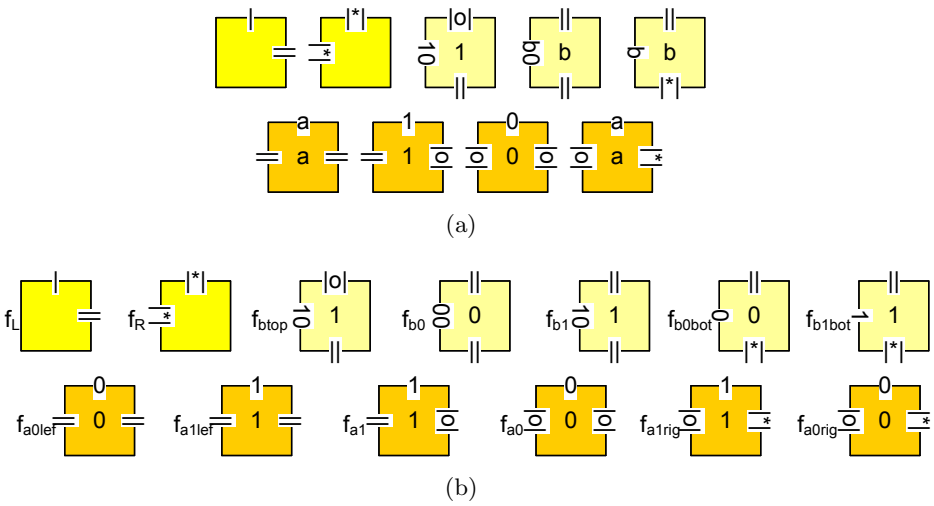


Figure 4.9: The concepts behind the tiles in T_{factors} include variables a and b (a). Each variable can take on as values the elements of the set $\{0, 1\}$. There are 13 actual tile types in T_{factors} (b). The value in the middle of each tile t indicates its $v(t)$ value (some tiles have no value in the middle because their v value will not be important and can be assumed to be 0) and each tile's name is written on its left.

4.2.1 Guessing the Factors

I first describe a tile system that, given a seed consisting of just a single tile will nondeterministically pick two natural binary numbers, α and β, such that $\alpha, \beta \geq 2$, and encode them using tiles. The system will use the set of tiles T_{factors}.

Figure 4.9(a) shows the concepts behind the tiles in T_{factors}. The concepts include variables a and b. Each variable can take on as values the elements of the set $\{0, 1\}$. Figure 4.9(b) shows the 13 actual tile types in T_{factors}. The symbols on the sides of the tiles indicate the binding domains of those sides. The value in the middle of each tile t indicates its $v(t)$ value (some tiles have no value in the middle because their v value will not be important and can be assumed to be 0).

Lemma 4.2.1 *Let $\Sigma_{\text{factors}} = \{|, ||, |^*|, |o|, 0, 1, 00, 10\}$. For all $\sigma \in \{|, ||, |^*|, |o|\}$, let $g_{\text{factors}}(\sigma, \sigma) = 4$ (for all other $\sigma \in \Sigma_{\text{factors}}$, $g(\sigma, \sigma)$ is not important for now). Let $\tau_{\text{factors}} = 4$. Let T_{factors} be as described in Figure 4.9(b). Let $\mathbb{S}_{\text{factors}} = \langle T_{\text{factors}}, g_{\text{factors}}, \tau_{\text{factors}} \rangle$. Let $S_{\text{factors}} \colon \mathbb{Z}^2 \to \{\gamma_s\}$ be a seed configuration with a single nonempty tile, such that $S(1, 1) = \gamma_s$ and $\forall x, y \in \mathbb{Z}$, $(x, y) \neq (1, 1) \implies S_{\text{factors}}(x, y) = \text{empty}$.*

Then for all $\alpha, \beta \geq 2$, $z \geq 0$, $\mathbb{S}_{\text{factors}}$ produces a final configuration F on S_{factors} such that:

1. $F(1, 1) = \gamma_s$
2. $F(1, 0) = f_{b\text{top}}$
3. $\forall i \in \{1, 2, \cdots, n_\beta - 2\}$, $F(1, -i) = f_{b\beta_k}$, *where* $k = n_\beta - i$

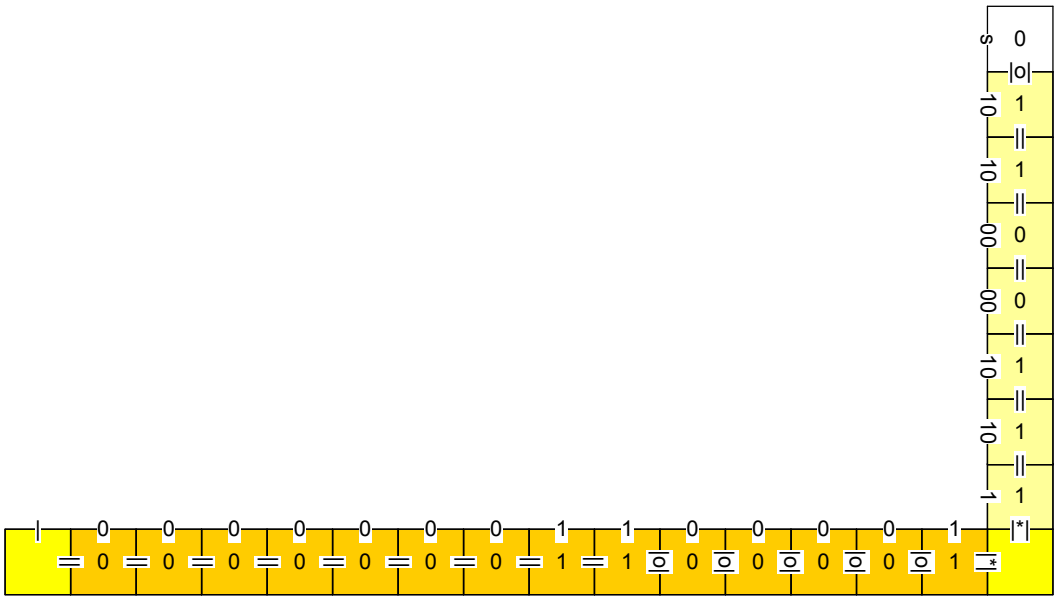


Figure 4.10: $\mathbb{S}_{\text{factors}}$ produces a final configuration F on S_{factors} such that F encodes two binary numbers. In this example, F encodes $103 = 1100111_2$ and $97 = 1100001_2$. Note that the number encoded in the lower row may have leading zeros; in this case it has 7 leading 0-tiles.

4. $F(1, -(n_\beta - 1)) = f_{b\beta_0\text{bot}}$
5. $F(1, -n_\beta) = f_R$
6. $F(0, -n_\beta) = f_{a\alpha_0\text{rig}}$
7. $\forall i \in \{1, 2, \cdots, \lambda_\alpha - 1\}$, $F(-i, -n_\beta) = f_{a0}$
8. $F(-\lambda_\alpha, -n_\beta) = f_{a1}$
9. $\forall i \in \{\lambda_\alpha + 1, \lambda_\alpha + 2, \cdots, n_\alpha - 1\}$, $F(-i, -n_\beta) = f_{a\alpha_i\text{lef}}$
10. $\forall i \in \{1, 2, \cdots, z\}$, $F(-(n_\alpha - 1 + i), -n_\beta) = f_{a0\text{lef}}$
11. $F(-(n_\alpha + z), -n_\beta) = f_L$

And for all final configurations F that $\mathbb{S}_{\text{factors}}$ produces on S_{factors}, F corresponds to some choice of $\alpha, \beta \geq 2$ and $z \geq 0$.

Intuitively, $\mathbb{S}_{\text{factors}}$ writes out, in order, the bits of α and β, for all $\alpha, \beta \geq 2$, and pads α with z extra 0-tiles, where $z \geq 0$. Figure 4.10 shows a sample final configuration that corresponds to $\beta = 103 = 1100111_2$, $\alpha = 97 = 1100001_2$, and $z = 7$.

Proof: Let $\alpha, \beta \geq 2$, let $z \geq 0$. I first show that tiles may attach to S_{factors} to produce a final configuration F that encodes α and β and pads β with z 0-tiles.

1. F and S_{factors} must agree at position $(1, 1)$, so $F(1, 1) = \gamma_s$.
2. Note that $bd_S(F(1, 1)) = |\text{o}|$ and the only tile type t with $bd_N(t) = |\text{o}|$ is $f_{b\text{top}}$, so $F(1, 0) = f_{b\text{top}}$.

3. Note that $bd_S(F(1,0)) = ||$, and there are four types of tiles t such that $bd_N(t) = ||$ (f_{b0}, f_{b1}, $f_{b0\text{bot}}$, and $f_{b1\text{bot}}$), so only those types of tiles may attach below. Two of those tile types have $bd_S(t) = ||$ (f_{b0} and f_{b1}), so again only those four tile types may attach below them. Therefore tiles may attach such that for all $i \in \{1, 2, \cdots, n_\beta - 2\}$, $F(1, -i) = f_{b\beta_k}$, where $k = n_\beta - i$.

4. Until finally, a tile of type $f_{b0\text{bot}}$ or $f_{b1\text{bot}}$ attaches so that $F(1, -(n_\beta - 1)) = f_{b\beta_0\text{bot}}$.

5. Note that $bd_S(F(1, -(n_\beta - 1))) = |^\star|$, and the only tile type t with $bd_N(t) = |^\star|$ is f_R, so $F(1, -n_\beta) = f_R$.

6. Note that $bd_W(F(1, -n_\beta)) = |^\star|$, and there are two types of tiles t such that $bd_E(t) = |^\star|$ ($f_{a0\text{rig}}$ and $f_{a1\text{rig}}$), so only types of tiles may attach at position $(0, -n_\beta)$. Thus the correct tile may attach such that $F(0, -n_\beta) = f_{a\alpha_0\text{rig}}$.

7. Note that $bd_W(F(0, -n_\beta)) = |\text{o}|$, and there are two types of tiles t such that $bd_E(t) = |\text{o}|$ (f_{a0} and f_{a1}), and only one of those two types has $bd_W(t) = |\text{o}|$ (f_{a0}). Therefore, for all $i \in \{1, 2, \cdots, \lambda_\alpha - 1\}$, f_{a0} can attach such that $F(-i, -n_\beta) = f_{a0}$.

8. Until finally, a tile of type f_{a1} attaches so that $F(-\lambda_\alpha, -n_\beta) = f_{a1}$.

9. Note that $bd_W(F(-\lambda_\alpha, -n_\beta)) = ||$, and there are three types of tiles t such that $bd_E(t) = ||$ ($f_{a0\text{lef}}$, $f_{a1\text{lef}}$, and f_L). Two of those tile types have $bd_W(t) = ||$ ($f_{a0\text{lef}}$ and $f_{a1\text{lef}}$), so again only those three tile types may attach to the left of them. Therefore, tiles may attach such that for all $i \in \{\lambda_\alpha + 1, \lambda_\alpha + 2, \cdots, n_\alpha - 1\}$, $F(-i, -n_\beta) = f_{a\alpha_i\text{lef}}$.

10. Similarly, to the west of the position $(-(n_\alpha - 1, -n_\beta))$, $f_{a0\text{lef}}$ may attach such that for all $i \in \{1, 2, \cdots, z\}$, $F(-(n_\alpha - 1 + i), -n_\beta) = f_{a0\text{lef}}$.

11. Until finally, a tile of type f_L attaches so that $F(-(n_\alpha + z), -n_\beta) = f_L$.

Let $v\colon T_{\text{factors}} \to \{0, 1\}$ be such that $v(f_{b0}) = v(f_{b0\text{bot}}) = v(f_{a0\text{lef}}) = v(f_{a0}) = v(f_{a0\text{rig}}) = v(f_L) = v(null) = 0$ and $v(f_{b\text{top}}) = v(f_{b1}) = v(f_{b1\text{bot}}) = v(f_{a1\text{lef}}) = v(f_{a1}) = v(f_{a1\text{rig}}) = 1$. For all other $t \in T_{\text{factors}}$ the $v(t)$ value does not matter and can be assumed to be either 0 or 1.

Note that:

- No more tiles may attach to F.
- $\sum_{i=0}^{\infty} v(F(1, -(n_\beta - 1 + i)))2^i = \beta$.
- $\sum_{i=0}^{\infty} v(F(-i, -n_\beta))2^i = \alpha$.
- For all $i \in \{1, 2, \cdots, z\}$, $v(F(-(n_\alpha - 1 + i), -n_\beta)) = 0$.

Thus for all choices of $\alpha, \beta \geq 2$, and $z \geq 0$, there exists a final configuration F produced by $\mathbb{S}_{\text{factors}}$ on S_{factors} that encodes α and β and pads α with z 0-tiles. Further note that for all final configurations F produced by $\mathbb{S}_{\text{factors}}$ on S_{factors}:

- F cannot encode $\beta < 2$ because $v(F(1, 0)) = 1$ implies that the most significant bit of β is 1 and $F(1, -1)$ cannot be f_R, thus β has at least 2 bits.
- F cannot encode $\alpha < 2$ because the tile encoding α_{λ_α} cannot be at position $(0, 1)$ and thus there exists a non-zero power of 2 in α.

Thus all final configurations F produced by $\mathbb{S}_{\text{factors}}$ on S_{factors} encode some $\alpha, \beta \geq 2$ and pad α with some $z \geq 0$ 0-tiles. ■

For a single choice of $\alpha, \beta \geq 2$, there are several final configurations that encode α and β. They differ in the choice of z. I am interested only in two of those final configurations, for $z = n_\beta$ and for $z = n_\beta - 1$, because the product of α and β is either $n_\alpha + n_\beta$ or $n_\alpha + n_\beta - 1$ bits long.

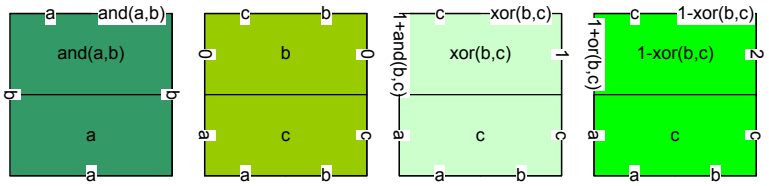
Lemma 4.2.2 (Factors assembly time lemma) *For all $\alpha, \beta \geq 2$, the assembly time of the final configuration F produced by $\mathbb{S}_{\text{factors}}$ on S_{factors} that encodes α and β and pads α with n_β or $n_\beta - 1$ 0-tiles is $\Theta(n_\alpha + n_\beta)$.*

Proof: For each tile in F to attach, a tile in a specific location must have attached before it (either to the north or to the east). Thus there is no parallelism in this assembly, and the assembly time equals the total number of tiles that attach, which is $\Theta(n_\alpha + n_\beta)$. ■

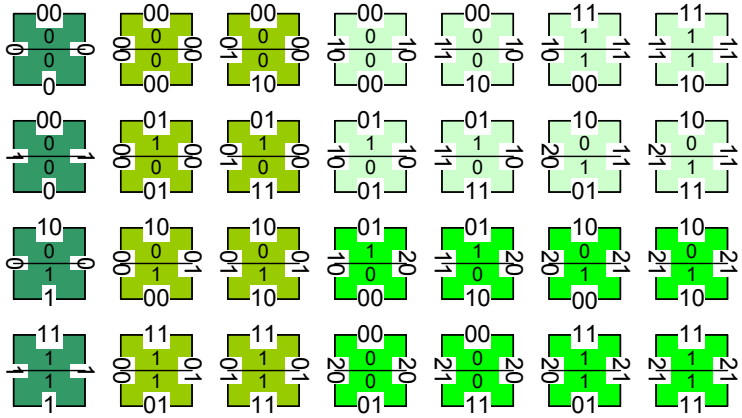
Lemma 4.2.3 *Let each tile that may attach to a configuration at a certain position attach there with a uniform probability distribution. For all $\alpha, \beta \geq 2$, let $\delta = \alpha\beta$. Then, the probability of assembling a particular final configuration encoding α and β and padding α with $n_\delta - n_\alpha$ 0-tiles is at least $\left(\frac{1}{4}\right)^{n_\beta}\left(\frac{1}{3}\right)^{n_\delta}$.*

Proof: I will calculate the probabilities of each tile attaching in its proper position and then, since the probabilities of attachments are independent, multiply those probabilities together to get the overall probability of a final configuration.

1. The seed automatically becomes part of the final configuration with probability $p_1 = 1$.
2. There is only 1 tile that may attach in position $(1, 0)$, so it attaches with probability $p_2 = 1$.
3. For the next $n_\beta - 2$ positions, out of 4 possible tiles that may attach, only 1 is the correct one, so the probability that the next $n_\beta - 2$ tiles attach correctly is $p_3 = \left(\frac{1}{4}\right)^{n_\beta - 2}$.
4. The tile representing the last bit of β is also 1 out of the possible 4 so the probability of it attaching is $p_4 = \frac{1}{4}$.
5. Only 1 tile may attach below the 0^{th} bit of β, so its probability of attaching is $p_5 = 1$.
6. There are 2 possible tiles that may attach to represent α's 0^{th} bit, and only 1 is correct, so the probability of it attaching is $p_6 = \frac{1}{2}$.
7. The next $\lambda_\alpha - 1$ tiles must be 1 of 2 possible tiles, so the probability of all of them attaching correctly is $p_7 = \left(\frac{1}{2}\right)^{\lambda_\alpha - 1}$.
8. The next tile encodes the smallest positive power of 2 in α and is 1 of 2 possible tiles, so its probability of attaching is $p_8 = \frac{1}{2}$.
9. The rest of the bits of α can be encoded using 1 specific tile from the 3 possibilities, so the probability of all of them attaching correctly is $p_9 = \left(\frac{1}{3}\right)^{n_\alpha - \lambda_\alpha - 1}$.
10. The next $n_\delta - n_\alpha$ tiles, depending on the desired final configuration, must be 1 of 3 possible tiles, so the probability of all of them attaching correctly is $p_{10} = \left(\frac{1}{3}\right)^{n_\delta - n_\alpha}$.
11. Finally, the probability of the last tile attaching is $p_{11} = \frac{1}{3}$.



(a)



(b)

Figure 4.11: The concepts behind the tiles in $T_\times$ (a) include variables a, b, and c, each of which can take on as values the elements of the set $\{0, 1\}$. There are 28 actual tile types in $T_\times$ (b). Note that for each tile t, its $v(t)$ value is indicated in the upper half of the middle of the tile.

The overall probability of a specific final configuration is:
$\prod_{i=1}^{11} p_i \geq \left(\frac{1}{4}\right)^{n_\beta - 1} \left(\frac{1}{2}\right)^{\lambda_\alpha + 1} \left(\frac{1}{3}\right)^{n_\alpha - \lambda_\alpha + n_\delta - n_\alpha} \geq \left(\frac{1}{4}\right)^{n_\beta} \left(\frac{1}{3}\right)^{n_\delta}$. ■

Corollary 4.2.4 *Let each tile that may attach to a configuration at a certain position attach there with a uniform probability distribution. For all $\alpha \geq \beta \geq 2$, let $\delta = \alpha\beta$. Then, the probability of assembling a particular final configuration encoding α and β and padding α with $n_\delta - n_\alpha$ 0-tiles is at least $\left(\frac{1}{6}\right)^{n_\delta}$.*

Proof: By Lemma 4.2.3, the probability of assembling a particular final configuration encoding α and β and padding α with $n_\delta - n_\alpha$ 0-tiles is at least $\left(\frac{1}{4}\right)^{n_\beta} \left(\frac{1}{3}\right)^{n_\delta}$. Because $\alpha \geq \beta$, $n_\delta \geq 2n_\beta$, so $\left(\frac{1}{4}\right)^{n_\beta} \left(\frac{1}{3}\right)^{n_\delta} \geq \left(\frac{1}{2}\right)^{n_\delta} \left(\frac{1}{3}\right)^{n_\delta} = \left(\frac{1}{6}\right)^{n_\delta}$. ■

4.2.2 Multiplying the Factors

I have just described a tile system that nondeterministically assembles the representations of two binary numbers. I will now add tiles to multiply those numbers.

Figure 4.11(a) shows the concepts behind the tiles in $T_\times$. The concepts include variables a, b, and c, each of which can take on as values the elements of the set $\{0, 1\}$. Figure 4.11(b) shows the 28 actual tile types in $T_\times$. These tiles are identical to the multiplier tiles from Section 4.1, thus I will reference Theorem 4.1.9.

Corollary 4.2.5 *For all $\sigma \in \Sigma_\times$, let $g'_\times(\sigma, \sigma) = 2$ and $\tau'_\times = 4$. Then $\mathbb{S}'_\times = \langle T_\times, g'_\times, \tau'_\times \rangle$ computes the function $f(\alpha, \beta) = \alpha\beta$.*

Proof: For all $\alpha, \beta \in \mathbb{N}$, $\mathbb{S}_\times$ computes the function $f(\alpha, \beta) = \alpha\beta$. Therefore, there exists some seed S_0, which encodes α and β, such that $\mathbb{S}_\times$ produces a unique final configuration F on S_0. Further, there exists at least one sequence of attachments $W = \langle\langle t_0, (x_0, y_0)\rangle, \langle t_1, (x_1, y_1)\rangle, \cdots, \langle t_k, (x_k, y_k)\rangle\rangle$ such that t_0 attaches at position (x_0, y_0) to S_0 to produce S_1, t_1 attaches at position (x_1, y_1) to S_1 to produce S_2, and so on to produce the final configuration $S_{k+1} = F$. Since for all $\sigma \in \Sigma_\times$, $g_\times(\sigma, \sigma) = 1$ and $\tau_\times = 2$, each tile t_i must have at least two non-empty neighbors to position (x_i, y_i) in S_i whose appropriate binding domains match t_i's binding domains.

By the unique final configuration corollary (Corollary 4.1.3), $\mathbb{S}'_\times$ produces a unique final configuration on S_0. Since for all i, each tile t_i has at least two non-empty neighbors to position (x_i, y_i) in S_i whose appropriate binding domains match t_i's binding domains, and for all $\sigma \in \Sigma_\times$, $g'_\times(\sigma, \sigma) = 2$ and $\tau'_\times = 4$, the tile t_i can attach to S_i to form S_{i+1}. Thus W is a valid sequence of attachments for $\mathbb{S}'_\times$ to S_0 and $\mathbb{S}'_\times$ must produce the unique final configuration $S_{k+1} = F$ on S_0, and thus compute the function $f(\alpha, \beta) = \alpha\beta$. ■

Intuitively, Corollary 4.2.5 says that doubling the strength of every binding domain and the temperature does not alter the logic of assembly of $\mathbb{S}_\times$. To compute the product of two numbers α and β, $\mathbb{S}_\times$ attaches tiles to a seed configuration, which encodes α and β, to reach a final configuration, which encodes $\alpha\beta$. The logic of the system dictates that every time a tile attaches, its south and east neighbors have already attached, and its west and north neighbors have not attached. Thus by doubling the strength of every binding domain and the temperature, a tile t can attach at position (x, y) in $\mathbb{S}'_\times$ iff t also attached at (x, y) in $\mathbb{S}_\times$. Thus the final configurations in the two systems will be identical, and therefore, $\mathbb{S}'_\times$ computes the function $f(\alpha, \beta) = \alpha\beta$.

Figure 4.12(a) shows a sample seed configuration $S_\times$, which encodes $\beta = 103 = 1100111_2$ and $\alpha = 97 = 1100001_2$. Figure 4.12(b) shows the final configuration F that $\mathbb{S}'_\times$ produces of $S_\times$. Along the top row of F, the binary number $10011100000111_2 = 9991$ encodes the product $\alpha\beta = 97 \cdot 103 = 9991$. It is not a coincidence that $S_\times$ looks exactly like a final configuration that $\mathbb{S}_{\text{factors}}$ produces on S_{factors}.

Lemma 4.2.6 *Let $\alpha, \beta \geq 2$ and let $S_\times$ be the final configuration produced by $\mathbb{S}_{\text{factors}}$ on S_{factors} that encodes α and β and has n_β padding 0-tiles. Let F be the unique final configuration that $\mathbb{S}'_\times$ produces on $S_\times$. Let $v\colon T_\times \to \{0, 1\}$ be defined as in Figure 4.12. Then F encodes the binary number $\delta = \alpha\beta = \sum_{i=0}^{\infty} v(F(-i, 0))2^i$. That is, the i^{th} bit of δ is $v(F(-i, 0))$. Further, if $\alpha\beta$ has only $n_\alpha + n_\beta - 1$ bits, then it is sufficient to pad S_{factors} with $n_\beta - 1$ 0-tiles.*

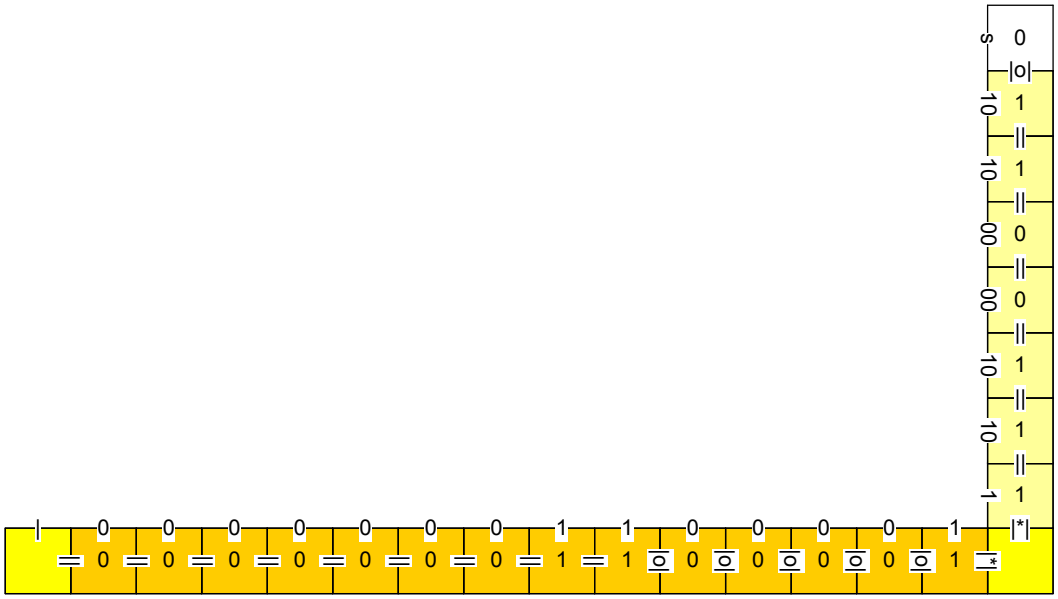
The lemma follows directly from the proof of Corollary 4.2.5 and Theorem 4.1.9. $S_\times$ forms the sides of a rectangle that has the same binding domains internal to the rectangle as the seed that $\mathbb{S}_\times$ uses in Section 4.1. $\mathbb{S}'_\times$ produces, on $S_\times$ the final configuration F, which encodes the product of the inputs on precisely the 0^{th} row, with the i^{th} bit of the product being $v(F(0, -i))$. Note that the product of two binary numbers α and β will always have either $n_\alpha + n_\beta$ or $n_\alpha + n_\beta - 1$ bits.

Lemma 4.2.7 (Multiplication assembly time lemma) *For all $\alpha, \beta \geq 2$, the assembly time for $\mathbb{S}_\times$ on a seed encoding α and β and padding α with n_β or $n_\beta - 1$ 0-tiles is $\Theta(n_\alpha + n_\beta)$.*

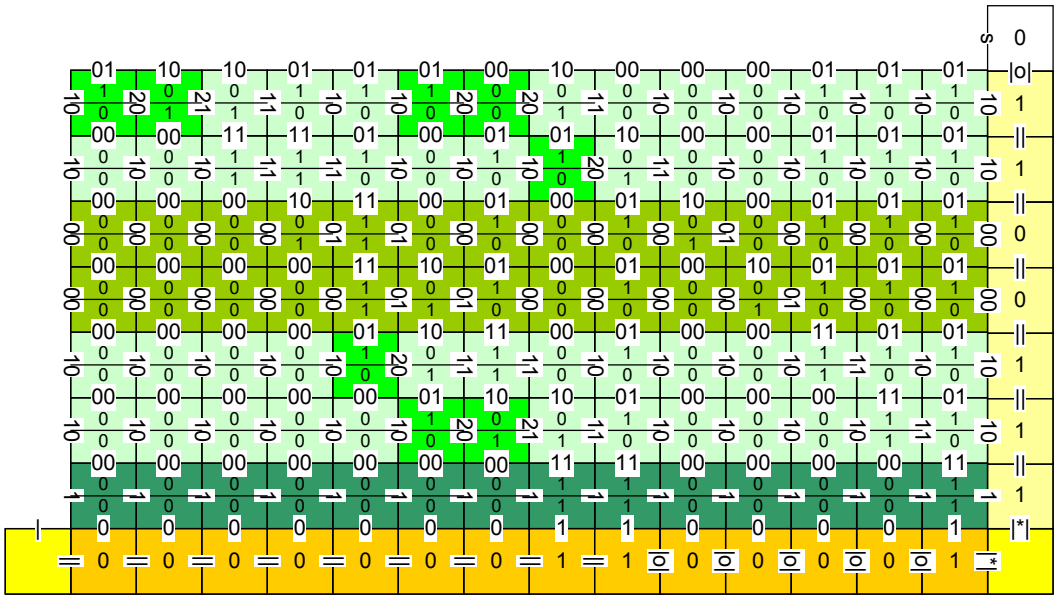
This lemma follows directly from the assembly time corollary (Corollary 4.1.5).

4.2.3 Checking the Product

I have described two systems: one to nondeterministically produce two random numbers and another to multiply them. I now present one more system with two functions: to make sure the multiplication is complete and to compare an input to the result of the multiplication.



(a)



(b)

Figure 4.12: A sample seed configuration $S_\times$ that encodes $\beta = 103 = 1100111_2$ and $\alpha = 97 = 1100001_2$ (a). Note that $S_\times$ is identical to a final configuration produced by $\mathbb{S}_{\text{factors}}$ on S_{factors}. $\mathbb{S}'_\times$ produces a final configuration F on $S_\times$, which encodes the product $\alpha\beta = 97 \cdot 103 = 9991 = 10011100000111_2$ along the top row (b).

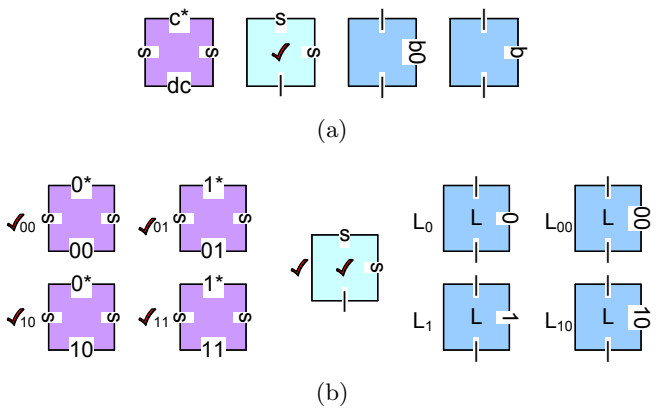


Figure 4.13: The concepts behind the tiles in $T_{\checkmark}$ (a) include variables b, c and d, each of which can take on as values the elements of the set $\{0, 1\}$. There are 9 actual tile types in $T_{\checkmark}$ (b); each tile's name is written on its left. The tile with a check mark in the middle will serve as the identifier tile.

Figure 4.13(a) shows the concepts behind the tiles in $T_{\checkmark}$. The concepts include variables b, c and d, each of which can take on as values the elements of the set $\{0, 1\}$. Figure 4.13(b) shows the 9 actual tile types in $T_{\checkmark}$. The tile with a check mark in the middle will serve as the identifier tile.

Lemma 4.2.8 *Let* $\Sigma_{\checkmark} = \{|, 0, 1, 00, 01, 10, 11, 0^{\star}, 1^{\star}, s\}$. *For all* $\sigma \in \{0^{\star}, 1^{\star}, s\}$, *let* $g_{\checkmark}(\sigma, \sigma) = 1$ *and for all* $\sigma' \in \{|, 0, 1, 00, 01, 10, 11\}$, *let* $g_{\checkmark}(\sigma', \sigma') = 2$. *Let* $\tau_{\checkmark} = 4$. *Let* $T_{\checkmark}$ *be as described in Figure 4.13(b). For all* $\alpha, \beta, \zeta \geq 2$, *let* $\delta = \alpha\beta$, *let* $S_{\times}$ *be the final configuration produced by* $\mathbb{S}_{\text{factors}}$ *on* S_{factors} *that encodes* α *and* β *and has* $n_{\delta} - n_{\alpha}$ *padding* 0*-tiles. Let* $F_{\times}$ *be the unique final configuration that* $\mathbb{S}'_{\times}$ *produces on* $S_{\times}$. *Let* $S_{\checkmark}$ *be such that:*

- $\forall x, y \in \mathbb{Z}$, $F_{\times}(x, y) \neq \textit{empty} \implies S_{\checkmark}(x, y) = F_{\times}(x, y)$.
- $S_{\checkmark}(-n_{\zeta}, 2) = \gamma_L$.
- $\forall i \in \{0, 1, \cdots, n_{\zeta} - 1\}$, $S_{\checkmark}(-i, 2) = \gamma_{\zeta_i}$.

Let $\mathbb{S}_{\checkmark} = \langle T_{\checkmark}, g_{\checkmark}, \tau_{\checkmark} \rangle$. *Then* $\mathbb{S}_{\checkmark}$ *produces a unique final configuration* $F_{\checkmark}$ *on* $S_{\checkmark}$ *and* $\alpha\beta = \zeta$ *iff* $F_{\checkmark}(-n_{\zeta}, 1) = \checkmark$.

Proof: First, observe that if the ✓ tile is ever to attach, it must attach in position $(-n_{\zeta}, 1)$ for three reasons. First, since the sum of the g values of the binding domains of ✓ is exactly 4, it must match its neighbors on all sides with non-null binding domains to attach at temperature 4. Second, the only tile with a south binding domain s, matching ✓'s north binding domain, is γ_L. And third, γ_L can only occur in the seed, and only in position $(-n_{\zeta}, 2)$.

Consider $F_{\checkmark}$. Working backwards, for a ✓ tile to attach at position $(-n_{\zeta}, 1)$, the tile directly south, at position $(-n_{\zeta}, 0)$ must have a north binding domain $|$. Therefore, that tile is one of the following four tiles: $\{L_0, L_1, L_{00}, L_{10}\}$. For any one of those four tiles to attach, its east neighbor's west binding domain must be 0, 1, 00 or 10, meaning two things: the first bit of the binding domain cannot be 2, which implies that the carry bit of the tile to the east is 0, and the second bit of the binding domain cannot be 1, which implies that shifted α has not run past the west bound of the computation. Together, those two properties ensure that the multiplication is proceeding correctly. In turn, the tile directly south of that position, at position $(-n_{\zeta}, -1)$, by the same reasoning, must be one of those four tiles and the tile to its east has the two above properties. Similarly, all the

tiles in the column $-n_\zeta$ must be one of those four tiles, until the position $(-n_\zeta, -n_\beta)$, where the tile must be f_L (by the definition, $S_\checkmark(-n_\delta, -n_\beta) = f_L$). Therefore, the ✓ tile may attach only if $n_\delta = n_\zeta$ and every row of the multiplication has a 0 carry bit and has not overshifted α.

Working backwards again, for a ✓ tile to attach at position $(-n_\zeta, 1)$ in $F_\checkmark$, the tile directly east must have a west binding domain s, and thus must be one of the following four tiles: $\{\checkmark_{00}, \checkmark_{01}, \checkmark_{10}, \checkmark_{11}\}$. For each of those tiles, the first digit of its north binding domain matches the second digit of its south binding domain. Therefore, $v(F_\checkmark(-(n_\zeta - 1), 0))$ must equal $v(F_\checkmark(-(n_\zeta - 1), 2))$. The same argument holds for the tile to the east of that tile, and so on until position $(1, 1)$, where $F_\checkmark(1, 1) = \gamma_s$. By Lemma 4.2.6, if $\delta = \alpha\beta$, then $\forall i \in \mathbb{N}$, $v(S_\checkmark(-i, 0)) = \delta_i$. Therefore, a ✓ tile may attach at position $(-n_\zeta, 1)$ to $F_\checkmark$ if and only if $\gamma = \alpha\beta$. ■

Figure 4.14(a) shows a sample seed $S_\checkmark$ for $\beta = 103 = 1100111_2$, $\alpha = 97 = 1100001_2$, and $\zeta = 9991 = 10011100000111_2$. Figure 4.14(b) shows the final configuration $F_\checkmark$ that $\mathbb{S}_\checkmark$ produces on $S_\checkmark$. Because $\zeta = \alpha\beta$, $F_\checkmark$ contains the tile ✓.

Lemma 4.2.9 (Checking assembly time lemma) *For all $\alpha, \beta, \zeta \geq 2$, if $\zeta = \alpha\beta$ then the assembly time of the final configuration F produced by $\mathbb{S}_\checkmark$ on $S_\checkmark$ that encodes α, β, and ζ and pads α with n_β or $n_\beta - 1$ 0-tiles is $\Theta(n_\zeta)$.*

Proof: Each tile in the 0^{th} row of F that attaches requires one other specific tile to attach first. The same is true for column n_ζ. Thus there is no parallelism in each of those two assembling processes, and the assembly time for each process equals the total number of tiles that attach in that process, thus the overall assembly time is $\Theta(\max(n_\zeta, n_\beta)) = \Theta(n_\zeta)$. ■

4.2.4 Factoring at Temperature Four

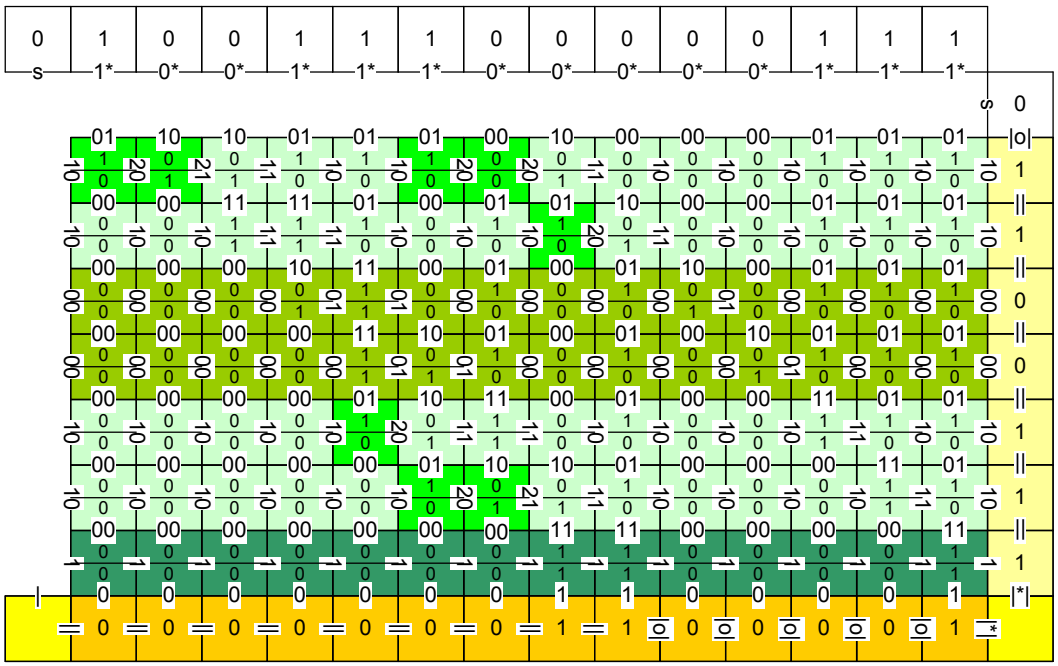
I have defined three different systems that perform the necessary pieces of factoring a number. I now put them together into a single system $\mathbb{S}_4$ and argue that $\mathbb{S}_4$ is a factoring system.

Theorem 4.2.10 (Temperature four factoring theorem) *Let $\Sigma_4 = \Sigma_{\text{factors}} \cup \Sigma_\times \cup \Sigma_\checkmark$. Let $T_4 = T_{\text{factors}} \cup T_\times \cup T_\checkmark$. Let g_4 be such that g_4 agrees with g_{factors}, $g'_\times$, and $g_\checkmark$ on their respective domains (note that for all elements of the domains of more than one of those functions, those functions agree). Let $\tau_4 = 4$. Then the system $\mathbb{S}_4 = \langle T_4, g_4, \tau_4 \rangle$ is a factoring tile system.*

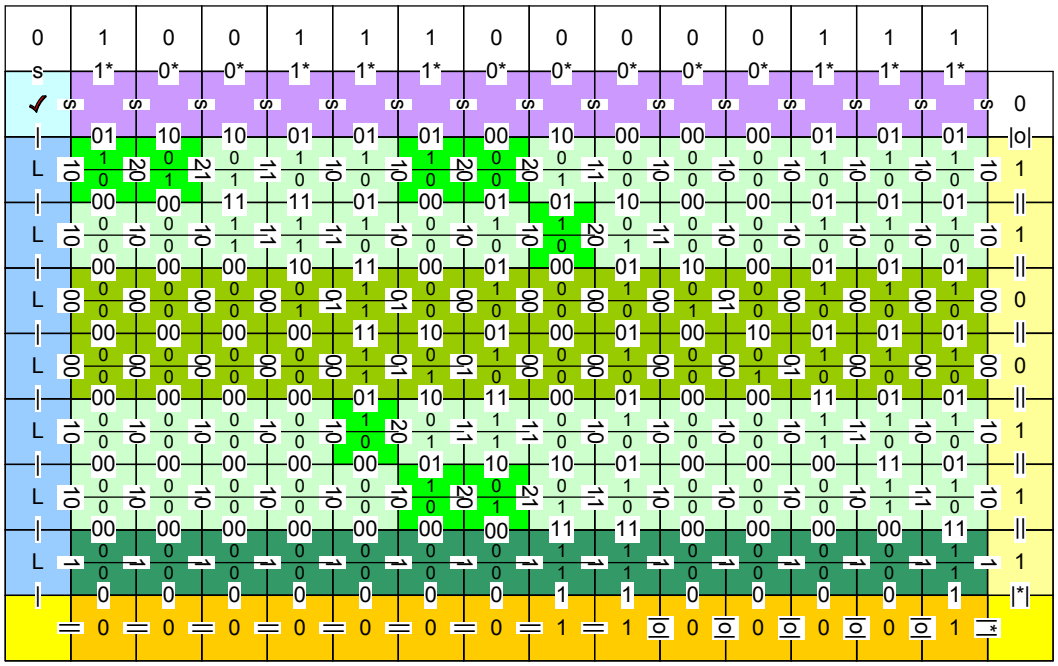
If a tile system is the combination of three distinct tile systems, the behavior of the large system need not be the combined behavior of the three smaller systems — the tiles from different systems can interfere with each other. However, I have designed $\mathbb{S}_{\text{factors}}$, $\mathbb{S}'_\times$, and $\mathbb{S}_\checkmark$ to work together, without interfering. For the most part, each system uses a disjoint set of binding domains, sharing binding domains only where tiles from the different systems are designed to interact. As a result, tiles from each system have a particular set of positions where they can attach: tiles from T_{factors} can only attach in column 1 and row $-n_\beta$, tiles from $T_\times$ can only attach in the rectangle defined by the rows 0 and $-(n_\beta - 1)$ and columns 0 and $-(n_\zeta - 1)$, and tiles from $T_\checkmark$ can only attach in column $-n_\zeta$ and row 1, thus the tiles do not interfere with each other.

Proof: (Theorem 4.2.10): For all $\zeta \geq 2$, Let S_4 be such that

- $S_4(1, 1) = \gamma_s$.
- $S_4(-n_\zeta, 2) = \gamma_L$.
- $\forall i \in \mathbb{N}$ such that $0 \leq i < n_\zeta$, $S_4(-i, 2) = \gamma_{\zeta_i}$



(a)



(b)

Figure 4.14: A sample seed configuration $S_{\checkmark}$ that encodes $\beta = 103 = 1100111_2$, $\alpha = 97 = 1100001_2$, and $\zeta = 9991 = 10011100000111_2$ (a). $\mathbb{S}_{\checkmark}$ produces a final configuration F on $S_{\checkmark}$, which includes a ✓ tile iff $\zeta = \alpha\beta$ (b).

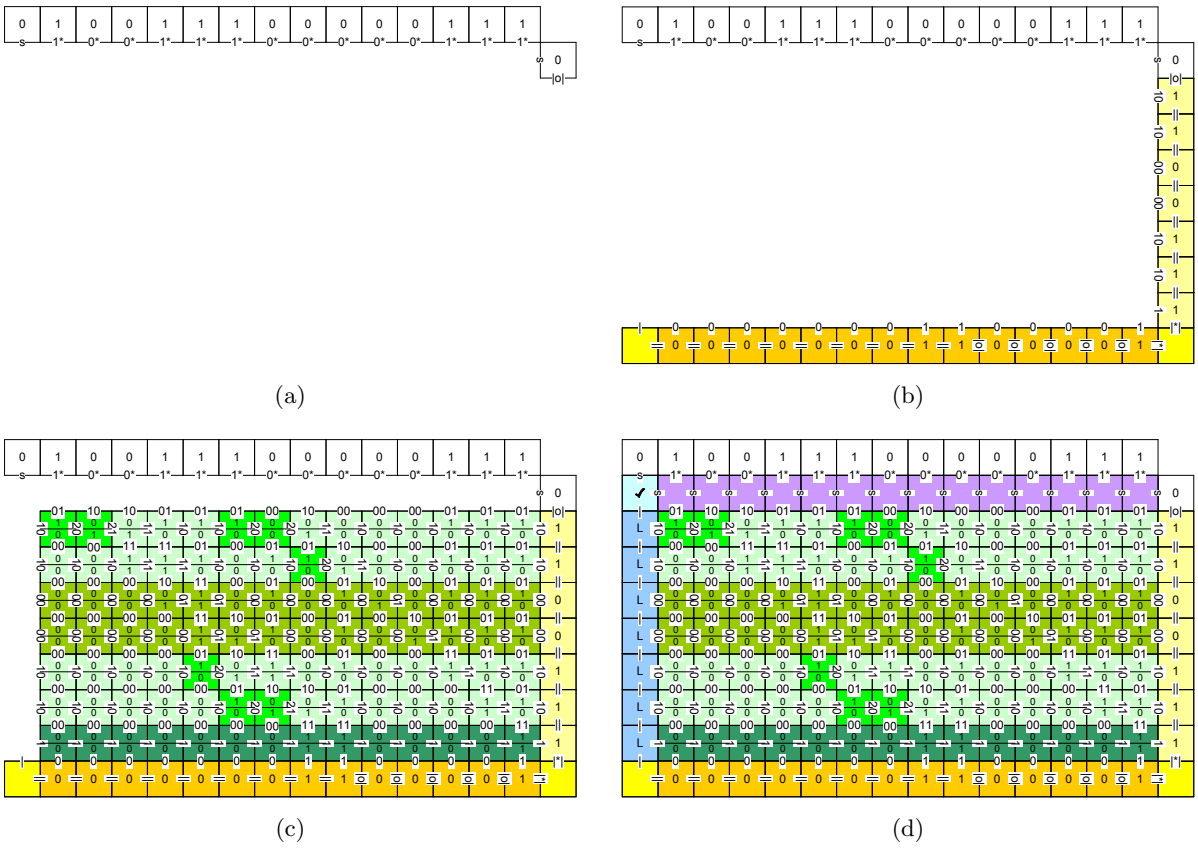


Figure 4.15: $\mathbb{S}_4$ factors a number ζ encoded in the seed configuration, e.g., $\zeta = 9991 = 10011100000111_2$ (a). In one sequence of possible attachments, encodings of two randomly selected numbers α and β attach nondeterministically to the seed such that $\alpha, \beta \geq 2$, e.g., $\alpha = 97 = 1100001_2$, $\beta = 103 = 1100111_2$ (b). $\mathbb{S}_4$ then deterministically multiplies α and β to encode $\delta = \alpha\beta$ (c), and then ensures the multiplication is complete and compares δ to ζ to check if they are equal. If they are equal, a special ✓ tile attaches to signify that the factors have been found (d).

For all binary numbers $\alpha, \beta \geq 2$, and for all $z \geq 0$, tiles from T_4 will attach to S_4 as follows: the tiles from T_{factors} nondeterministically attach to encode two binary numbers: α in column 1 and β in row $-n_\beta$, padding α with z 0-tiles, as described in Lemma 4.2.1. By Lemma 4.2.6 the tiles from $T_\times$ attach, in the rectangle defined by the rows 0 and $-(n_\beta - 1)$ and columns 0 and $-(n_\alpha + z - 1)$, to encode $\delta = \alpha\beta$ in the top row. Finally, tiles from $T_\checkmark$ attach, in column $-n_\zeta$ and row 1, such that, by Lemma 4.2.8 the ✓ tile only attaches if $\delta = \zeta$ and $z = n_\zeta - n_\alpha \in \{n_\beta, n_\beta - 1\}$. Let the identifier tile be ✓. Thus only if there exists a choice of $\alpha, \beta \geq 2$ such that $\alpha\beta = \zeta$ will the identifier tile attach. Further, if the identifier tile does attach to a configuration F_4, then F_4 encodes α and β as defined in Lemma 4.2.1. Therefore, $\mathbb{S}_4$ nondeterministically and identifiably computes the function $f(\zeta) = \langle \alpha, \beta \rangle$. ■

Figure 4.15(a) shows the seed configuration S_4 for $\zeta = 9991 = 1100000111_2$. Figures 4.15(b), 4.15(c), and 4.15(d) show one possible progression of tiles attaching to produce a final configuration that $\mathbb{S}_4$ produces on S_4.

I mentioned in Section 3.2 that the factoring system I describe does not exactly fit the definition of a system computing a function nondeterministically. The reason is that the two factors are

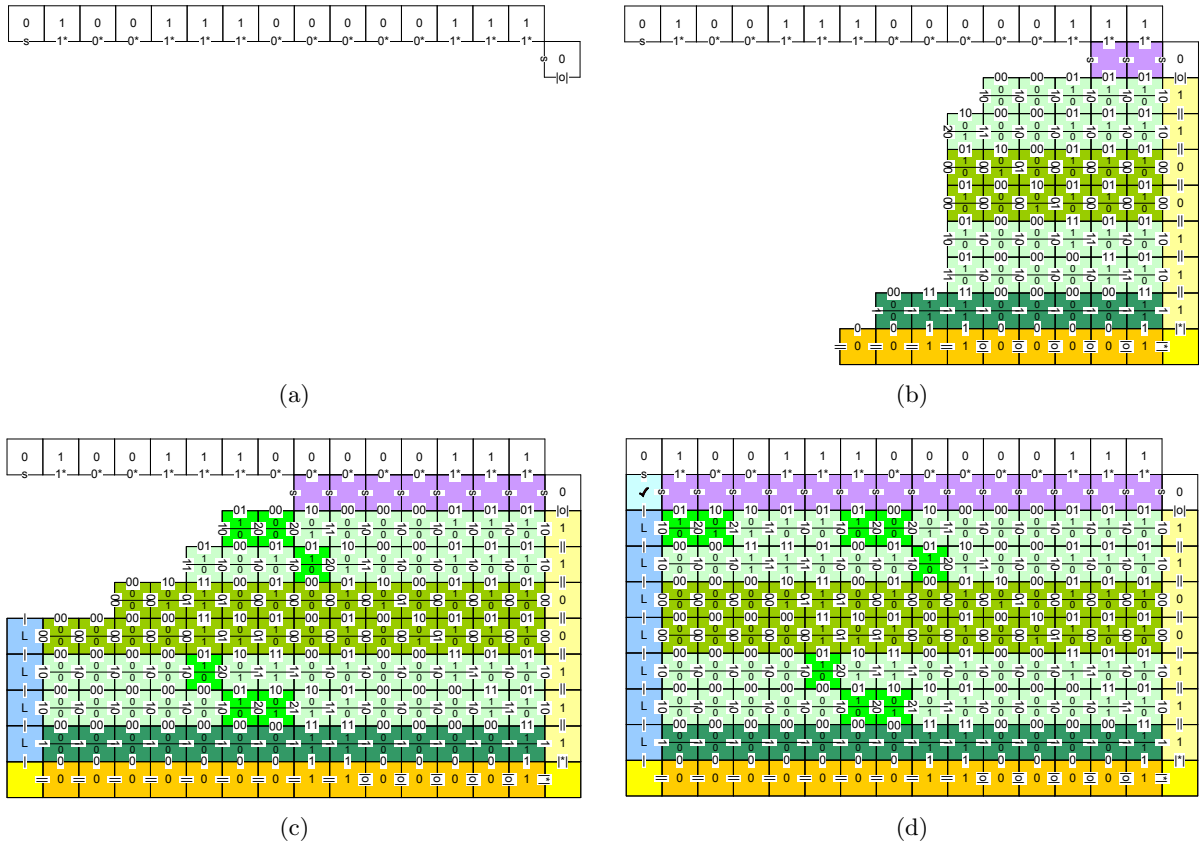


Figure 4.16: $\mathbb{S}_4$ factors a number ζ encoded in the seed configuration, e.g., $\zeta = 9991 = 10011100000111_2$ (a). In a sequence of possible attachments alternate to that in Figure 4.15, some tiles from all 3 sets T_{factors}, $T_\times$, and $T_\checkmark$ attach without waiting for all the attachments from the others sets to complete (b) and (c). Each set's tiles have designated areas of attachment and cannot attach outside of those areas, so they do not interfere with each other. Finally, for each choice of $\alpha, \beta \geq 2$ and $z \geq 0$, all sequences of attachments result in the unique final configuration for those α, β, and z, containing a special ✓ tile iff $\alpha\beta = \zeta$ and $z = n_\zeta - n_\alpha$ (d).

encoded along column 1 and row $-n_\beta$. Thus the answer encoding functions depend on the size of the final configuration. However, it would be simple to modify the factoring system slightly, leaving all the same functionality, but also copying α and β to fixed positions, say in row 0. This modification would make the system fit the definitions perfectly. I will not bother to formally define such a system here.

While it is possible for tiles to attach in exactly the order shown in Figure 4.15, other orders of attachments are also possible. For example, some tiles in $T_\times$ may attach before some other tiles in T_{factors} do, but they do not interfere with each other because they attach in disjoint sets of positions. Figure 4.16 shows an alternate possible progression of tiles attaching to the seed configuration to produce a final configuration that $\mathbb{S}_4$ produces on S_4.

I have shown the details of the final configuration that finds factors α and β of ζ. It is also interesting to observe what happens when the configuration encodes an α and β whose product does not equal ζ or when α is padded with the wrong number of 0-tiles. Figure 4.17(a) demonstrates an attempted assembly on choices of $\alpha = 91 = 1011011_2$ and $\beta = 84 = 1010100_2$. The multiplication finds the product $\alpha\beta = 7644 = 1110111011100_2$ and because $7644 \neq \zeta$, the tiles along row 1 do not

attach and thus the ✓ tile cannot attach. Figure 4.17(b) encodes the correct choices of α and β; however, it does not pad the α with enough 0-tiles. The multiplication cannot complete, and the tiles along the west column do not attach and thus the ✓ tile cannot attach.

I now examine the assembly time of $\mathbb{S}_4$ and the fraction of nondeterministic assemblies that will produce the factors (or the probability of finding the factors).

Lemma 4.2.11 (Factoring assembly time lemma) *For all $\alpha, \beta, \zeta \geq 2$ such that $\alpha\beta = \zeta$, the assembly time for $\mathbb{S}_4$ to produce a final configuration F that encodes α and β is $\Theta(n_\zeta)$.*

Proof: By the factors assembly time lemma (Lemma 4.2.2), the encoding of α and β will take $\Theta(n_\alpha + n_\beta) = \Theta(n_\zeta)$ steps. By the multiplication assembly time lemma (Lemma 4.2.7), multiplying α and β will take $\Theta(n_\alpha + n_\beta) = \Theta(n_\zeta)$ steps. By the checking assembly time lemma (Lemma 4.2.9), checking if $\alpha\beta = \zeta$ will take $\Theta(n_\zeta)$ steps. When working together, these systems do not affect each other's speed (though they may work in parallel), so the assembly time for $\mathbb{S}_4$ to produce a final configuration F that encodes α and β is $\Theta(n_\zeta)$. ■

Lemma 4.2.12 (Probability of assembly lemma) *For all $\zeta \geq 2$, given the seed S_4 encoding ζ, assuming each tile that may attach to a configuration at a certain position attaches there with a uniform probability distribution, the probability that a single nondeterministic execution of $\mathbb{S}_4$ finds $\alpha, \beta \geq 2$ such that $\alpha\beta = \zeta$ is at least $\left(\frac{1}{6}\right)^{n_\zeta}$*

Proof: In the worst case, a composite number ζ has only 2 prime factors (counting multiplicity). Either of the 2 factors could be α and the other β, but assume $\alpha \geq \beta$ (by forcing the larger factor to be α, I underestimate the probability). By Corollary 4.2.4, the probability p of assembling a particular configuration encoding α and β and padding α with $n_\zeta - n_\alpha$ 0-tiles is at least $\left(\frac{1}{6}\right)^{n_\zeta}$, as long as $\zeta = \alpha\beta$. ■

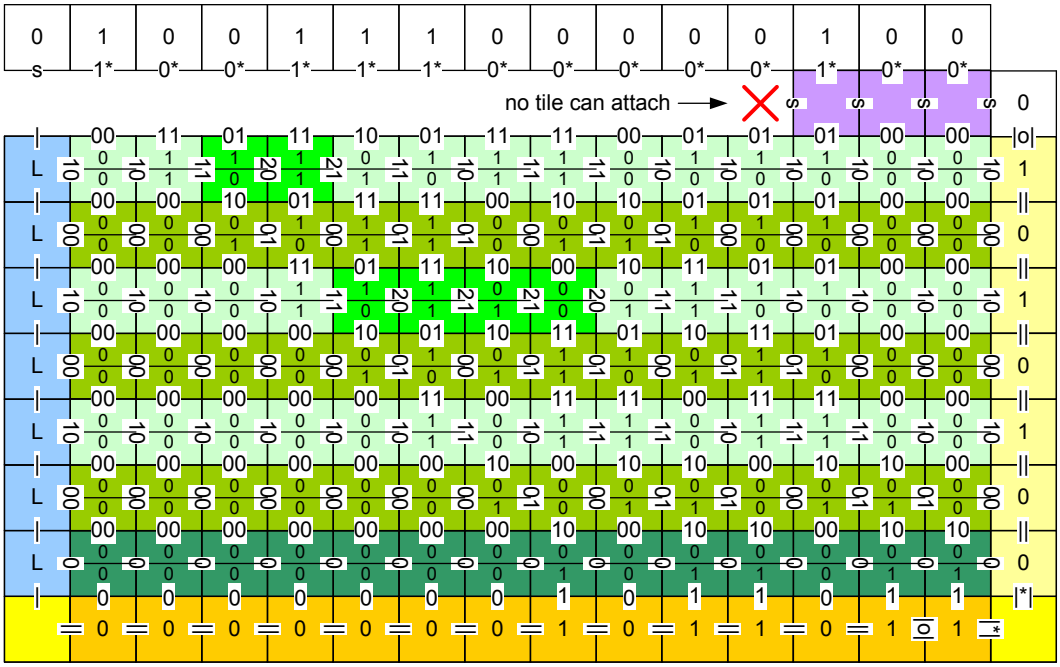
The implication of the probability of assembly lemma (Lemma 4.2.12) is that a parallel implementation of $\mathbb{S}_4$, such as a DNA implementation like those in [12, 88], with 6^{n_ζ} seeds has at least a $1 - \frac{1}{e} \geq 0.5$ chance of finding ζ's factors and one with $100(6^{n_\zeta})$ seeds has at least a $1 - \left(\frac{1}{e}\right)^{100}$ chance.

The lemma provides a lower bound on the probability of a successful assembly achievable by a distribution of tile attachment probabilities. I can derive an upper bound on that probability, for large ζ, by arguing that as α and β become large, the dominant majority of the tiles in T_{factors} that must attach to encode α are f_{a0lef} and f_{a1lef} and to encode β are f_b0 and f_b1. Because the distribution of 0 and 1 bits in a random number is even, and at least one of α and β must become large as ζ becomes large, for all probability distributions of tile attachments, the probability that a single nondeterministic execution of $\mathbb{S}_4$ finds $\alpha, \beta \geq 2$ such that $\alpha\beta = \zeta$ is at most $\left(\frac{1}{2}\right)^{n_\zeta}$. If ζ is selected as a product of two numbers with their bits chosen uniformly and independently, the best probability of guessing those number one can hope for is $\left(\frac{1}{2}\right)^{n_\zeta}$.

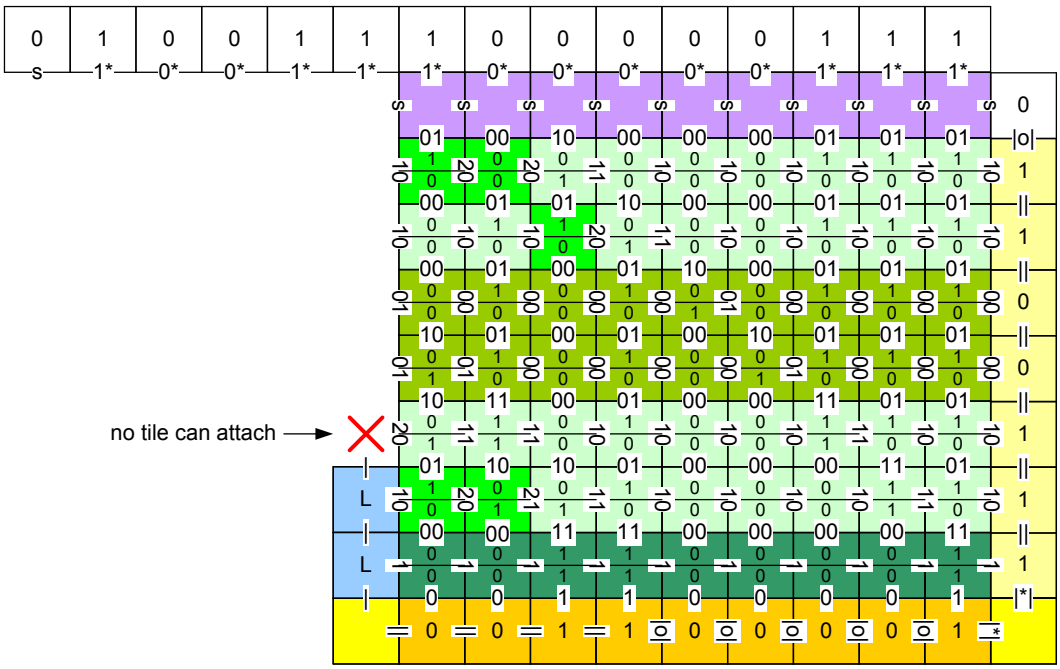
Note that T_4 contains 50 distinct tiles.

4.2.5 Factoring at Temperature Three

I have described how $\mathbb{S}_4$ factors at temperature four. It was simpler to explain how $\mathbb{S}_4$ works at temperature four, mostly because I could double the strength of every binding domain and the temperature of $\mathbb{S}_\times$ from Section 4.1; however, it is actually possible to use roughly the same tiles to compute at temperature three, by altering the strength function. The key observation is that no tile needs all four of its neighbors in a configuration in order to attach. The most a tile needs is three, e.g., the ✓ tile.



(a)



(b)

Figure 4.17: If $\alpha\beta \neq \zeta$ or the number of 0-tiles padding α is incorrect, the ✓ tile cannot attach. In (a), the product of $91 = 1011011_2$ and $84 = 1010100_2$ does not equal $9988 = 10011100000100_2$, so the tiles along the 0^{th} row do not attach. In (b), α is padded with too few 0-tiles and the tiles in the west column do not attach. Note that both these configurations are final configurations.

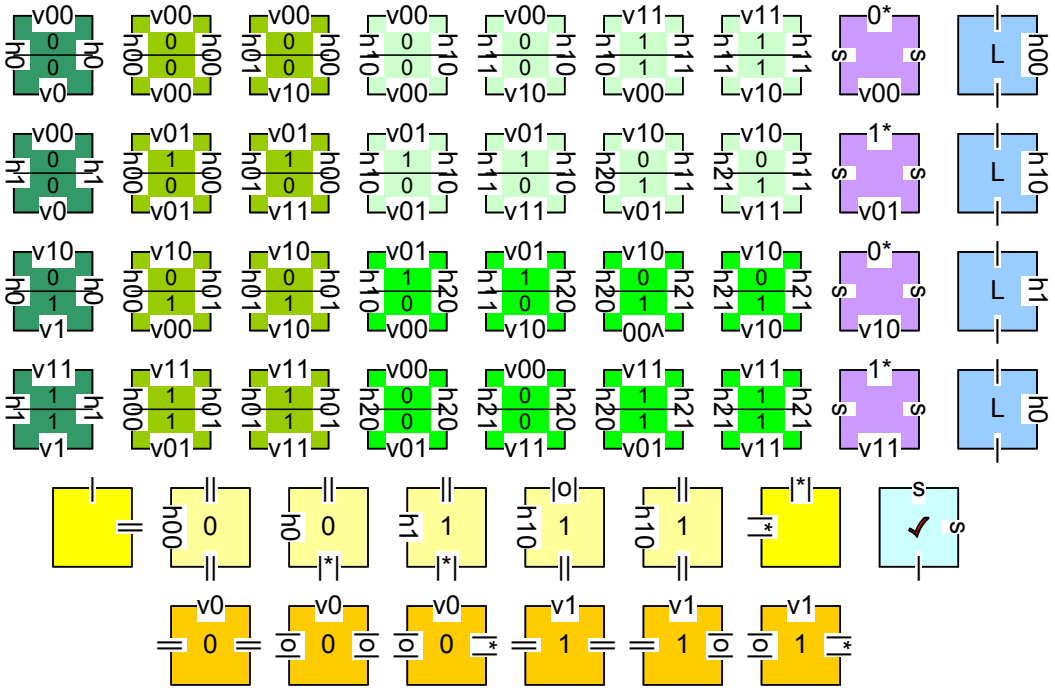


Figure 4.18: In order to factor at temperature three, some of the tiles from T_4 had to be modified with new binding domains. The new tiles form the set T_3.

I will need to differentiate some of the east-west binding domains from their previously identical north-south binding domain counterparts. That is, suppose that some of the binding domains of the tiles in T_4 were prefixed by v (for vertical) or h (for horizontal). Formally, let $\Sigma_3 = \{|o|, ||, |^*|, |, s, v0, v1, v00, v01, v10, v11, 0^*, 1^*, h0, h1, h00, h01, h10, h11, h20, h21\}$. The strengths of the binding domains need to be defined such that each tile attaches only under the same conditions under which it attached in $\mathbb{S}_4$. The tiles in T_{factors} must attach with only 1 neighbor present, so for all $\sigma \in \{|o|, ||, |^\star|\}$, let $g_3(\sigma, \sigma) = 3$. The tiles in $T_\times$ must attach when 2 of their neighbors, the south and the east neighbors, are present so for all $\sigma \in \{h0, h1, h00, h01, h10, h11, h20, h21\}$, let $g_3(\sigma, \sigma) = 2$ and for all $\sigma' \in \{v0, v1, v00, v01, v10, v11\}$, let $g_3(\sigma', \sigma') = 1$. (Note that the alternative — making the horizontal binding domains have strength 1 and vertical strength 2 — would work for the tiles in $T_\times$, but would not work for the tiles in $T_\checkmark$.) The tiles L_0, L_1, L_{00}, and L_{10} must attach when 2 of their neighbors are present, and since their east binding domains have already been defined to be of strength 2, let $g_3(|, |) = 1$. Finally, the other tiles in $T_\checkmark$ only attach when 3 of their neighbors are present, and all their binding domains so far have been defined to be of strength 1, so for all $\sigma \in \{0^\star, 1^\star, s\}$, let $g_3(\sigma, \sigma) = 1$. Figure 4.18 shows the tiles with the new binding domains labeled.

Theorem 4.2.13 (Temperature three factoring theorem) *Let T_3 be defined by Figure 4.18. Let $\tau_3 = 3$. Then $\langle T_3, g_3, \tau_3 \rangle$ is a factoring tile system.*

Proof: Let $\alpha, \beta, \zeta \geq 2$, and let $S3_0$ be the seed that encodes ζ as in Theorem 4.2.10. Then $\zeta = \alpha\beta$ iff $\mathbb{S}_4$ produces some final configuration F on $S3_0$ that encodes α and β and contains the identifier tile ✓ and all final configurations that contain ✓ encode factors of ζ.

Let W be a sequence of attachments in $\mathbb{S}_4$ that produces F on $S3_0$. $W = \langle\langle t_0, (x_0, y_0)\rangle, \langle t_1, (x_1, y_1)\rangle, \cdots, \langle t_k, (x_k, y_k)\rangle\rangle$ such that t_0 attaches at position (x_0, y_0) to $S3_0$ to produce $S3_1$, t_1

attaches at position (x_1, y_1) to $S3_1$ to produce $S3_2$, and so on to produce the final configuration $S3_{k+1} = F$.

Remember that g_3 is designed such that every tile in T_3 attaches in $\mathbb{S}_3$, under and only under the same conditions as its counterpart tile in T_4 attached in $\mathbb{S}_4$. Thus the sequence of attachments W is also a valid sequence of attachments for $\mathbb{S}_3$. Therefore, on seed $S3_0$, encoding ζ, for all final configuration F produced by $\mathbb{S}_3$ on $S3_0$, F contains ✓ iff F encodes α and β such that $\zeta = \alpha\beta$. ■

Lemma 4.2.14 *For all $\alpha, \beta, \zeta \geq 2$ such that $\alpha\beta = \zeta$, the assembly time for $\mathbb{S}_4$ to produce a final configuration F that encodes α and β is $\Theta(n_\zeta)$.*

Lemma 4.2.15 *For all $\zeta \geq 2$, given the seed S encoding ζ, assuming uniform distributions of all tiles, the probability that a single nondeterministic execution of $\mathbb{S}_3$ finds $\alpha, \beta \geq 2$ such that $\alpha\beta = \zeta$ is at least $\left(\frac{1}{6}\right)^{n_\zeta}$*

Lemmas 4.2.14 and 4.2.15 follow from the factoring assembly time lemma (Lemma 4.2.11) and the probability of assembly lemma (Lemma 4.2.12), respectively, and the fact that $\mathbb{S}_3$ follows the exact same logic as $\mathbb{S}_4$.

At the end of Section 4.3, I will discuss a possible approach to factoring at temperature 2.

As with $\mathbb{S}_4$, $\mathbb{S}_3$ uses 50 tile types.

4.3 Solving *SubsetSum* with Tiles

In this section, I will examine a tile system that decides an NP-complete set called *SubsetSum*. This system appears in my paper [27].

SubsetSum is a well known NP-complete problem. The set *SubsetSum* is a set of pairs: a finite sequence $\vec{B} = \langle B_1, B_2, \cdots, B_n \rangle \in \mathbb{N}^n$, and a target number $v \in \mathbb{N}$, such that $\langle \vec{B}, v \rangle \in$ *SubsetSum* iff $\exists \vec{c} = \langle c_1, c_2, \cdots, c_n \rangle \in \{0,1\}^n$ such that $\sum_{i=1}^{n} c_i B_i = v$. In other words, the sum of some subset of numbers of $\vec{B}$ equals exactly v.

In order to explain the system that nondeterministically decides *SubsetSum*, I will first define three smaller systems that perform pieces of the necessary computation. The first system subtracts numbers, and given the right conditions, will subtract a B_i from v. The second system computes the identity function and just copies information (this system will be used when a B_i should not be subtracted from v). The third system nondeterministically guesses whether the next B_i should or should not be subtracted. Finally, I will add a few other tiles that ensure that the computations went as planned and attach an identifier tile if the execution found that $\langle \vec{B}, v \rangle \in$ *SubsetSum*. The system works by nondeterministically choosing a subset of $\vec{B}$ to subtract from v and comparing the result to 0.

I remind the reader of a couple of corollaries I will use in this chapter. In some proofs, I will refer to the unique final configuration corollary (Corollary 4.1.3), which states that if all the tiles in a system have unique east-south binding domain pairs, then on some seed configurations, that system always produces a unique final configuration, and to the assembly time corollary (Corollary 4.1.5), which states that the final configuration produced by such systems assembles in time linear in the largest dimension of the seed.

Whenever considering a number $\alpha \in \mathbb{N}$, I will refer to the size of α, in bits, as n_α. I will further refer to the i^{th} bit of α as α_i; that is, for all $i \in \mathbb{N}$, $\alpha_i \in \{0,1\}$ such that $\sum_i \alpha_i 2^i = \alpha$. The least significant bit of α is α_0.

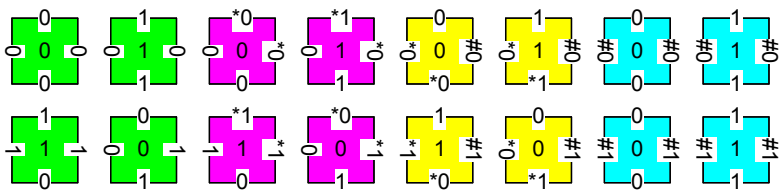


Figure 4.19: There are 16 tiles in T_-. The value in the middle of each tile t represents that tile's $v(t)$ value.

4.3.1 Subtraction

In this section, I will describe a system that subtracts positive integers. It is similar to one of the addition systems from Section 4.1, contains 16 tiles, and will subtract one bit per row of computation.

Figure 4.19 shows the 16 tiles of T_-. The value in the middle of each tile t represents that tile's $v(t)$ value. Intuitively, the system will subtract the i^{th} bit on the i^{th} row. The tiles to the right of the i^{th} location will be blue; the tile in the i^{th} location will be yellow; the next tile, the one in the $(i+1)^{\text{st}}$ location, will be magenta; and the rest of the tiles will be green. The purpose of the yellow and magenta tiles is to compute the diagonal line, marking the i^{th} position on the i^{th} row. Figure 4.20 shows two sample executions of the subtracting system.

Lemma 4.3.1 *Let $\Sigma_- = \{0, 1, {}^\star 0, {}^\star 1, \#0, \#1\}$, let T_- be the set of tiles defined by Figure 4.19, let $g_- = 1$, let $\tau_- = 2$, and let $\mathbb{S}_- = \langle T_-, g_-, \tau_- \rangle$. Let $\alpha, \beta \in \mathbb{N}$ and let $\delta = \alpha - \beta$. Let S_- be a seed configuration such that there exists some $(x_0, y_0) \in \mathbb{Z}^2$ such that:*

- *$bd_N(S_-(x_0 - 1, y_0)) = {}^\star\alpha_0$.*
- *For all $i \in \{1, 2, \cdots, n_\alpha - 1\}$, $bd_N(S_-(x_0 - i - 1, y_0)) = \alpha_i$.*
- *For all $j \in \{0, 1, \cdots, n_\beta - 1\}$, $bd_W(S_-(x_0, y_0 + j + 1)) = \#\beta_j$.*
- *For all other positions (x, y), $(x, y) \notin S_-$.*

Then $\mathbb{S}_-$ produces a unique final configuration F_- on S_- such that
$\alpha \geq \beta \implies$

- *For all $i \in \{0, 1, \cdots, n_\alpha - 1\}$, $bd_N(F_-(x_0 - i - 1, y_0 + n_\beta)) \in \{\delta_i, {}^\star\delta_i\}$.*
- *For all $j \in \{0, 1, \cdots, n_\beta - 1\}$, $bd_W(F_-(x_0 - n_\alpha, y_0 + j + 1)) \in \{0, {}^\star 0\}$.*

and $\alpha < \beta \implies$

- *There exists $j \in \{0, 1, \cdots, n_\beta - 1\}$ such that $bd_W(F_-(x_0 - n_\alpha, y_0 + j + 1)) \in \{1, {}^\star 1\}$.*

Proof: By the unique final configuration corollary (Corollary 4.1.3), $\mathbb{S}_-$ produces a unique final configuration on S_-. Call that configuration F_-.

To simplify notation, given x_0, y_0, for all $i, j \in \mathbb{Z}$, let $p(i, j) = (x_0 - i - 1, y_0 + j + 1)$. The purpose of this notation is to more easily identify data in the construction. I expect the north binding domain of the tile in position $p(i, j)$ to code for the i^{th} bit of ${}_j\delta$ (see definition of ${}_j\delta$ below), also denoted ${}_j\delta_i$. Further, I will refer to row $y_0 + j + 1$ as row $p(\cdots, j)$, and column $x_0 - i - 1$ as column $p(i, \cdots)$. Thus, it is sufficient to show F_- is such that:

$\alpha \geq \beta \implies$

- For all $i \in \{0, 1, \cdots, n_\alpha - 1\}$, $bd_N(F_-(p(i, n_\beta - 1))) \in \{\delta_i, {}^\star\delta_i\}$.
- For all $j \in \{0, 1, \cdots, n_\beta - 1\}$, $bd_W(F_-(p(n_\alpha - 1, j))) \in \{0, {}^\star 0\}$.

and $\alpha < \beta \implies$

- There exists $j \in \{0, 1, \cdots, n_\beta - 1\}$ such that $bd_W(F_-(p(n_\alpha - 1, j))) \in \{1, {}^\star 1\}$.

For all $j \in \{-1, 0, 1, \cdots, n_\beta - 1\}$, let ${}_j\delta = \alpha - \sum_{k=0}^{j} \beta_k 2^k$. (That is, ${}_j\delta$ is the difference between α and the number formed by the $j+1$ least significant bits of β.) I show, by induction on j, that for all $i \in \{0, 1, \cdots, j, j+2, \cdots, n_\alpha - 1\}$, $bd_N(F_-(p(i, j))) = {}_j\delta_i$ and $bd_N(F_-(p(j+1, j))) = {}^\star{}_j\delta_{j+1}$.

First, note some properties of ${}_j\delta$: ${}_{-1}\delta = \alpha$, ${}_{n_\beta - 1}\delta = \alpha - \beta = \delta$, and for all $i \in \{0, 1, \cdots, j\}$, ${}_j\delta_i = {}_{j+1}\delta_i$.

Base case: ($j = -1$). Thus ${}_j\delta = {}_{-1}\delta = \alpha$. S_- and F_- must agree everywhere S_- is not *empty*, and by definition of S_-, $bd_N(S_-(p(0, -1))) = {}^\star\alpha_0$ and for all $i \in \{1, 2, \cdots, n_\alpha - 1\}$, $bd_N(S_(- p(i, -1))) = \alpha_i$. Thus, for $j = -1$, for all $i \in \{1, 2, \cdots, n_\alpha - 1\}$, $bd_N(F_-(p(i, j))) = {}_j\delta_i$ and $bd_N(F_-(p(j+1, j))) = {}^\star{}_j\delta_{j+1}$.

Inductive step: I assume that for all $i \in \{0, 1, \cdots, j, j+2, \cdots, n_\alpha - 1\}$, $bd_N(F_-(p(i, j))) = {}_j\delta_i$ and $bd_N(F_-(j+1, j)) = {}^\star{}_j\delta_{j+1}$. I will show that for all $i \in \{0, 1, \cdots, j+1, j+3, \cdots, n_\alpha - 1\}$, $bd_N(F_-(p(i, j+1))) = {}_{j+1}\delta_i$ and $bd_N(F_-(p(j+2, j+1))) = {}^\star{}_{j+1}\delta_{j+2}$.

Consider row $p(\ldots, j+1)$. Consider the tile t that attaches in position $p(0, j+1)$ (assume for a second that $j \neq 0$). By the definition of S_-, $bd_W(F_-(p(-1, j+1))) = \#\beta_{j+1}$ and by the inductive hypothesis, $bd_N(F_-(p(0, j))) = {}_j\delta_0$. Thus t's east binding domain must be either #0 or #1 and south binding domain must be either 0 or 1, so t must be a blue tile. Therefore, $bd_W(t) = bd_E(t) = \#\beta_{j+1}$ and $bd_N(t) = bd_S(t) = {}_j\delta_0$. Note that this argument holds for the tile to the west of position $p(0, j+1)$, as long as the north binding domain of the tile below does not start with *, and so on. By the inductive hypothesis, for all $i \in \{0, 1, \cdots, j\}$, $bd_N(F_-(p(i, j))) = {}_j\delta_i$ and thus those binding domains do not start with *. Thus, for all $i \in \{0, 1, \cdots, j\}$, $bd_W(F_-(p(i, j+1))) = \#\beta_{j+1}$ and $bd_N(F_-(p(i, j+1))) = {}_j\delta_i = {}_{j+1}\delta_i$. (Note that for $j = 0$, no blue tiles attach so the earlier assumption is justified. For $j = 0$ the argument starts here.)

Observe that for all non-blue tiles (yellow, magenta, and green), looking only at the numerical value of the binding domains (ignoring the preceding * and #) the north binding domain is the result of subtracting the east binding domain from the south binding domain, modulo 2, and the west binding domain is 1 iff the east binding domain is greater than the south binding domain.

Consider the tile t that attaches in position $p(j+1, j+1)$. I just showed that $bd_E(t)$ must be $\#\beta_{j+1}$. By the inductive hypothesis, $bd_N(F_-(p(j+1, j))) = {}^\star{}_j\delta_{j+1}$, so t must be yellow. From above, a yellow tile's north binding domain is the difference of its south and east binding domains, and its west binding domain is 1 iff the east binding domain is greater than the south binding domain. Thus $bd_N(t)$ is ${}_j\delta_{j+1} - \beta_{j+1} = {}_{j+1}\delta_{j+1}$ and $bd_W(t) = {}^\star 1$ if ${}_j\delta_{j+1} < \beta_{j+1}$, thus "borrowing a 1" and $bd_W(t) = {}^\star 0$ otherwise.

Consider the tile t that attaches in position $p(j+2, j+1)$. I just showed that $bd_E(t)$ must be ${}^\star 0$ if there is no need to borrow a 1 and ${}^\star 1$ otherwise. By the inductive hypothesis, $bd_N(F_-(p(j+2, j))) = {}_j\delta_{j+2}$, so t must be magenta. Thus $bd_N(t)$ is ${}^\star{}_{j+1}\delta_{j+2}$ and $bd_W(t) = 1$ if there is a need to borrow a 1 again, and $bd_W(t) = 0$ otherwise.

Consider the tile t that attaches in position $p(j+3, j+1)$. I just showed that $bd_E(t)$ must be 0 if there is no need to borrow a 1 and 1 otherwise. By the inductive hypothesis, $bd_N(F_-(p(j+3, j))) = {}_j\delta_{j+3}$, so t must be green. Thus $bd_N(t)$ is ${}_{j+1}\delta_{j+3}$ and $bd_W(t) = 1$ if there is a need to borrow a 1 again, and $bd_W(t) = 0$ otherwise. The same holds for the tile to west of position $p(j+3, j+1)$, and

so on, until position $p(n_\alpha - 1, j+1)$, thus green tiles attach. Also, $bd_W(F_-(p(n_\alpha - 1, j+1))) = 1$ if the number being subtracted exceeds ${}_{j+1}\delta$ and $bd_W(F_-(p(n_\alpha - 1, j+1))) = 0$ otherwise.

Thus, in row $p(\cdots, j+1)$, for all $i \in \{0, 1, \cdots, j+1, j+3, \cdots, n_\alpha - 1\}$, $bd_N(F_-(p(i, j+1))) = {}_{j+1}\delta_i$ and $bd_N(F_-(p(j+2, j+1))) = {}^\star{}_{j+1}\delta_{j+2}$. Also, $bd_W(F_-(p(n_\alpha - 1, j+1))) \in \{1, {}^\star 1\}$ iff the number being subtracted exceeds ${}_{j+1}\delta$.

Therefore, $\alpha \geq \beta \implies$

- Let $j = n_\beta - 1$. Since ${}_{n_\beta - 1}\delta = \delta$, in row $p(\cdots, n_\beta - 1)$, for all $i \in \{0, 1, \cdots, n_\alpha - 1\}$, $bd_N(F_-(p(i, n_\beta - 1))) \in \{\delta_i, {}^\star\delta_i\}$.
- For all $j \in \{0, 1, \cdots, n_\beta - 1\}$, $bd_W(F_-(p(n_\alpha - 1, j))) \in \{0, {}^\star 0\}$.

and, $\alpha < \beta \implies$

- There exists $j \in \{0, 1, \cdots, n_\beta - 1\}$ such that $bd_W(F_-(p(n_\alpha - 1, j))) \in \{1, {}^\star 1\}$.

■

Thus $\mathbb{S}_-$ is a system that is capable of subtracting numbers. Formally, using the definition of a tile assembly model deterministically computing a function:

Theorem 4.3.2 *Let $f\colon \mathbb{N}^2 \to \mathbb{N}$ be such that for all $\alpha, \beta \in \mathbb{N}$ such that $\alpha \geq \beta$, $f(\alpha, \beta) = \alpha - \beta$ and for all other α, β, $f(\alpha, \beta)$ is undefined. Then $\mathbb{S}_-$ computes the function f.*

Proof: This theorem follows from Lemma 4.3.1, and the fact that for all $t \in T_-$, $v(t) = bd_N(t)$. ■

Lemma 4.3.3 *The assembly time of $\mathbb{S}_-$ is $\Theta(n)$ steps to subtract an n-bit number from another n-bit number.*

Proof: The lemma follows directly from the assembly time corollary (Corollary 4.1.5). ■

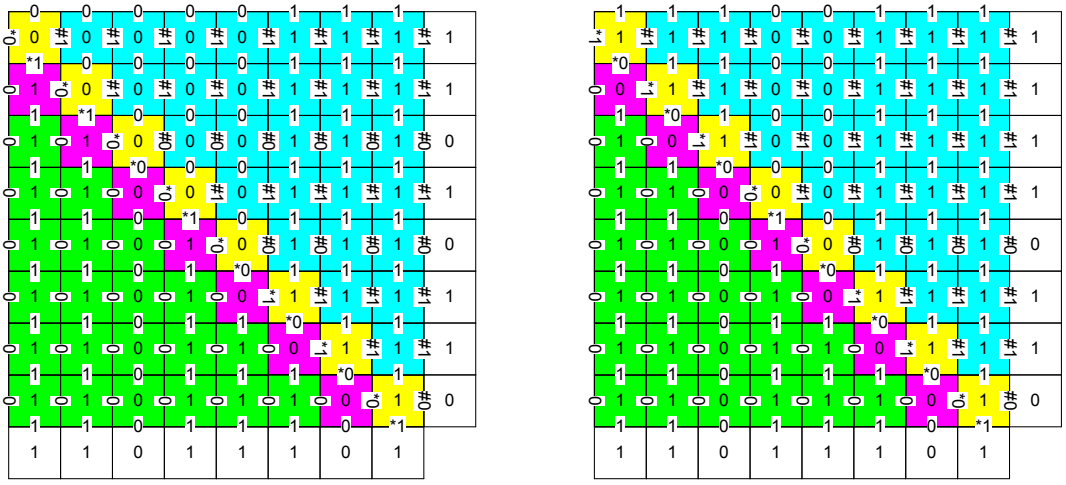
Figure 4.20 shows sample executions of $\mathbb{S}_-$. In Figure 4.20(a), the system subtracts $214 = 11010110_2$ from $221 = 11011101_2$ to get $7 = 111_2$. The inputs are encoded along the bottom row ($221 = 11011101_2$) and rightmost column ($214 = 11010110_2$). The output is on the top row ($7 = 00000111_2$). Note that because $214 \leq 221$, all the west binding domains of the leftmost column contain a 0. In Figure 4.20(b), the system attempts to subtract $246 = 11110110$ from $221 = 11011101_2$, but because $246 > 221$, the computation fails, and indicates its failure because the topmost leftmost west binding domain contains a 1.

This system is very similar to an adding system from Section 4.1.2.3, but not the smallest adding system from Section 4.1. While this system has 16 tiles, it is possible to design a subtracting system with 8 tiles, that is similar to the 8-tile adding system from Section 4.1.2.1.

4.3.2 Identity

I now describe a system that ignores the input on the rightmost column, and simply copies upwards the input from the bottom row. This is a fairly straight-forward system that will not need much explanation.

Lemma 4.3.4 *Let $\Sigma_{\mathrm{x}} = \{\mathsf{x}0, \mathsf{x}1, \#0, \#1\}$, let T_{x} be the set of tiles defined by Figure 4.21, let $g_{\mathrm{x}} = 1$, let $\tau_{\mathrm{x}} = 2$, and let $\mathbb{S}_{\mathrm{x}} = \langle T_{\mathrm{x}}, g_{\mathrm{x}}, \tau_{\mathrm{x}} \rangle$. Let $\alpha, \beta \in \mathbb{N}$. Let S_{x} be a seed configuration such that there exists some $(x_0, y_0) \in \mathbb{Z}^2$ such that:*



(a) (b)

Figure 4.20: An example of $\mathbb{S}_-$ subtracting numbers. In (a), the system subtracts $214 = 11010110_2$ from $221 = 11011101_2$ to get $7 = 111_2$. The inputs are encoded along the bottom row ($221 = 11011101_2$) and rightmost column ($214 = 11010110_2$). The output is on the top row ($7 = 00000111_2$). Note that because $214 \leq 221$, all the west binding domains of the leftmost column contain a 0. In (b), the system attempts to subtract $246 = 11110110$ from $221 = 11011101_2$, but because $246 > 221$, the computation fails, and indicates its failure with the top- and leftmost west binding domain containing a 1.

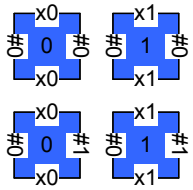


Figure 4.21: There are 4 tiles in T_{x}. The value in the middle of each tile t represents that tile's $v(t)$ value.

- *For all* $i \in \{0, 1, \cdots, n_\alpha - 1\}$, $bd_N(S_{\text{x}}(x_0 - i - 1, y_0)) = \text{x}\alpha_i$.
- *For all* $j \in \{0, 1, \cdots, n_\beta - 1\}$, $bd_W(S_{\text{x}}(x_0, y_0 + j + 1)) = \#\beta_j$.
- *For all other positions* (x, y), $(x, y) \notin S_{\text{x}}$.

Then $\mathbb{S}_{\text{x}}$ *produces a unique final configuration* F_{x} *on* S_{x}, *and for all* $i \in \{0, 1, \cdots, n_\alpha - 1\}$, $bd_N(F_{\text{x}}(x_0 - i - 1, y_0 + n_\beta)) = \text{x}\alpha_i$ *and for all* $j \in \{0, 1, \cdots, n_\beta - 1\}$, $bd_W(F_-(x_0 - n_\alpha, y_0 + j + 1)) = \text{x}0$.

Proof: By the unique final configuration corollary (Corollary 4.1.3), $\mathbb{S}_{\text{x}}$ produces a unique final configuration on S_{x}. Call that configuration F_{x}. It is clear that the configuration will fill the rectangle outlined by the seed, with the exception of the bottom right corner, because for every possible pair of west-north binding domains of the tiles in T_{x} and in S_{x}, there is a tile with a matching east-south binding domain.

Every tile $t \in T_{\text{x}}$ has $bd_S(t) = bd_N(t)$ so for all $i \in \{0, 1, \cdots, n_\alpha - 1\}$, $bd_N(F_{\text{x}}(x_0 - i - 1, y_0 + n_\beta)) = \text{x}\alpha_i$. Every tile $t \in T_{\text{x}}$ has $bd_W(t) = \text{x}0$ so for all $j \in \{0, 1, \cdots, n_\beta - 1\}$, $bd_W(F_-(x_0 - n_\alpha, y_0 + j + 1)) = \text{x}0$. ■

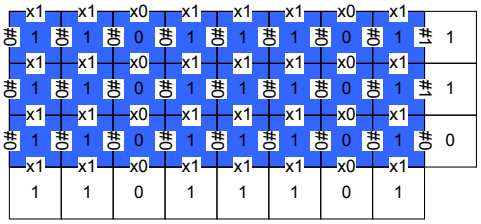


Figure 4.22: An example of an $\mathbb{S}_{\text{x}}$ execution. The system simply copies the input on the bottom row upwards, to the top column.

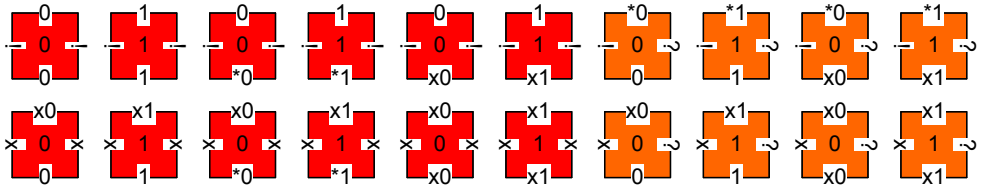


Figure 4.23: The are 20 tiles in $T_?$. The value in the middle of each tile t represents that tile's $v(t)$ value. Unlike the red tiles, the orange tiles do not have unique east-south binding domain pairs, and thus will attach nondeterministically.

Lemma 4.3.5 *The assembly time of* $\mathbb{S}_{\text{x}}$ *is* $\Theta(n_\alpha + n_\beta)$.

Proof: The lemma follows directly from the assembly time corollary (Corollary 4.1.5). ■

Figure 4.22 shows a sample execution of the $\mathbb{S}_{\text{x}}$ system. The system simply copies the input on the bottom row upwards, to the top column.

4.3.3 Nondeterministic Guess

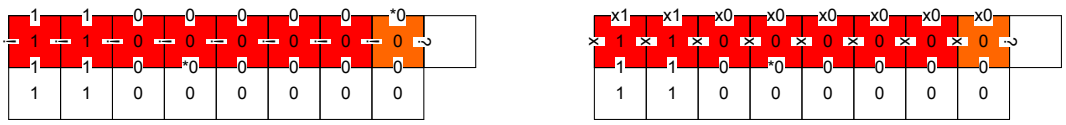
In this section, I describe a system that nondeterministically decides whether or not the next B_i should be subtracted from v. It does so by encoding the input for either the $\mathbb{S}_-$ system or the $\mathbb{S}_{\text{x}}$ system.

Lemma 4.3.6 *Let* $\Sigma_? = \{?, !, 0, 1, \text{x}0, \text{x}1, {}^\star 0, {}^\star 1\}$, *let* $T_?$ *be the set of tiles defined by Figure 4.23, let* $g_? = 1$, *let* $\tau_? = 2$, *and let* $\mathbb{S}_? = \langle T_?, g_?, \tau_? \rangle$. *Let* $\alpha \in \mathbb{N}$. *Let* $S_?$ *be a seed configuration such that there exists some* $(x_0, y_0) \in \mathbb{Z}^2$ *such that:*

- $bd_N(S_?(x_0 - 1, y_0)) \in \{\alpha_i, \text{x}\alpha_i\}$
- *For all* $i \in \{1, 2, \cdots, n_\alpha - 1\}$, $bd_N(S_?(x_0 - i - 1, y_0)) \in \{\alpha_i, {}^\star\alpha_i, \text{x}\alpha_i\}$.
- $bd_W(S_?(x_0, y_0 + 1)) = ?$.
- *For all other positions* (x, y), $(x, y) \notin S_?$.

Then $\mathbb{S}_?$ *produces one of two final configurations* $F_?$ *on* $S_?$. *Either,*

- *for all* $i \in \{0, 1, \cdots, n_\alpha - 1\}$, $bd_N(F_?(x_0 - i - 1, y_0 + 1)) = \text{x}\alpha_i$ *and* $bd_W(F_?(x_0 - n_\alpha, y_0 + 1)) = \text{x}$, *or*
- $bd_N(F_?(x_0 - 1, y_0 + 1)) = {}^\star\alpha_i$, *and for all* $i \in \{1, 2, \cdots, n_\alpha - 1\}$, $bd_N(F_?(x_0 - i - 1, y_0 + 1)) = \alpha_i$, *and* $bd_W(F_?(x_0 - n_\alpha, y_0 + 1)) = !$.



(a) (b)

Figure 4.24: Two examples of $\mathbb{S}_?$ executions. In (a), the system attaches tiles with ! east-west binding domains, preparing a valid seed for $\mathbb{S}_-$, and in (b), the system attaches tiles with x east-west binding domains, preparing a valid seed for $\mathbb{S}_\mathrm{x}$.

Proof: Because $bd_W(S_\mathrm{x}(x_0, y_0 + 1)) = ?$, only an orange tile may attach in position $(x_0 - 1, y_0 + 1)$. Suppose an orange tile t attaches such that $bd_W(t) = \mathrm{x}$. Then $bd_S(t) \in \{\alpha_0, \mathrm{x}\alpha_0\} \implies bd_N(t) = \mathrm{x}\alpha_0$. Further, to the west of that position, only red tiles with west binding domain x can attach, and for all of those, since their south binding domains $\in \{\alpha_i, {}^\star\alpha_i, \mathrm{x}\alpha_i\}$, their north binding domains must be $\mathrm{x}\alpha_i$. Thus, for all $i \in \{0, 1, \cdots, n_\alpha - 1\}$, $bd_N(F_?(x_0 - i - 1, y_0 + 1)) = \mathrm{x}\alpha_i$ and $bd_W(F_?(x_0 - n_\alpha, y_0 + 1)) = \mathrm{x}$.

Now suppose an orange tile t attaches at position $(x_0 - 1, y_0 + 1)$ such that $bd_W(t) = !$. Then $bd_S(t) \in \{\alpha_0, \mathrm{x}\alpha_0\} \implies bd_N(t) = {}^\star\alpha_0$. Further, to the west of that position, only red tiles with west binding domain ! can attach, and for all of those, since their south binding domains $\in \{\alpha_i, {}^\star\alpha_i, \mathrm{x}\alpha_i\}$, their north binding domains must be α_i. Thus, $bd_N(F_?(x_0 - 1, y_0 + 1)) = {}^\star\alpha_i$, and for all $i \in \{1, 2, \cdots, n_\alpha - 1\}$, $bd_N(F_?(x_0 - i - 1, y_0 + 1)) = \alpha_i$, and $bd_W(F_?(x_0 - n_\alpha, y_0 + 1)) = !$. ■

Lemma 4.3.7 *The assembly time of* $\mathbb{S}_?$ *is* $\Theta(n_\alpha)$.

Proof: The lemma follows directly from the assembly time corollary (Corollary 4.1.5). ■

Figure 4.24 shows two possible executions of $\mathbb{S}_?$. In Figure 4.24(a), the system attaches tiles with ! east-west binding domains, preparing a valid seed for $\mathbb{S}_-$, and in Figure 4.24(b), the system attaches tiles with x east-west binding domains, preparing a valid seed for $\mathbb{S}_\mathrm{x}$. Only one tile, the orange tile, attaches nondeterministically, determining which tiles attach to its west.

4.3.4 Deciding *SubsetSum*

I have described three systems that I will now use to design a system to decide *SubsetSum*. Intuitively, I plan to write out the elements of $\vec{B}$ on a column and v on a row, and the system will nondeterministically choose some of the elements from $\vec{B}$ to subtract from v. The system will then check to make sure that no subtracted element was larger than the number it was being subtracted from, and whether the result is 0. If both conditions are satisfied, a special identifier tile will attach to signify that $\langle \vec{B}, v \rangle \in$ *SubsetSum*.

Theorem 4.3.8 *Let* $\Sigma_{SS} = \Sigma_- \cup \Sigma_\mathrm{x} \cup \Sigma_? \cup \{|\}$. *Let* $T_{SS} = T_- \cup T_\mathrm{x} \cup T_? \cup T_\checkmark$, *where* $T_\checkmark$ *is defined by Figure 4.25. Let* $g_{SS} = 1$ *and* $\tau_{SS} = 2$. *Let* $\mathbb{S}_{SS} = \langle T_{SS}, g_{SS}, \tau_{SS} \rangle$. *Then* $\mathbb{S}_{SS}$ *nondeterministically decides SubsetSum with the black* ✓ *tile from* $T_\checkmark$ *as the identifier tile.*

Proof: Let $n \in \mathbb{N}$, let $\vec{B} = \langle B_1, B_2, \cdots, B_n \rangle \in \mathbb{N}^n$, and let $v \in \mathbb{N}$. Let Γ_{SS} be as defined by Figure 4.26. Let the seed S_{SS} be as follows (see Figure 4.27(a) for an example of a valid seed):

- For all $i \in \{0, 1, \cdots, n_v - 1\}$, $S_{SS}(-i, 0) = \gamma_{tv_i}$.
- $S_{SS}(-n_v, 0) = \gamma_{left}$.

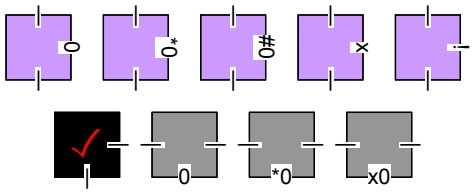


Figure 4.25: There are 9 tiles in $T_{\checkmark}$. The black tile with a ✓ in the middle will serve as the identifier tile.

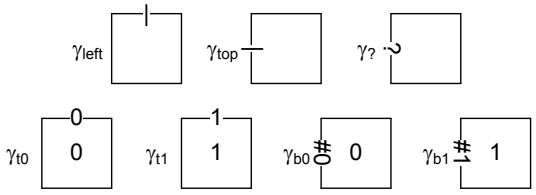


Figure 4.26: There are 7 tiles in Γ_{SS}. The value in the middle of each tile t represents that tile's $v(t)$ value and each tile's name is written on its left.

- For all $k \in \{1, 2, \cdots n\}$, $S_{SS}\left(1, 1 + \sum_{j=1}^{k-1}\left(n_{B_j} + 1\right)\right) = \gamma_?$.
- For all $k \in \{1, 2, \cdots n\}$, for all $i \in \{0, 1, \cdots, n_{B_k} - 1\}$, $S_{SS}\left(1, 2 + i + \sum_{j=1}^{k-1}\left(n_{B_j} + 1\right)\right) = \gamma_{b(B_k)_i}$.
- $S_{SS}\left(1, 1 + \sum_{j=1}^{n}\left(n_{B_j} + 1\right)\right) = \gamma_{top}$.
- And for all other positions (x, y), $(x, y) \notin S_{SS}$.

Because T_{-}, T_{x}, $T_?$, and $T_{\checkmark}$ have disjoint sets of south-east binding domain pairs, and because a tile attaches to the assembly only when its south and east binding domains match (as described in the step configuration lemma (Lemma 4.1.1)), only one of those sets contains tiles that can attach at each position.

$S_{SS}(1, 1) = \gamma_?$, so tiles from $T_?$ may attach in position $(0, 1)$. By Lemma 4.3.6, one of two things will happen on row 1: either tiles with east-west binding domains ! will attach, or tiles with east-west binding domains x will attach. Thus one nondeterministic sequence of attachments (case 1) will result in the north binding domains of the tiles in row 1 encoding the bits of v, and the other nondeterministic sequence of attachments (case 2) will result in the north binding domains of the tiles in row 1 encoding bits of v with x preceding every bit. In case 1, tiles from T_{-} will attach, and by Lemma 4.3.1, these tiles will attach deterministically to encode the bits of $v - B_1$ in the north binding domains of row $n_{B_1} + 1$. In case 2, tiles from T_{x} will attach, and by Lemma 4.3.4, these tiles will attach deterministically to encode the bits of v in the north binding domains of row $n_{B_1} + 1$.

Note that $S_{SS}\left(1, n_{B_1} + 2\right) = \gamma_?$ so the process can repeat with B_2, and so on. Thus for each nondeterministic sequence of attachments, in the final configuration, the north binding domains of the tiles in row $\sum_{j=1}^{n}\left(n_{B_j} + 1\right)$ encode $v - \sum_{j=1}^{n} c_j B_j$, where $\vec{c} = \langle c_1, c_2, \cdots, c_n\rangle \in \{0, 1\}^n$ and there exists a nondeterministic execution for each of the 2^n possible assignments of $\vec{c}$.

Only lavender tiles can attach in column $-n_v$ because their north-south binding domains are | and thus they can only attach to each other or north of γ_{left}, which only occurs in position

$(-n_v, 0)$. By Lemma 4.3.1, one of the B_j being subtracted is greater than the number it is being subtracted from iff there exists a q such that the tile t that attaches at position $(-n_v - 1, q)$ has $bd_W(t) \in \{1, {}^\star 1\}$. In column $-n_v$, lavender tiles from $T_\checkmark$ attach only to tiles with west binding domains x, !, or that contain a 0. Thus a tile attaches in position $\left(-n_v, \sum_{j=1}^{n} \left(n_{B_j} + 1\right)\right)$ iff no subtracted numbers have exceeded the number they were subtracted from. Since that tile must be lavender, its north binding domain is |.

The black ✓ tile may only attach at position $\left(-n_v, 1 + \sum_{j=1}^{n} \left(n_{B_j} + 1\right)\right)$ because that is the only position with a tile to its south that may have the | north binding domain and the tile to its east that may have the | west binding domain. The gray tiles in $T_\checkmark$ attach in row $1 + \sum_{j=1}^{n} \left(n_{B_j} + 1\right)$, starting from column 0, then 1, etc. iff all the north binding domains of row $\sum_{j=1}^{n} \left(n_{B_j} + 1\right)$ contain 0. Thus a tile can attach in position $\left(-n_v + 1, 1 + \sum_{j=1}^{n} \left(n_{B_j}\right) + 1\right)$ only if there exists a choice of $\vec{c} \in \{0,1\}^n$ such that $v - \sum_{j=1}^{n} c_j B_j = 0$.

Therefore, the black ✓ tile attaches iff there exists a choice of $\vec{c} \in \{0,1\}^n$ such that $v = \sum_{j=1}^{n} c_j B_j$, or in other words, $\langle \vec{B}, v \rangle \in \mathit{SubsetSum}$. Therefore, $\mathbb{S}_{SS}$ nondeterministically decides *SubsetSum*. ■

Figure 4.27 shows an example execution of $\mathbb{S}_{SS}$. Figure 4.27(a) encodes a seed configuration with $v = 75 = 1001011_2$ along the bottom row and $\vec{B} = \langle 11 = 1011_2,\ 25 = 11001_2,\ 37 = 100101 + 2,\ 39 = 100111_2 \rangle$ along the rightmost column. Tiles from T_{SS} attach to the seed configuration, nondeterministically testing all possible values of $\vec{c} \in \{0,1\}^4$. Figure 4.27(b) shows one such possible execution, the one that corresponds to $\vec{c} = \langle 1, 1, 0, 1 \rangle$. Because $11 + 25 + 39 = 75$, the ✓ tile attaches in the top left corner.

I have described configurations that code for the correct $\vec{c}$ to allow the ✓ tile to attach. It is also interesting to see what happens if improper nondeterministic choices of $\vec{c}$ are made. Figure 4.28(a) shows a final configuration in which one of the B_i values being subtracted is bigger than v. The leftmost column of tiles does not complete and the ✓ tile cannot attach. Figure 4.28(b) shows a final configuration of an execution that never tries to subtract a number that is too big, but the result does not equal 0. Thus the top row does not complete and the ✓ tile does not attach. Both these configurations are final, and no more tiles can attach.

Lemma 4.3.9 *The assembly time of $\mathbb{S}_{SS}$ is linear in the size of the input (number of bits in $\langle \vec{B}, v \rangle$).*

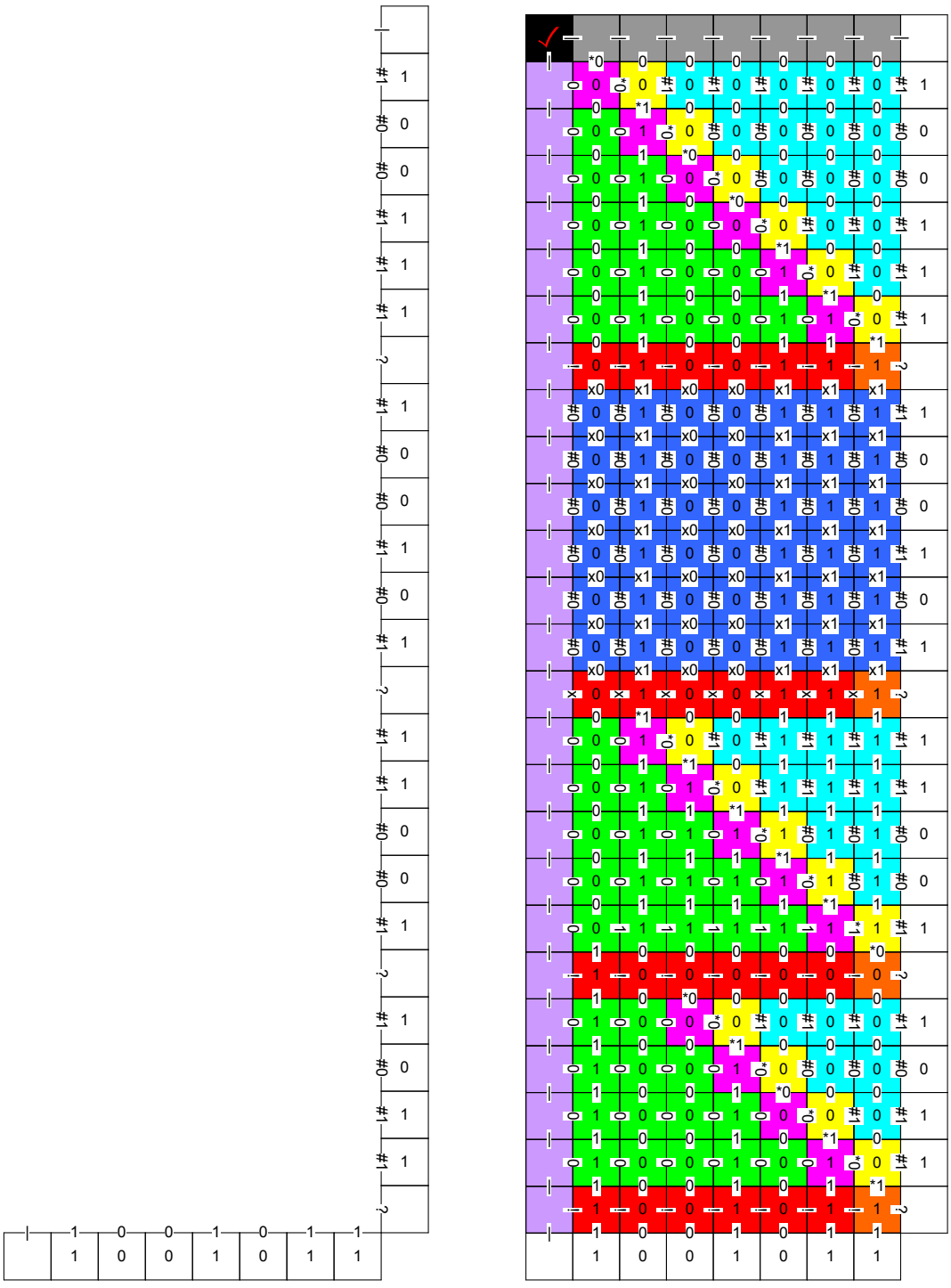
Proof: The lemma follows from Lemmas 4.3.3, 4.3.5, 4.3.7, and the facts that

- $\Theta\left(\sum_{k=1}^{n} n_{B_k}\right)$ tiles attach in column $-n_v$ and
- $\Theta(n_v)$ tiles attach on row $1 + \sum_{j=1}^{n} \left(n_{B_j} + 1\right)$.

■

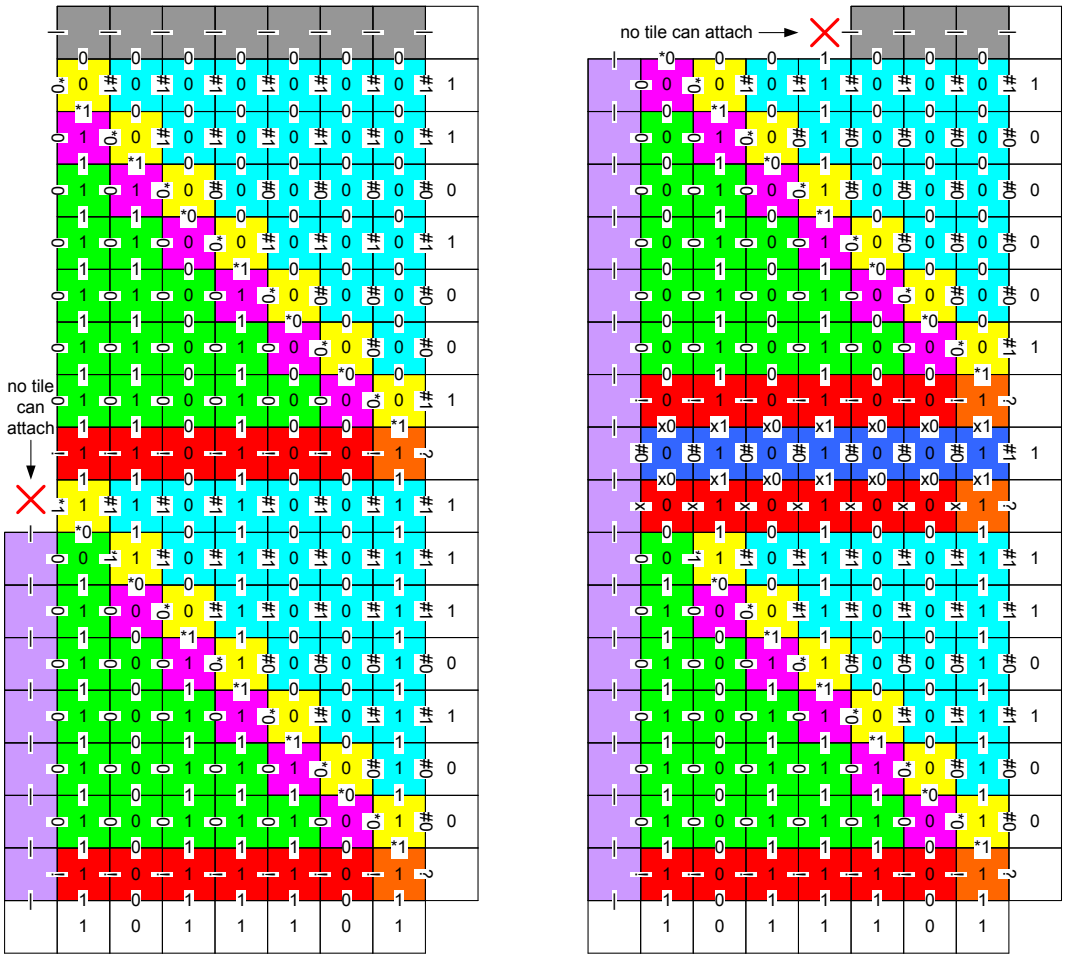
Lemma 4.3.10 *For all $n \in \mathbb{N}$, for all $\langle \vec{B}, v \rangle \in \mathbb{N}^n \times \mathbb{N}$, assuming each tile that may attach to a configuration at a certain position attaches there with a uniform probability distribution, the probability that a single nondeterministic execution of $\mathbb{S}_{SS}$ succeeds in attaching a ✓ tile if $\langle \vec{B}, v \rangle \in$ SubsetSum is at least $\left(\frac{1}{2}\right)^n$.*

Proof: If $\langle \vec{B}, v \rangle \in \mathit{SubsetSum}$, then there exists at least one choice of $\vec{c} \in \{0,1\}^n$ such that $v = \sum_{j=1}^{n} c_j B_j$. When tiles of $\mathbb{S}_{SS}$ attach to S_{SS}, only the tiles with the ? east binding domains may attach nondeterministically, there are two choices of tiles of the ones that do have the ? east binding domain, and there are exactly n places where such tiles may attach. Thus at least $\left(\frac{1}{2}\right)^n$ of the assemblies attach a ✓ tile. ■



(a) (b)

Figure 4.27: An example of $\mathbb{S}_{SS}$ solving a *SubsetSum* problem. Here, $v = 75 = 1001011_2$, and $\vec{B} = \langle 11 = 1011_2, 25 = 11001_2, 37 = 100101_2, 39 = 100111_2 \rangle$. The seed configuration encodes v on the bottom row and $\vec{B}$ on the rightmost column (a). The fact that $75 = 11 + 25 + 39$ implies that $\langle \vec{B}, t \rangle \in \mathit{SubsetSum}$, thus at least one final configuration (b) contains the ✓ tile.



(a) (b)

Figure 4.28: If the execution of $\mathbb{S}_{SS}$ does not prove membership in *SubsetSum*, the ✓ tile does not attach. If one or more of the B_i values is bigger than the number it is subtracted from, the leftmost column of tiles does not complete and the ✓ tile cannot attach (a). Similarly, if final result of subtracting some B_i values does not equal 0, the top row does not complete and the ✓ tile does not attach (b). Both these configurations are final configurations and no more tiles can attach.

Lemma 4.3.10 implies that a parallel implementation of $\mathbb{S}_{SS}$, such as a DNA implementation like those in [12, 88], with 2^n seeds has at least a $1 - \frac{1}{e} \geq 0.5$ chance of correctly deciding whether a $\langle \vec{B}, v \rangle \in \mathit{SubsetSum}$. An implementation with 100 times as many seeds has at least a $1 - \left(\frac{1}{e}\right)^{100}$ chance.

Note that T_{SS} has 49 computational tile types and uses 7 tile types to encode the input.

In Section 4.2, I showed a 50-tile system that factors numbers at temperature three. Because *SubsetSum* is NP-complete, the decision version of factoring numbers ($\langle n, a \rangle \in \mathit{Factor}$ iff there exists a prime factor of n less than a) can be reduced to *SubsetSum* and solved using $\mathbb{S}_{SS}$, thus using fewer tiles and executing the model at a lower temperature, although at the cost of having to execute several instances of $\mathbb{S}_{SS}$ (one has to solve polynomially many factoring decision problems in order to find the actual factors of a number).

4.4 Solving Satisfiability with Tiles

In this section, I will examine a system that decides an NP-complete set called SAT. This system appears in my paper [28].

The Boolean satisfiability (SAT) problem is a well-known NP-complete problem. Let $n \in \mathbb{N}$, then for all $0 \leq i < n$, let x_i be a Boolean variable that can take on values from the set $\{\mathit{TRUE}, \mathit{FALSE}\}$. Let the set of literals be the set of those variables and their negations ($\bigcup_i \{x_i, \neg x_i\}$), where $\neg \mathit{TRUE} = \mathit{FALSE}$, and $\neg \mathit{FALSE} = \mathit{TRUE}$. A clause is a disjunction of literals, e.g., $(x_0 \vee \neg x_1 \vee x_2)$. A Boolean formula, in conjunctive normal form (CNF), is a conjunction of clauses, e.g., $(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_3 \vee x_2 \vee \neg x_2)$. If every clause has exactly k literals, the Boolean formula is said to be in kCNF. A Boolean formula is satisfiable iff there exists some assignment of each variable to an element of $\{\mathit{TRUE}, \mathit{FALSE}\}$ such that the formula evaluates to TRUE.

The notions of truth assignment and literal selection are in some sense parallel, and I will use them somewhat interchangeably. Formally, every literal selection corresponds to a truth assignment. Thus I will sometimes use a literal selection, e.g., $x_0, \neg x_1, x_2$, to specify a truth assignment, in this case $x_0 = x_2 = \mathit{TRUE}$ and $x_1 = \mathit{FALSE}$.

The k-SAT problem is, given a kCNF Boolean formula, to determine whether or not it is satisfiable. It is well known that 1-SAT and 2-SAT can be solved in polynomial time, while each k-SAT for $k \geq 3$ is NP-complete. Formally, k-SAT is the set of all Boolean formula in kCNF that are satisfiable. To solve k-SAT means to decide the set k-SAT.

The rest of this section discusses solving satisfiability nondeterministically in the tile assembly model. Section 4.4.1 describes a system that uses $\Theta(n^2)$ distinct tiles to solve k-SAT for an arbitrary $k \in \mathbb{N}$, where n is the number of distinct variables in the Boolean formula. This system is similar to the system presented in [69], but I offer formal proofs of the system's correctness and speed. Section 4.4.2 describes a system that uses $\Theta(1)$ distinct tiles to solve k-SAT for an arbitrary $k \in \mathbb{N}$.

4.4.1 Naïve Approach

Lagoudakis et al. proposed a tile system for nondeterministically deciding whether a 3CNF Boolean formula is satisfiable [69]. This system uses $\Theta(n^2)$ distinct tiles, for a formula with n distinct variables, and can be adapted to decide the satisfiability of k-SAT for arbitrary $k \in \mathbb{N}$. While Lagoudakis et al. do not formally define what it means for a tile system to solve a problem or compute a function and do not formally argue that their system does in fact solve satisfiability, I believe their system can be made to fit my definitions and proven correct. What I present here is

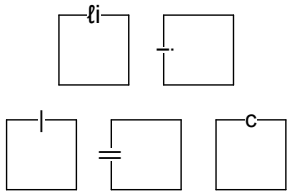


Figure 4.29: The concepts behind the tiles in Γ_n^2. For every $n \in \mathbb{N}$, the set Γ_n^2 contains $3n+3$ tiles: the three bottom-row tiles, $2n$ left top-row tiles (where ℓi enumerates over all the $2n$ literals), and n right top-row tiles (where $0 \leq i < n$).

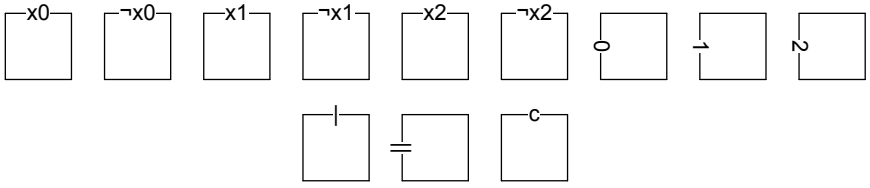


Figure 4.30: The 12 tiles in Γ_3^2.

a slight variant of their system, which follows the same basic logic. While it is of some interest to formally prove this system's correctness and speed, as I do below, the main reason for including this system is that it will explain part of the logic of the more complex system presented in Section 4.4.2.

I now describe a family of tile systems that determine whether a Boolean formula with $n \in \mathbb{N}$ distinct variables is satisfiable (because the size of the system depends on the number of variables, there will be a different system for every n, and thus I refer to the systems for all n as a family). I will refer to these systems as $\mathbb{S}_n^2$.

The idea behind encoding the input is to encode the Boolean formula in the 0^{th} row of the seed configuration with a unique tile for each possible literal, prefixing each clause with a special clause tile, and to encode the variables in the 0^{th} column with a unique tile for each variable. Figure 4.29 shows the concepts behind the tiles in Γ_n^2, used by $\mathbb{S}_n^2$. The bottom row shows three helper tiles that are the same for all n, and the top row shows the two tiles used to encode the Boolean formula (left) and the variable index (right). Thus, for a given $n \in \mathbb{N}$, the set Γ_n^2 contains the three bottom-row tiles, $2n$ left top-row tiles (where ℓi enumerates over all the $2n$ literals), and n right top-row tiles (where $0 \leq i < n$). Thus $|\Gamma_n^2| = 3n + 3$. For example, for $n = 3$, Figure 4.30 shows the 12 tiles of the set Γ_3^2.

I will use the tiles in Γ_n^2 to encode an n-variable Boolean formula in a specific way. Informally, I will place tiles representing the formula's literals in the 0^{th} row such that the literals of each clause are together, place the special clause tile to the east of each clause, place the variable tiles in the 0^{th} column, and place special end tiles in the west-most and north-most positions on that row and column. I explain the seed set up more formally below. Figure 4.31 shows a sample seed encoding the 3-variable Boolean formula $(x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$ using the tiles from Γ_3^2. Note that while this example tries to follow some logical order, the order of the variables, the clauses, and the literals within each clause is not important.

The main idea of the computation is to have tiles attach nondeterministically in column -1 to select either *TRUE* or *FALSE* for each variable and then to "sweep" those choices westward, checking if a literal in a clause evaluates to *TRUE*. Whenever a literal evaluates to *TRUE*, that information propagates northward, and along the top row, tiles attach to ensure that at least one

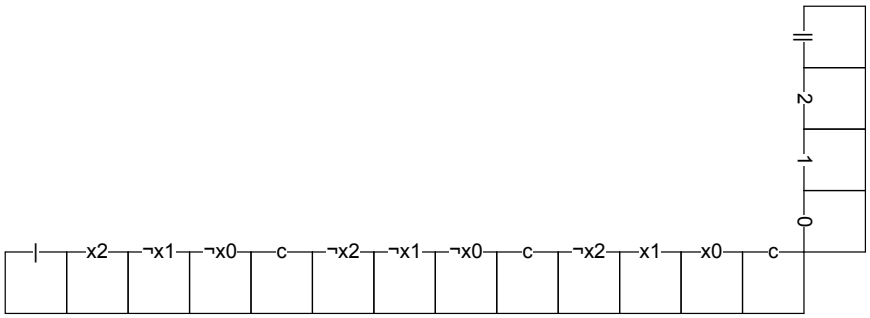


Figure 4.31: The seed encoding a 3-variable Boolean formula $(x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$ using the tiles from Γ_3^2.

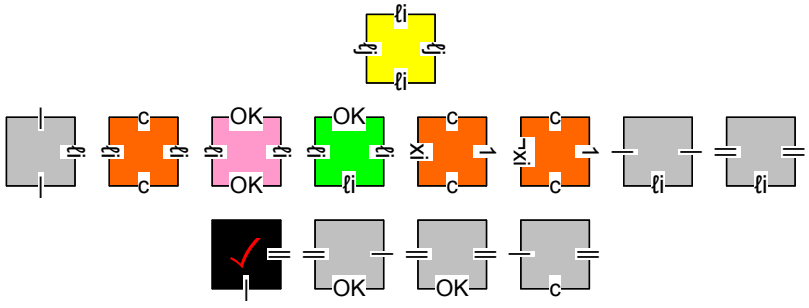


Figure 4.32: The concepts behind the tiles in T_n^2. For every $n \in \mathbb{N}$, the set T_n^2 contains $4n^2 + 12n + 4$ tiles: the four bottom-row tiles, including the special ✓ tile, $14n$ middle-row tiles, with ℓi enumerating over all the literals and $0 \leq i < n$, and $4n^2 - 2n$ top row tiles, with ℓi enumerating over all the literals and ℓj enumerating over all the literals such that $\ell i \neq \ell j$.

literal in every clause evaluates to *TRUE*. Iff that is the case, a special ✓ tile attaches in the northwest corner.

The system $\mathbb{S}_n^2$ will use the set of computational tiles T_n^2. Figure 4.32 shows the concepts behind the tiles in T_n^2. The bottom row shows four helper tiles, including the special ✓ tile, that are the same for all n. For each tile in the middle row there will be $\Theta(n)$ tiles in T_n^2, with ℓi enumerating over all the literals and $0 \leq i < n$. Finally, the top row shows the concept behind the tile which will expand to $\Theta(n^2)$ tiles in T_n^2, with ℓi enumerating over all the literals and ℓj enumerating over all the literals such that $\ell i \neq \ell j$. Thus, for a given $n \in \mathbb{N}$, the set T_n^2 contains the four bottom-row tiles, $14n$ middle-row tiles, and $4n^2 - 2n$ top-row tiles. Thus $|T_n^2| = 4n^2 + 12n + 4$. For example, for $n = 3$, Figure 4.33 shows the 76 tiles of the set T_3^2. Note that Lagoudakis et al. claim that their systems use $2n^2 + 12n + 10$ distinct tiles, but after careful analysis, I disagree with their calculations and believe their systems actually use significantly more tiles, even more than my system's $4n^2 + 12n + 4$. Most notably, their analysis assumes that there are $2n^2 - n$ tiles identical to my yellow tiles, whereas in reality there are $4n^2 - 2n$ such tiles.

The tiles of T_3^2 attach to a seed configuration, such as the one in Figure 4.31, to nondeterministically select a truth assignment of the variables and check if that assignment satisfies the Boolean formula, as shown in Figure 4.34.

Theorem 4.4.1 *For all $n \in \mathbb{N}$, let $\Sigma_n^2 = \{\mathsf{c}, |, ||, i, \mathsf{x}_i, \neg\mathsf{x}_i, \mathrm{OK}\}$, where $0 \leq i < n$. Let T_n^2 be as defined in Figure 4.32. Let $g_n^2 = 1$ and $\tau_n^2 = 2$. Let $\mathbb{S}_n^2 = \langle T_n^2, g_n^2, \tau_n^2 \rangle$. Then $\mathbb{S}_n^2$ nondeterministically*

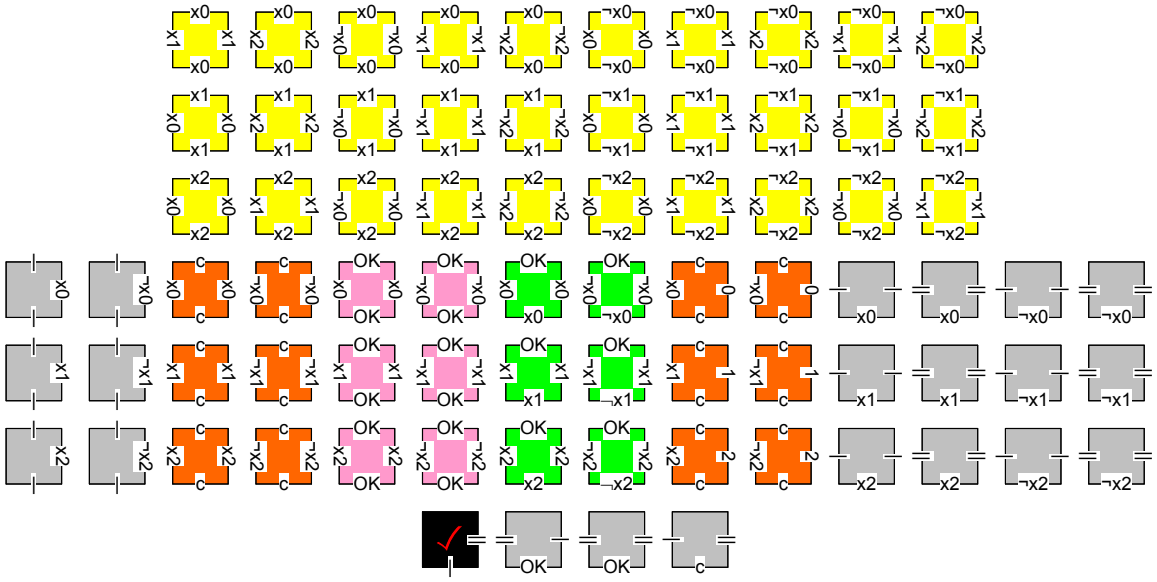


Figure 4.33: The 76 tiles in T_3^2.

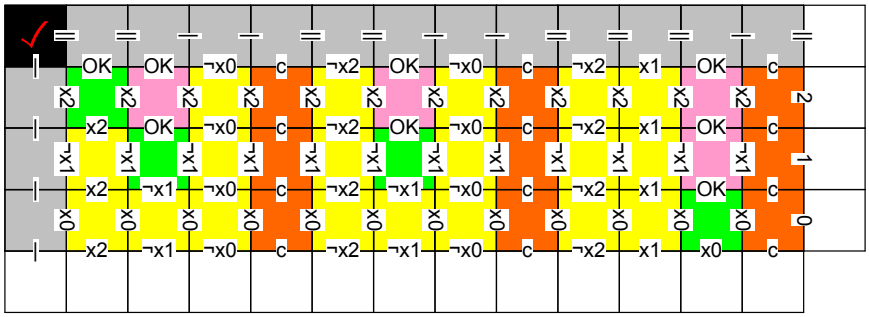


Figure 4.34: Tiles from T_n^2 attach to the seed to nondeterministically select a truth assignment. Here $\phi = (x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$ and the literal selection is $x_0, \neg x_1, x_2$ meaning that $x_0 = x_2 = \mathit{TRUE}$ and $x_1 = \mathit{FALSE}$. Tiles attach to allow the ✓ tile to attach in the northwest corner iff the assignment satisfies the Boolean formula encoded by the seed.

decides k-SAT (for all $k \in \mathbb{N}$) with up to n distinct variables per formula with the black ✓ tile from T_n^2 as the identifier tile.

Proof: To show that $\mathbb{S}_n^2$ nondeterministically decides *k-SAT* with up to n distinct variables per formula with the black ✓ tile from T_n^2 as the identifier tile, I will first describe how to construct the seed from a Boolean formula ϕ with at most n distinct variables, then argue which tiles will attach to that seed, and finally conclude that the final assembly will contain the ✓ tile iff there exists a truth assignment that makes ϕ evaluate to *TRUE*.

Let ϕ be a Boolean formula in kCNF, for some arbitrary $k \in \mathbb{N}$, with at most n distinct variables. Without loss of generality, I assume that the distinct variables are $x_0, x_1, \cdots, x_{n-1}$. Let Γ_n^2 be as defined in Figure 4.29. Let m be the number of clauses in ϕ (numbered $0, 1, \cdots, m-1$).

To assist the readability of this proof, I will define two helper functions: $x\colon \mathbb{N} \to \mathbb{N}$ and $y\colon \mathbb{N} \to \mathbb{N}$. These functions will help identify positions on the 2-D grid. For all $\hat{m} \in \mathbb{N}$, let $x(\hat{m}) = -\hat{m}(k+1) - 1$. The intuition is that I will use the $(k+1)$ columns to the west of, and

including column $x(\hat{m})$ for clause $\hat{m}$. For all $\hat{n} \in \mathbb{N}$, let $y(\hat{n}) = \hat{n} + 1$. The intuition is that I will use the row $y(\hat{n})$ for variable $\hat{n}$.

I define the seed S_n^2 that encodes ϕ as follows:

- $S_n^2(0, y(n)) = \gamma_{||}$, where $\gamma_{||}$ is the tile in Γ_n^2 with the west binding domain $||$.
- For all $0 \leq \hat{n} < n$, $S_n^2(0, y(\hat{n})) = \gamma_{\hat{n}}$, where $\gamma_{\hat{n}}$ is the tile in Γ_n^2 with the west binding domain $\hat{n}$.
- For all $0 \leq \hat{m} < m$, $S_n^2(x(\hat{m}), 0) = \gamma_c$, where γ_c is the tile in Γ_n^2 with the north binding domain c.
- For all $0 \leq \hat{m} < m$, for all $0 < \hat{k} \leq k$, $S_n^2(x(\hat{m}) - \hat{k}), 0) = \gamma_\ell$, where ℓ is the $\hat{k}^{th}$ literal of the $\hat{m}^{th}$ clause of ϕ, and γ_ℓ is the tile in Γ_n^2 with the north binding domain ℓ.
- $S_n^2(x(m), 0) = \gamma_|$, where $\gamma_|$ is the tile in Γ_n^2 with the north binding domain $|$.
- And for all other positions (v, w), $S_n^2(v, w) = empty$.

Figure 4.31 shows a sample seed for a 3-SAT formula ϕ with 3 distinct variables ($\phi = (x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$).

Note that because $\tau_n^2 = 2$, $g_n^2 = 1$, and the seed is in the shape of a horizontally-reflected L, a tile may only attach in the $\mathbb{S}_n^2$ system when its south and east neighbors are present, and only if its appropriate binding domains match those neighbors' appropriate binding domains, by the unique final configuration lemma (Lemma 4.1.2).

I examine the tiles that can attach in column -1, in positions $(-1, y(0))$ through $(-1, y(n) - 1)$. Because the west binding domains of the tiles in S_n^2 to the east of those positions are all $\hat{n}$, where $0 \leq \hat{n} < n$, the only tiles that might attach in this column are the orange tiles with $\hat{n}$ east binding domains. By induction, because the north binding domain of $S_n^2(-1, 0)$ is c, and all the south and north binding domains for those orange tiles are c, those tiles will match their neighbors on the south and east sides, and thus will attach. For each $\hat{n}$, there are 2 possible tiles that may attach in position $(-1, y(\hat{n}))$. One of these tiles will have the west binding domain $\mathsf{x}_{\hat{n}}$ and the other the west binding domain $\neg\mathsf{x}_{\hat{n}}$. These tiles will attach nondeterministically, in some sense "selecting" only a single literal for each variable. Every set of tile attachments corresponds to a particular literal selection, and every literal selection has a set of tile attachments associated with it. Given a particular literal selection, the rest of the assembly will be deterministic (note, as I go through the proof, that no position in the rest of the assembly will have an east neighbor with a west binding domain $\hat{n}$, and that no tile in T_n^2 other than those with $\hat{n}$ as their east binding domain have another tile with the same east and south binding domain pair). For the remainder of this proof, I fix the literal selection made in column -1 and refer to it as the *fixed assignment*.

Examine all the colored (neither black nor gray) tiles in T_n^2. Other than the tiles with $\hat{n}$ (where $0 \leq \hat{n} < n$) east binding domains that I just showed can only attach in column -1, these are the only tiles that may attach in the rectangle defined by, and including, columns -2 and $x(m) - 1$ and rows $y(0)$ and $y(n) - 1$. (This fact follows because the gray and black tiles may only attach in columns with a $|$ north binding domain or rows with a $|$ or $||$ west binding domain, and by induction, since gray tiles are the only ones with such north and west binding domains, they can only attach in columns and rows with those binding domains in the seed, and those are column $x(m)$ and row $y(n)$, by definition of the seed.) Let the set T_D contain only those colored tiles with an east binding domain not equal to some $\hat{n}$. Now observe that for all the tiles t in T_D, $bd_W(t) = bd_E(t)$ and either $bd_N(t) = bd_S(t)$ or $bd_N(t) = \text{OK}$. Thus I conclude, by induction, that as these tiles attach to S_n^2, as

long as every column of the rectangle is complete, the west binding domains of the tiles in row i are the same as the west binding domains of the tiles in row -1, and thus they encode the fixed literal selection, and that as long as every row of the rectangle is complete, the north binding domain of the tile in column p is either the same as the north binding domain or the seed's tile in position $(p, 0)$ or is OK.

Let ℓ be the $\hat{k}^{th}$ literal of the $\hat{m}^{th}$ clause of ϕ. I now prove that the north binding domain of the tile in position $(x(\hat{m}) - \hat{k}, y(n-1))$ is OK iff ℓ is in the fixed assignment, and thus the $\hat{m}^{th}$ clause is satisfied. The north binding domain of the tile in position $(x(\hat{m}) - \hat{k}, 0)$ is ℓ, by the definition of the seed. Let $\hat{n}$ be such that ℓ is either $x_{\hat{n}}$ or $\neg x_{\hat{n}}$. Then ℓ cannot match the literals in the assignment in rows $y(0)$ through $y(\hat{n}-1)$, and thus the north binding domain of the tile in position $(x(\hat{m}) - \hat{k}, y(\hat{n}-1))$ is ℓ. Iff ℓ is in the fixed assignment, then the tile that attaches in position $(x(\hat{m}) - \hat{k}, y(\hat{n}))$ must have matching east and south binding domains, and thus must be green, and thus has the north binding domain OK. Otherwise, the east and south binding domains do not match, and a yellow tile must attach, with the north binding domain ℓ. Since ℓ cannot match the literals in the assignment in rows $y(\hat{n}+1)$ through $y(n)-1$, iff the green tile had attached, rose tiles attach to the north, propagating the OK binding domain to row $y(n)-1$, and otherwise yellow tiles attach, propagating the ℓ binding domain. Thus the north binding domain of the tile in position $(x(\hat{m}) - \hat{k}, y(n)-1)$ is OK iff ℓ is in the fixed assignment.

Consider the tiles that attach in column $x(m)$. Because the seed has a north binding domain | in that column, by induction, the gray tiles with north and south binding domains | may attach. Since there such a is a tile for every possible literal on its east binding domain, these tiles will propagate the | binding domain to the north of tile in position $(x(m), y(n)-1)$. Consider the tiles that attach in row $y(n)$. Because the seed has the west binding domain || in that row, by induction the gray tiles with east and west binding domains in the set $\{|, ||\}$ may attach. I now show, by induction, that the west binding domain of the tile in position $(x(m)+1, y(n))$ is || iff every clause has at least one literal in the fixed assignment, and is thus satisfied. In the base case, by the definition of the seed, the west binding domain of the tile in position $(x(0)+1, y(n))$ is ||. Now I assume that the west binding domain of the tile in position $(x(\hat{m})+1, y(n))$ is || and show that the west binding domain of the tile in position $(x(\hat{m}+1)+1, y(n))$ is || iff the $\hat{m}^{th}$ clause is satisfied. Since the north binding domain of the tile in position $(x(\hat{m}), y(n-1))$ is c, the gray tile with the south binding domain c must attach in position $(x(\hat{m}), y(n))$, and its west binding domain is |. Since there is a gray tile with every possible literal as its south binding domain and | as its east and west binding domains, the | will propagate to the west until either position $(x(\hat{m}+1)+1, y(n))$ or some north binding domain is OK. If the binding domain is OK, then some literal in clause $\hat{m}$ is in the fixed assignment and thus this clause is satisfied, the gray tile with the south binding domain OK and west binding domain || attaches, and the || is propagated to the west by the gray tiles with literal and OK south binding domains and || east and west binding domains to the tile in position $(x(\hat{m}+1)+1, y(n))$. Observe that there is no tile with a south binding domain c and east binding domain |, thus the west binding domain of the tile in position $(x(m)+1, y(n))$ is || iff every clause has at least one literal in the fixed assignment, and is thus satisfied. If there exists at least one unsatisfied clause, the position $(x(m)+1, y(n))$ will either be empty, or the tile in that position will have the west binding domain |.

The ✓ tile has an east binding domain || and south binding domain | thus it can only attach in position $(x(m), y(n))$ and only if the west binding domain of the tile in position $(x(m)+1, y(n))$ is ||. Thus the ✓ tile can attach iff every clause is satisfied by the fixed assignment.

Since every possible assignment will be explored nondeterministically, if any one of them satisfies every clause, the ✓ tile will attach. If no assignment exists that satisfies every clause, then the ✓

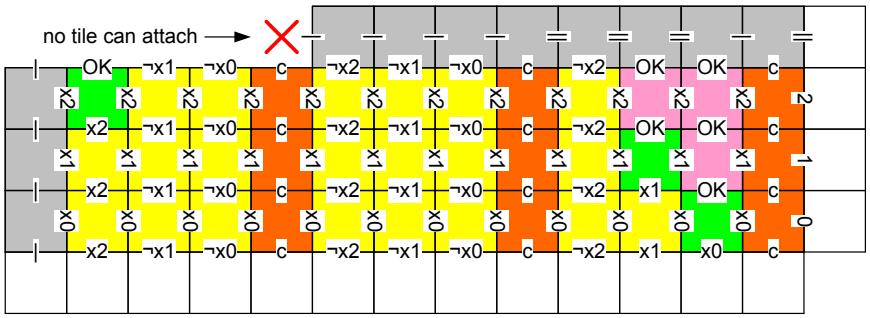


Figure 4.35: For some nondeterministic choices of the truth assignment, the Boolean formula encoded by the seed is not satisfied. Here, $\phi = (x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$ and the literal selection is x_0, x_1, x_2 meaning that $x_0 = x_1 = x_2 = TRUE$. The second clause is not satisfied and the ✓ tile never attaches. Note that this is a final configuration and no more tiles may attach.

tile will never attach. Thus $\mathbb{S}_n^2$ nondeterministically decides k-SAT (for all $k \in \mathbb{N}$) with up to n distinct variables per formula with the black ✓ tile from T_n^2 as the identifier tile. ■

Figure 4.35 shows a nondeterministically selected truth assignment that does not satisfy the Boolean formula. Because the formula is not satisfied, the ✓ tile never attaches.

Lemma 4.4.2 *The assembly time of $\mathbb{S}_n^2$ is linear in the size of the input ϕ.*

This lemma follows directly from the assembly time lemma (Lemma 4.1.4).

Lemma 4.4.3 *For all $n \in \mathbb{N}$, for all Boolean formula ϕ on n distinct variables, assuming each tile that may attach to a configuration at a certain position attaches there with a uniform probability distribution, the probability that a single nondeterministic execution of $\mathbb{S}_n^2$ succeeds in attaching a ✓ tile if ϕ is satisfiable is at least $\left(\frac{1}{2}\right)^n$.*

Proof: Only the tiles in column -1 attach nondeterministically, and at each of those positions there are exactly two tiles that may attach. If ϕ is satisfiable, then there exists at least one assignment that satisfies it, and thus a particular choice at each of the n nondeterministic positions in column -1 will select a satisfying assignment. The probability of the correct tile attaching at each location is $\frac{1}{2}$, and thus the probability of the whole column attaching to represent the correct assignment is $\left(\frac{1}{2}\right)^n$. Since the rest of the assembly is deterministic, $\left(\frac{1}{2}\right)^n$ is the probability that a single nondeterministic execution of $\mathbb{S}_n^2$ succeeds in attaching a ✓ tile if ϕ is satisfiable. ■

In summary, $\mathbb{S}_n^2$ decides whether a kCNF Boolean formula ϕ on n variables is in k-SAT, has $4n^2 + 12n + 4 = \Theta(n^2)$ computational tile types and uses $3n + 3 = \Theta(n)$ tile types to encode the input. It computes in time linear in the size of the input, and each assembly has the probability of at least $\left(\frac{1}{2}\right)^n$ of finding the satisfying assignment, if one exists.

4.4.2 Constant-Size Tileset Approach

I now describe a nondeterministic tile system $\mathbb{S}_{SAT}$, which will follow a logic similar to that of $\mathbb{S}_n^2$, but will use only a constant number of tiles. The idea of $\mathbb{S}_{SAT}$ is to encode ϕ in the same way $\mathbb{S}_n^2$ did, but instead of using a single tile for each literal, the literals will be encoded by a tile that indicates whether the literal is a negation (I place this tile in the east-most position), and a series of 0 and 1 tiles encoding, in binary, the number of the variable. For example, the literal x_5 would

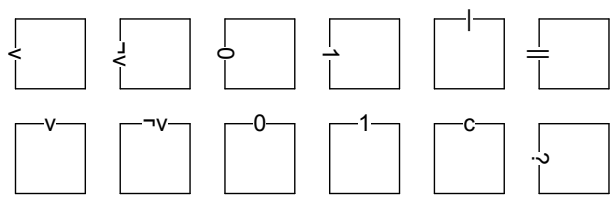


Figure 4.36: The 12 tiles in Γ_{SAT}.

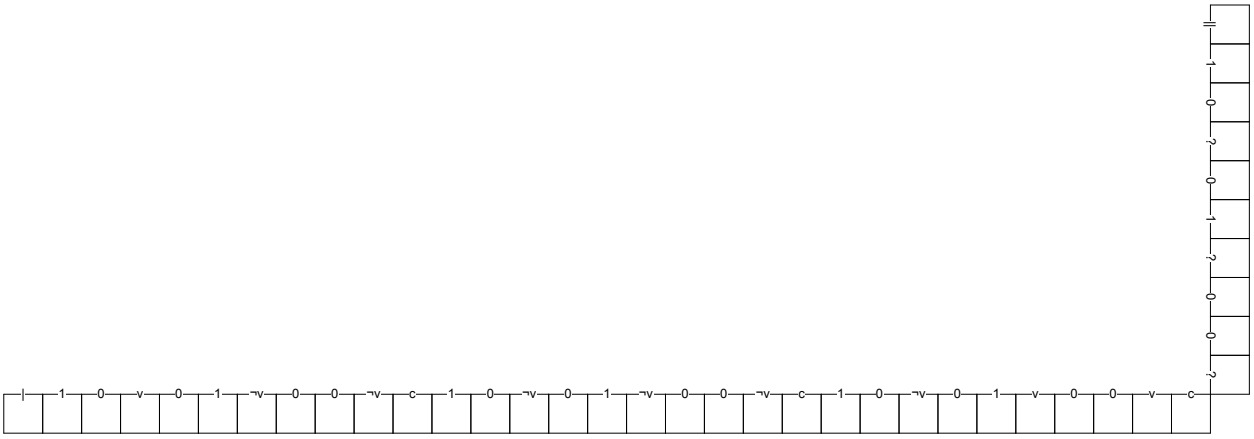


Figure 4.37: The seed encoding a 3-variable Boolean formula $(x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$ using the tiles from Γ_{SAT}.

be encoded by a v tile, and by a 1 tile, then a 0 tile, and then a 1 tile (101v) because $5 = 101_2$. The literal $\neg x_4$ would be encoded by a $\neg$v tile, and by a 1 tile, then a 0 tile, and then a 0 tile (100$\neg$v) because $4 = 100_2$. For consistency, I will use the same number of bits to encode all variables, e.g., if my ϕ has 7 distinct variables, I will need three bits to encode x_7, so I will encode x_1 as 001v. Similarly, I will encode the variables in the 0^{th} column using this binary encoding method. The assemblies in $\mathbb{S}_{SAT}$ will be larger in size than the assemblies in $\mathbb{S}_n^2$ because what used to be encoded by a single tile will now be represented by a $\Theta(\log n) \times \Theta(\log n)$ block of tiles, but the overall logic will remain the same. The key to $\mathbb{S}_{SAT}$ is building the logic of the blocks to correctly match literals without affecting the inherited logic of $\mathbb{S}_n^2$.

There are 12 tiles in Γ_{SAT}, no matter how large ϕ is or how many variables it contains. I will use the tiles in Γ_{SAT} to encode an n-variable Boolean formula in a specific way. Informally, I will encode the formula's literals in the 0^{th} row, as described above, such that the literals of each clause are together, place the special clause tile to the east of each clause, place the encoded variables in the 0^{th} column, and place special end tiles in the west-most and north-most positions on that row and column. I explain the seed set up more formally below. Figure 4.37 shows a sample seed encoding the 3-variable Boolean formula $(x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$ using the tiles from Γ_{SAT}. Note that while this example tries to follow some logical order, the order of the variables, the clauses, and the literals within each clause is not important.

Just as before, the main idea of the computation is to have tiles attach nondeterministically in column -1 to select either *TRUE* or *FALSE* for each variable and then to "sweep" those choices westward, checking if a literal in a clause evaluates to *TRUE*. The comparison of literals will take place within a $\Theta(\log n) \times \Theta(\log n)$ block of tiles. Whenever a literal evaluates to *TRUE*, that information propagates northward, and along the top row, tiles attach to ensure that at least one

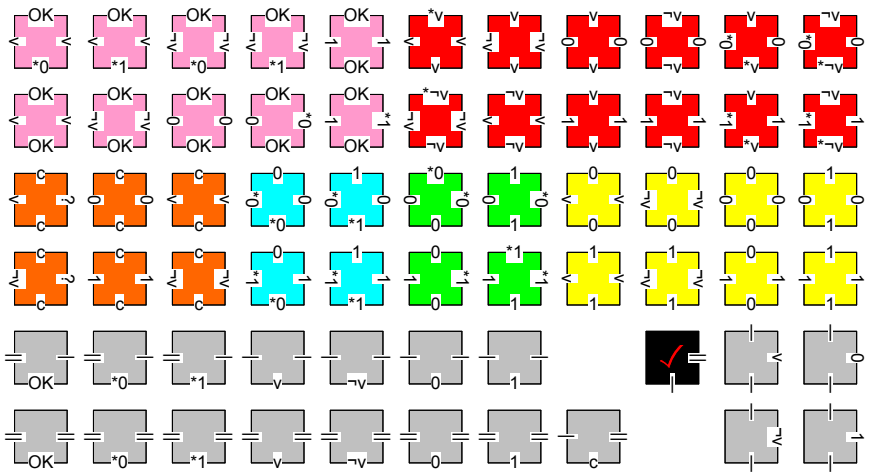


Figure 4.38: The 64 tiles in T_{SAT}.

literal in every clause evaluates to *TRUE*. Iff that is the case, a special ✓ tile attaches in the northwest corner.

The system $\mathbb{S}_{SAT}$ will use the set of computational tiles T_{SAT}. Figure 4.38 shows the 64 tiles in T_{SAT}. The colors of the tiles are coordinated with the colors of the T_n^2 system — the tiles of the same colors perform the same functions.

The tiles of T_{SAT} attach to a seed configuration, such as the one in Figure 4.37, to nondeterministically select a truth assignment of the variables and check if that assignment satisfies the Boolean formula, as shown in Figure 4.39.

Lemma 4.4.4 *Let a configuration S (confined to some $\nu \times \nu$ square with the southeast tile at position (x_0, y_0)) be such that:*

- $bd_N(S(x_0 - 1, y_0)) \in \{\mathsf{v}, \neg\mathsf{v}\}$,
- $bd_W(S(x_0, y_0 + 1)) \in \{\mathsf{v}, \neg\mathsf{v}\}$,
- *for all* $1 < i \leq \nu$, $bd_N(S(x_0 - i, y_0)) \in \{0, 1\}$,
- *for all* $1 < i \leq \nu$, $bd_W(S(x_0, y_0 + i)) \in \{0, 1\}$, *and*
- *for all positions* (x, y) *within the square,* $S(x, y) = empty$.

And let $g_= = 1$ *and* $\tau_= = 2$, *and* $\mathbb{S}_= = \langle T_{SAT}, g_=, \tau_= \rangle$. *Then* $\mathbb{S}_=$ *produces a final unique configuration F on S such that:*

- $bd_N(F(x_0 - \nu, y_0 + \nu))$ *contains a* $\star$ *iff for all* $0 < i \leq \nu$, $bd_N(x_0 - i, y_0) = bd_W(x_0, y_0 + i)$,
- *no other tile in row* $y_0 + \nu$ *has a north binding domain that contains a* $\star$, *and*
- *no tile in column* $x_0 - \nu$ *has a west binding domain that contains a* $\star$.

Proof: Let a relationship $\approx$ be defined on binding domains such that given binding domains a and b, $a \approx b$ iff the portion of a that is not $\star$ exactly equals the portion of b that is not $\star$. For example, $0 \approx 0 \approx \star 0 \not\approx \star 1$. Observe the for all tiles $t \in T_{SAT}$, either $bd_N(t) = \text{OK}$ or $bd_N(t) \approx bd_S(t)$, and either $bd_E(t) =?$ or $bd_E(t) \approx bd_W(t)$. That is to say, other than the tiles with

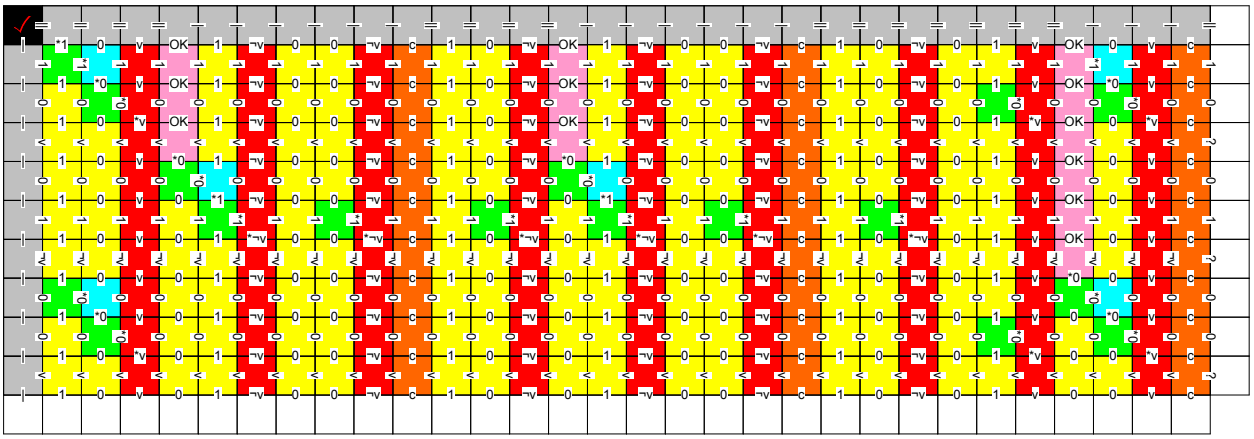


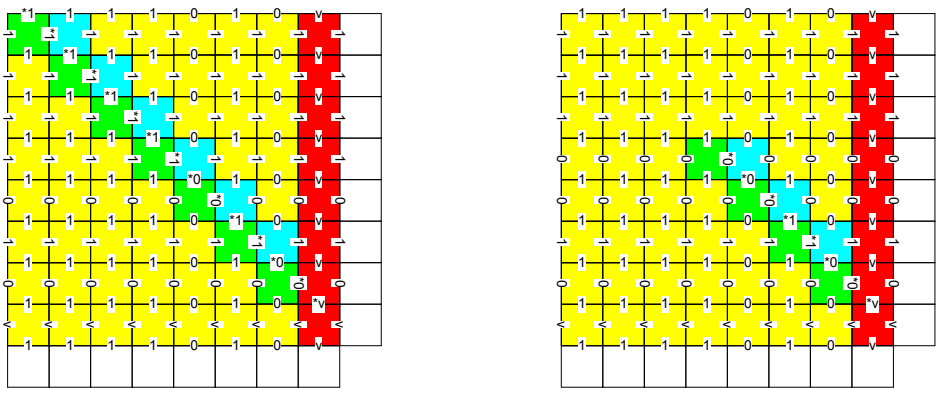
Figure 4.39: Tiles from T_{SAT} attach to the seed to nondeterministically select a truth assignment. Here $\phi = (x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$ and the literal selection is $x_0, \neg x_1, x_2$ meaning that $x_0 = x_2 = TRUE$ and $x_1 = FALSE$. Tiles attach to allow the ✓ tile to attach in the northwest corner iff the assignment satisfies the Boolean formula encoded by the seed.

a ?, the east binding domain $\approx$ the west binding domain, and other than the tiles with an OK, the north binding domain $\approx$ the south binding domain. It will become clear in this proof that the tiles with ? and OK binding domains will never attach to S, thus the west binding domain of all tiles in row r will $\approx bd_W(S(x_0, r))$ and the north binding domains of all tiles in column c will $\approx bd_N(S(c, y_0))$.

I now show by induction that iff for all $\mu \leq \nu$, for all $0 < i \leq \mu$, $bd_N(S(x_0 - i, y_0)) = bd_W(S(x_0, y_0 + i))$ then the only tile in row $y_0 + \mu$ whose north binding domain contains a $\star$ is in position $(x_0 - \mu, y_0 + \mu)$ and that no tile in column $x_0 - \mu$ has a west binding domain that contains a $\star$. Otherwise, none of those binding domains contain a $\star$.

First, examine the base case. Let $\mu = 1$. Examine row $(y_0 + \mu)$. By the definition of S, in position $(x_0 - \mu, y_0 + \mu)$, one of the red tiles from T_{SAT} must attach. Iff the east and south binding domains of that red tile are equal (and thus $bd_N(S(x_0 - \mu, y_0)) = bd_W(S(x_0, y_0 + \mu))$) then the north binding domain of the tile contains a $\star$. It follows that the yellow tiles will attach in row $(y_0 + \mu)$ to the west of that red tile. The yellow tiles contain no binding domains with $\star$.

Now assume that if for all $0 < i \leq \mu$, $bd_N(S(x_0 - i, y_0)) = bd_W(S(x_0, y_0 + i))$ then the only tile in row $y_0 + \mu$ whose north binding domain contains a $\star$ is in position $(x_0 - \mu, y_0 + \mu)$ and that no tile in column $x_0 - \mu$ has a west binding domain that contains a $\star$, and that otherwise, none of those binding domains contain a $\star$. I will now show that if $bd_N(S(x_0 - (\mu + 1), y_0)) = bd_W(S(x_0, y_0 + (\mu + 1)))$, then the only tile in row $y_0 + \mu + 1$ whose north binding domain contains a $\star$ is in position $(x_0 - (\mu + 1), y_0 + (\mu + 1))$ and no tile in column $x_0 - (\mu + 1)$ has a west binding domain that contains a $\star$, and that otherwise, none of those binding domains contain a $\star$. Examine row $(y_0 + (\mu + 1))$. If the north binding domain of the tile in position $(x_0 - \mu, y_0 + \mu)$ contains a $\star$ (thus the appropriate north and west binding domains of S have matched up to μ), then a blue tile must attach in position $(x_0 - \mu 1, y_0 + (\mu + 1))$, with a west binding domain containing a $\star$. Then a green tile must attach to the west, in position $(x_0 - (\mu + 1), y_0 + (\mu + 1))$. Iff the east and south binding domains of that green tile are $\approx$ (and thus $bd_N(S(x_0 - (\mu + 1), y_0)) = bd_W(S(x_0, y_0 + (\mu + 1)))$) then the green tile's north binding domain contains a $\star$. Yellow tiles attach in column $x_0 - (\mu + 1)$ below that green tile. Thus if $bd_N(S(x_0 - (\mu + 1), y_0)) = bd_W(S(x_0, y_0 + (\mu + 1)))$, then the only tile in



(a) (b)

Figure 4.40: Tiles comparing two inputs. In (a), the comparison is between 1111010v and 1111010v. Because the two inputs are the same, the northwest tile's north binding domain contains a $\star$, and none of the rest of the exposed binding domains do. In (b), the comparison is between 1111010v and 1110010v. Because the two inputs do not match, no exposed binding domain contains a $\star$.

row $y_0 + \mu + 1$ whose north binding domain contains a $\star$ is in position $(x_0 - (\mu + 1), y_0 + (\mu + 1))$ and no tile in column $x_0 - (\mu + 1)$ has a west binding domain that contains a $\star$, and otherwise, none of those binding domains contain a $\star$.

The lemma follows when $\mu = \nu$. ■

Lemma 4.4.4 describes a subset of T_{SAT} attaching to compare two inputs. Figure 4.40(a) shows a comparison of 1111010v and 1111010v. Because the two inputs are the same, the northwest tile's north binding domain contains a $\star$, and none of the rest of the exposed binding domains do. Figure 4.40(b) shows a comparison of 1111010v and 1110010v. Because the two inputs do not match, no exposed binding domain contains a $\star$.

Theorem 4.4.5 *Let* $\Sigma_{SAT} = \{\mathsf{c}, ?, |, ||, \mathsf{v}, {}^\star\mathsf{v}, \neg\mathsf{v}, {}^\star\neg\mathsf{v}, 0, {}^\star 0, 1, {}^\star 1, \text{OK}\}$. *Let* T_{SAT} *be as defined in Figure 4.38. Let* $g_{SAT} = 1$ *and* $\tau_{SAT} = 2$. *Let* $\mathbb{S}_{SAT} = \langle T_{SAT}, g_{SAT}, \tau_{SAT} \rangle$. *Then* $\mathbb{S}_{SAT}$ *nondeterministically decides k-SAT (for all* $k \in \mathbb{N}$*) with the black* ✓ *tile from* T_{SAT} *as the identifier tile.*

Proof: To show that $\mathbb{S}_{SAT}$ nondeterministically decides *k-SAT* with the black ✓ tile from T_{SAT} as the identifier tile, I will first describe how to construct the seed from a Boolean formula ϕ, then argue which tiles will attach to that seed, and finally conclude that the final assembly will contain the ✓ tile iff there exists a truth assignment that makes ϕ evaluate to *TRUE*.

Let ϕ be a Boolean formula in kCNF, for some arbitrary $k \in \mathbb{N}$. Let n be the number of distinct variables in ϕ, and let $\nu = \lceil \lg n \rceil + 1$. The value ν is the number of tiles used to encode each variable ($\lg n$ tiles to encode the variable's number in binary and 1 tile to encode whether the variable is negated). Note that $\mathbb{S}_{SAT}$ works for all n, and ν depends on n only to properly encode the variables in the seed. Without loss of generality, I assume that the distinct variables of ϕ are $x_0, x_1, \cdots, x_{n-1}$. Let Γ_{SAT} be as defined in Figure 4.36. Let m be the number of clauses in ϕ, numbered $0, 1, \cdots, m-1$.

To assist the readability of this proof, I will define three helper functions: $x\colon \mathbb{N} \to \mathbb{N}$, $v\colon \mathbb{N} \to \mathbb{N}$ and $y\colon \mathbb{N} \to \mathbb{N}$. These functions will help identify positions on the 2-D grid. For all $\hat{m} \in \mathbb{N}$, let $x(\hat{m}) = -\hat{m}(k\nu + 1) - 1$. The intuition is that I will use the $(k\nu + 1)$ columns to the west of, and including column $x(\hat{m})$ for clause $\hat{m}$. For all $\hat{k} \in \mathbb{N}$, let $v(\hat{k}) = \hat{k}\nu + 1$. The intuition is that I will

the ν columns to the west of, and including column $x(\hat{m}) - v(\hat{k})$ for the $\hat{k}^{th}$ literal in the $\hat{m}$ clause (where the literals of a clause are $\ell_0, \ell_1, \cdots, \ell_{k-1}$). For all $\hat{n} \in \mathbb{N}$, let $y(\hat{n}) = \hat{n}\nu + 1$. The intuition is that I will use ν rows to the north of, and including row $y(\hat{n})$ for variable $\hat{n}$.

I define the seed S_{SAT} that encodes ϕ as follows:

- $S_{SAT}(0, y(n)) = \gamma_{||}$, where $\gamma_{||}$ is the tile in Γ_{SAT} with the west binding domain $||$.
- For all $0 \leq \hat{n} < n$, $S_{SAT}(0, y(\hat{n})) = \gamma_{?}$, where $\gamma_{?}$ is the tile in Γ_{SAT} with the west binding domain ?.
- For all $0 \leq \hat{n} < n$, for all $0 \leq i < \nu - 1$, $S_{SAT}(0, y(\hat{n}) + i + 1) = \gamma_z$, where z is the i^{th} bit of $\hat{n}$ ($z = \left\lfloor \frac{\hat{n}}{2^i} \right\rfloor \bmod 2$) and γ_z is the tile in Γ_{SAT} with the west binding domain z.
- For all $0 \leq \hat{m} < m$, $S_{SAT}(x(\hat{m}), 0) = \gamma_c$, where γ_c is the tile in Γ_{SAT} with the north binding domain c.
- For all $0 \leq \hat{m} < m$, for all $0 \leq \hat{k} < k$, $S_{SAT}(x(\hat{m}) - v(\hat{k}), 0) = \gamma_{\neg}$, where if the $\hat{k}^{th}$ literal in the $\hat{m}^{th}$ clause is some unnegated variable then $\gamma_{\neg}$ is the tile in Γ_{SAT} with the north binding domain v and if the $\hat{k}^{th}$ literal in the $\hat{m}^{th}$ clause is the negation of some variable then $\gamma_{\neg}$ is the tile in Γ_{SAT} with the north binding domain $\neg\mathsf{v}$.
- For all $0 \leq \hat{m} < m$, for all $0 \leq \hat{k} < k$, for all $0 \leq i < \nu - 1$, $S_{SAT}(x(\hat{m}) - v(\hat{k}) - i - 1, 0) = \gamma_z$, where if w is such that the $\hat{k}^{th}$ literal in the $\hat{m}^{th}$ clause is either x_w or $\neg x_w$, then z is the i^{th} bit of w ($z = \left\lfloor \frac{w}{2^i} \right\rfloor \bmod 2$) and γ_z is the tile in Γ_{SAT} with the north binding domain z.
- $S_{SAT}(x(m), 0) = \gamma_{|}$, where $\gamma_{|}$ is the tile in Γ_{SAT} with the north binding domain $|$.
- And for all other positions (v, w), $S_{SAT}(v, w) = \mathit{empty}$.

Figure 4.37 shows a sample seed for a 3-SAT formula ϕ with 3 distinct variables ($\phi = (x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$).

Note that because $\tau_{SAT} = 2$, $g_{SAT} = 1$, and the seed is in the shape of a horizontally-reflected L, a tile may only attach in the $\mathbb{S}_{SAT}$ system when its south and east neighbors are present, and only if its appropriate binding domains match those neighbors' appropriate binding domains, by the unique final configuration lemma (Lemma 4.1.2).

I examine the tiles that can attach in column -1, in positions $(-1, y(0))$ through $(-1, y(n) - 1)$. First, consider the positions $(-1, y(\hat{n}))$, for all $0 \leq \hat{n} < n$. Because the west binding domains of the tiles in S_{SAT} to the east of those positions are all ?, the only tiles that might attach in this column are the orange tiles with ? east binding domains. In the other positions, the orange tiles with the east and west binding domains 0 and 1 may attach. By induction, because the north binding domain of $S_{SAT}(-1, 0)$ is c, and all the south and north binding domains for those orange tiles are c, those tiles will match their neighbors on the south and east sides, and thus will attach. For each $\hat{n}$, there are 2 possible tiles that may attach in position $(-1, y(\hat{n}))$. One of these tiles will have the west binding domain v and the other the west binding domain $\neg\mathsf{v}$. These tiles will attach nondeterministically, in some sense "selecting" only a single literal for each variable. Every set of tile attachments corresponds to a particular literal selection, and every literal selection has a set of tile attachments associated with it. Given a particular literal selection, the rest of the assembly will be deterministic (note, as I go through the proof, that no position in the rest of the assembly will have an east neighbor with a west binding domain ?, and that no tile in T_{SAT} other than those with ? as their east binding domain have another tile with the same east and south binding domain

pair). For the remainder of this proof, I fix the literal selection made in column -1 and refer to it as the *fixed assignment*.

Examine all the colored (neither black nor gray) tiles in T_{SAT}. Other than the tiles with ? east binding domains that I just showed can only attach in column -1, these are the only tiles that may attach in the rectangle defined by, and including, columns -2 and $x(m) - 1$ and rows $y(0)$ and $y(n) - 1$. (This fact follows because the gray and black tiles may only attach in columns with a | north binding domain or rows with a | or || west binding domain, and by induction, since gray tiles are the only ones with such north and west binding domains, they can only attach in columns and rows with those binding domains in the seed, and those are column $x(m)$ and row $y(n)$, by definition of the seed.) Let the set T_D contain only those colored tiles with an east binding domain not equal to ?. Now observe that for all the tiles t in T_D, $bd_W(t) \approx bd_E(t)$ and either $bd_N(t) \approx bd_S(t)$ or $bd_N(t) = \text{OK}$ (using the definition of $\approx$ from the proof of Lemma 4.4.4). Thus I conclude, by induction, that as these tiles attach to S_{SAT}, as long as every column of the rectangle is complete, the west binding domains of the tiles in row i are $\approx$ the west binding domains of the tiles in row -1, and thus they encode the fixed literal selection, and that as long as every row of the rectangle is complete, the north binding domain of the tile in column p is either $\approx$ the north binding domain or the seed's tile in position $(p, 0)$ or is OK.

Let ℓ be the $\hat{k}^{th}$ literal of the $\hat{m}^{th}$ clause of ϕ. I now prove that the north binding domain of the tile in position $(x(\hat{m}) - (v(\hat{k} + 1) + 1), y(n) - 1)$ is either OK or contains a $\star$ iff ℓ is in the fixed assignment, and thus the $\hat{m}^{th}$ clause is satisfied. Examine the ν columns to the west of, and including column $x(\hat{m}) - v(\hat{k})$. Those columns' intersections with the rows $y(0)$ through $y(1) - 1$ are a $\nu \times \nu$ square matching the description of the $\nu \times \nu$ square in Lemma 4.4.4. The north binding domains to the south of the square encode ℓ. The west binding domains to the east of the square encode the literal of the fixed assignment that is either x_0 or $\neg x_0$. By Lemma 4.4.4, the north binding domain of the tile in position $(x(\hat{m}) - v(\hat{k} + 1) + 1, y(1) - 1)$ contains a $\star$ iff ℓ is in the fixed assignment, and thus if the $\hat{m}^{th}$ clause is satisfied. If that binding domain has no $\star$, then by induction, ℓ is compared with each other literal in the fixed assignment in rows $y(1)$ through $y(n) - 1$. If ever a north binding domain in column $(x(\hat{m}) - v(\hat{k} + 1) + 1, y(1) - 1)$ contains a $\star$ (except in row $y(n) - 1$), then the tile that attaches to it has the east binding domain either v or $\neg\mathsf{v}$ and thus must be rose and has the north binding domain OK. That OK binding domain is then propagated to row $y(n) - 1$ by the rose tiles. Thus iff ℓ is in the fixed assignment, the north binding domain of the tile in position $(x(\hat{m}) - v(\hat{k} + 1) + 1, y(n) - 1)$ is either OK or contains a $\star$.

Consider the tiles that attach in column $x(m)$. Because the seed has a north binding domain | in that column, by induction, the gray tiles with north and south binding domains | may attach. Since for every element of $\{\mathsf{v}, \neg\mathsf{v}, 0, 1\}$, there is a tile with that east binding domain, these tiles will propagate the | binding domain to the north of tile in position $(x(m), y(n) - 1)$.

Consider the tiles that attach in row $y(n)$. Because the seed has the west binding domain || in that row, by induction the gray tiles with east and west binding domains in the set $\{|, ||\}$ may attach. I now show, by induction, that the west binding domain of the tile in position $(x(m) + 1, y(n))$ is || iff every clause has at least one literal in the fixed assignment, and is thus satisfied. In the base case, by the definition of the seed, the west binding domain of the tile in position $(x(0) + 1, y(n))$ is ||. Now I assume that the west binding domain of the tile in position $(x(\hat{m}) + 1, y(n))$ is || and show that the west binding domain of the tile in position $(x(\hat{m} + 1) + 1, y(n))$ is || iff the $\hat{m}^{th}$ clause is satisfied. Since the north binding domain of the tile in position $(x(\hat{m}), y(n - 1))$ is c, the gray tile with the south binding domain c must attach in position $(x(\hat{m}), y(n))$, and its west binding domain is |. Since there is a gray tile with every element of $\{\mathsf{v}, \neg\mathsf{v}, 0, 1\}$ as its south binding domain and | as its east and west binding domains, the | will propagate to the west until either position $(x(\hat{m} + 1) + 1, y(n))$ or some north binding domain is either OK or contains a $\star$. If the

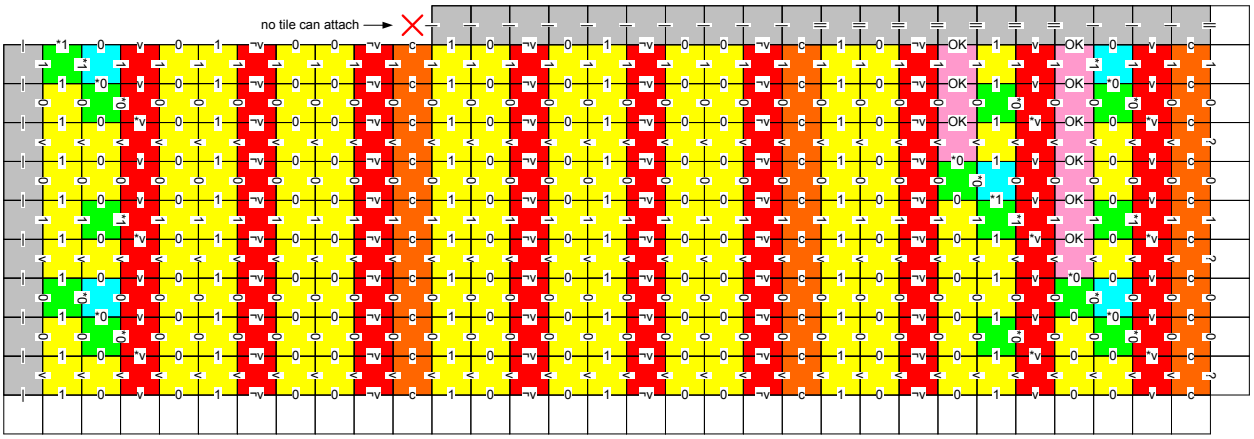


Figure 4.41: For some nondeterministic choices of the truth assignment, the Boolean formula encoded by the seed is not satisfied. Here, $\phi = (x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$ and the literal selection is x_0, x_1, x_2 meaning that $x_0 = x_1 = x_2 = \mathit{TRUE}$. The second clause is not satisfied and the ✓ tile never attaches. Note that this is a final configuration and no more tiles may attach.

binding domain is OK or contains a $\star$, then some literal in clause $\hat{m}$ is in the fixed assignment and thus this clause is satisfied, a gray tile with the south binding domain OK or containing a $\star$ and west binding domain || attaches, and the || is propagated to the west by the gray tiles with $\{\mathsf{v}, \neg\mathsf{v}, 0, 1, \star 0, \star 1, \mathrm{OK}\}$ south binding domains and || east and west binding domains to the tile in position $(x(\hat{m}+1)+1, y(n))$. Observe that there is no tile with a south binding domain c and east binding domain |, thus the west binding domain of the tile in position $(x(m)+1, y(n))$ is || iff every clause has at least one literal in the fixed assignment, and is thus satisfied. If there exists at least one unsatisfied clause, the position $(x(m)+1, y(n))$ will either be empty, or the tile in that position will have the west binding domain |.

The ✓ tile has an east binding domain || and south binding domain | thus it can only attach in position $(x(m), y(n))$ and only if the west binding domain of the tile in position $(x(m)+1, y(n))$ is ||. Thus the ✓ tile can attach iff every clause is satisfied by the fixed assignment.

Since every possible assignment will be explored nondeterministically, if any one of them satisfies every clause, the ✓ tile will attach. If no assignment exists that satisfies every clause, then the ✓ tile will never attach. Thus $\mathbb{S}_{SAT}$ nondeterministically decides k-SAT (for all $k \in \mathbb{N}$) with the black ✓ tile from T_{SAT} as the identifier tile. ■

Figure 4.41 shows a nondeterministically selected truth assignment that does not satisfy the Boolean formula. Because the formula is not satisfied, the ✓ tile never attaches.

Lemma 4.4.6 *The assembly time of $\mathbb{S}_{SAT}$ is linear in the number of bits necessary to describe ϕ (i.e., the size of ϕ).*

Proof: For a ϕ in kCNF with m clauses and n distinct variables, each literal can be described using $\Theta(\log n)$ bits. Thus ϕ can be described using $\Theta(mk \log n)$ bits. The dimensions of the rectangle formed by the seed S_{SAT} are $\Theta(mk \log n) \times \Theta(n \log n)$ and it follows from the assembly time lemma (Lemma 4.1.4) that the assembly time of $\mathbb{S}_{SAT}$ is $\Theta(mk \log n)$, which is linear in the size of ϕ. ■

Lemma 4.4.7 *For all $n \in \mathbb{N}$, for all Boolean formula ϕ on n distinct variables, assuming each tile that may attach to a configuration at a certain position attaches there with a uniform probability*

distribution, the probability that a single nondeterministic execution of $\mathbb{S}_{SAT}$ *succeeds in attaching a* ✓ *tile if* ϕ *is satisfiable is at least* $\left(\frac{1}{2}\right)^n$.

Proof: The only tiles that attach nondeterministically in $\mathbb{S}_{SAT}$ attach in positions $(-1, y(\hat{n}))$ for all $0 \leq \hat{n} < n$. At each of those positions there are exactly two tiles that may attach. If ϕ is satisfiable, then there exists at least one assignment that satisfies it, and thus a particular choice at each of the n nondeterministic positions in column -1 will select a satisfying assignment. The probability of the correct tile attaching at each location is $\frac{1}{2}$, and thus the probability of the whole column attaching to represent the correct assignment is $\left(\frac{1}{2}\right)^n$. Since the rest of the assembly is deterministic, $\left(\frac{1}{2}\right)^n$ is the probability that a single nondeterministic execution of $\mathbb{S}_{SAT}$ succeeds in attaching a ✓ tile if ϕ is satisfiable. ■

In summary, $\mathbb{S}_{SAT}$ decides whether a kCNF Boolean formula ϕ on n variables is in k-SAT, has $64 = \Theta(1)$ computational tile types and uses $12 = \Theta(1)$ tile types to encode the input. It computes in time linear in the size of the input, and each assembly has the probability of at least $\left(\frac{1}{2}\right)^n$ of finding the satisfying assignment, if one exists.

I have designed two systems that solve well known NP-complete problems k-SAT, for all $k \in \mathbb{N}$, in the tile assembly model. The first system, $\mathbb{S}_n^2$, uses $\Theta(n^2)$ distinct tiles to decide a Boolean formula with n distinct variables and is closely related to a previously described though unproven system [69]. The second system, $\mathbb{S}_{SAT}$, uses $64 = \Theta(1)$ distinct tiles to decide a Boolean formula. I prove the correctness of both systems and analyze their size and time complexities. Both systems compute in time linear in the input size. Each nondeterministic assembly has a probability of success of at least $\left(\frac{1}{2}\right)^n$, where n is the number of distinct variables. Thus a parallel implementation of $\mathbb{S}_{SAT}$, such as a DNA implementation like those in [12,88], with 2^n seeds has at least a $1 - \frac{1}{e} \geq 0.5$ chance of correctly deciding whether a Boolean formula is satisfiable. An implementation with 100 times as many seeds has at least a $1 - \left(\frac{1}{e}\right)^{100}$ chance.

Tile system solutions to problems that require such unique identifiers on variables, nodes, or edges have resorted to using $\Theta(n)$ tiles to encode the input, and no fewer than $\Theta(n^2)$ tiles to compute, for inputs of size n [69]. My proposal is the first constant-size tileset solution that solves such a problem, and the mechanism I design for uniquely addressing variables is completely portable to solving other such problems, such as graph problems, many of which are known to be NP-complete.

Part II

Software Architectural Style for Internet-Sized Networks

Chapter 5

Background and Related Work in Software Engineering

The Internet's growth has created networks with great computing potential without a clear way to harness that potential to solve memory- and processor time-intensive problems. Networks, such as the Internet, have the potential to solve NP-complete problems (and other problems for which we do not know polynomial-time solutions) quickly, but as their individual nodes may be unreliable or malicious, users may desire guarantees that their computations are correct and are kept confidential. Mechanisms for distributing computation over such large networks are likely to require a great deal of collaboration, while the large size of the network is likely to require that collaboration to scale well.

I propose an architectural style, called the tile style, for discreetly distributing computation over a large network. This style heavily leverages the self-assembly work presented in this dissertation.

The tile architectural style is particularly applicable to problems that are computationally intensive and easily parallelizable. Computationally intensive problems are ones that a single computer is unlikely to solve quickly, while easily parallelizable problems are ones that inherently yield a large number of parallel threads. For example, all NP-complete problems have both of those properties [98]. Further, my work is applicable to users who desire discreetness and have access to large but unreliable networks. By discreetness, I mean that the user does not want others to find out the input or the algorithm. By large but unreliable network, I mean a network, such as the Internet, that is partially or entirely outside of the user's control, and perhaps even hostile.

What follows are three scenarios that are at the heart of the problems the tile architectural style tackles.

1. A large university wishes to digitally deliver recent graduates' transcripts to graduate schools and employers. This information is sensitive, and needs to be encrypted using the recipient's public key and digitally signed using the university's private key. It may take the university's transcript department's computer a long time to encrypt and sign the thousands of requests, but fortunately it has access to the university's network of computers. While that network may be large, the individual nodes are insecure and cannot be trusted with sensitive data such as the university's private key or the students' transcripts.

2. An espionage agency is attempting to break an RSA code sent by an enemy. The agency wishes to use a large network to factor the enemy's public key; however, it cannot allow anyone to know the key's factors or even whose key it is factoring. Since the agency has access to the Internet, an incredibly large network of computers, it should be feasible to factor nondeterministically, or through brute force. However, the problem is to do so discreetly, without the nodes on the network learning the problem or the input.

3. A pharmaceutical company has finished running a large clinical trial and has collected data that need to be analyzed. The data are sensitive and the company does not want that data to become public prematurely, but it wishes to use a large public network in order to process the data. Moreover, not only are the data private, some of the analysis algorithms may be proprietary as well. The company needs a way to distribute the computation on an insecure network while making neither the data nor the algorithms public.

The above scenarios will each result in a complex distributed software system. It has been shown that such systems are most effectively approached from an architectural perspective (e.g., [81]). In particular, software architectural *styles* present generic design solutions that can be applied to problems with shared characteristics.

I propose to create a software architectural style that allows distributing problems over a large network in a discreet, fault-tolerant, and scalable manner. To that end, I will rely on the theoretical study of self-assembly and a formal model of crystal growth, called the tile assembly model [106]. This model is Turing universal, thus it can compute all the functions that a traditional computer program can. Systems in this model show remarkable fault tolerance, self-regeneration, distribution of information among components, and scalability, and a software architecture that implements the rules of such systems should inherit these properties.

The tile architectural style can be evaluated theoretically, using mathematical analysis of the architecture, and empirically, using a tile style-based system on a large network solving an NP-complete problem.

In this chapter, I will discuss software engineering work related to building large distributed systems. I will overview software architectures and their role in the design and implementation of large systems and then cover several categories of large distributed systems, such as grid computing, and systems that ask personal computers to donate their CPU time to computation. Then, I will discuss theoretical work on how much help one can expect to get from other computational resources, both in the realm of classical and quantum computing. Finally, I will overview work on secure multy-party computation.

5.1 Software Architectures

As in all engineering fields, as one builds larger and more complex systems, it becomes impossible to keep all levels of design in mind at all times. The most effective way of engineering large systems is breaking them up into subsystems, engineering those, perhaps by recursively breaking those up into smaller systems, and then abstracting away the details of the underlying systems to combine them into a large product. It has even been argued that large and complex systems cannot be efficiently built without a high-level design that describes the elements from which the system is built, the types and modes of interaction between them, and patterns of their composition [95]. Software architecture has been identified as an important part of building almost all large systems [81].

A standard software architecture consists of a description of the *components* of a system, which includes the data the components handle and the procedures they execute on that data, *connectors* that allow interaction between those components, and configurations that describe how the components come together to form the system [95]. A single architecture can have several views, each emphasizing different aspects of that configuration and its underlying system.

The particular goal of software architectures is to facilitate the development of large systems from smaller systems, encouraging reuse of previously engineered blocks and concepts. When used correctly, architectures can lead to the ability to use components designed by other architects, for

purposes far from their original intent, in varying environments, and under different interaction protocols.

Systems that have a poor underlying software architecture can be disastrous, while a good one helps to ensure the system's key properties, such as performance, reliability, portability, scalability, and interoperability [81].

Architectures provide engineers with a high-level conceptual understanding of a system, creating clear representations of the system, while abstracting away design, implementation, and deployment details until a time when those details become relevant. The goal of the architecture is to equip the engineer with the necessary conceptual tools and abstractions, before forcing her to resort to using software development tools.

One abstraction is the architectural style. A style is a set of design rules that identify the kinds of building blocks that may be used to compose a system, together with the local or global constraints on the composition [95]. Styles encapsulate the best design practices and successful system organizations [17, 72]. They have also been considered as an economical way of developing architecture-based systems [79]. Several architectural styles have been in use for a number of years, including client-server, pipe and filter, blackboard [95], and model-view-controller [66]. Other styles have emerged in the last decade from the research in software architectures, including C2 [101], GenVoca [17], and REST [52].

Several overviews of architectural styles exist [52,62,95]. The preliminary classification of styles given in [94] distinguishes between them on the basis of control and data interactions, as well as the types of analysis relevant to each style.

Software architecture can be used to "force" a software system to conform to certain rules, thus resulting in some desired properties. For example, mandating that two components communicate via implicit invocation can result in systems that are more easily evolvable. However, it is also possible to provide desired system properties as an emergent behavior of the architecture without forcing restrictions on the system designer. For example, Mikic-Rakic et al. [77] have argued that for a system to be self-healing, the system must be self-observant and alter its behavior in hostile environments. However, in my proposed architectural style, the system will exhibit properties of self-healing naturally, without observing or altering its behavior. Similarly, Devanbu et al. [47] have argued that security, a crucial property of most modern software systems, may be implemented in the connectors mediating the interactions among the system's components. Accordingly, my architectural style allows for security in the connectors; however, discreetness, one aspect of security, is an *emergent* property of the style.

While there are several definitions of architectural styles (e.g., [16, 41, 95]), I directly leverage Mikic-Rakic et al.'s [77] definition, in formulating the tile style. They have argued that an architectural style can be described along five dimensions: external structure, topology rules, behavior, interaction, and data flow [77]. External structure describes the "outside view" of the components in the architectural style; topology rules describe the allowed paths of interaction between those components; behavior describes the components' internal function and state; interaction captures the collaboration between the components; and data flow specifies the structure of the data exchanged by the components. I will follow this scheme in defining the tile architectural style.

There are several software architectural styles, certain aspects of which I will use in designing my tile architectural style. I now briefly discuss these styles and their uses.

5.1.1 Client-Server Architectural Style

A client-server architectural style has been in use for years [95]. It classifies the components of the system as either clients or servers. The clients may present the servers with queries, and the

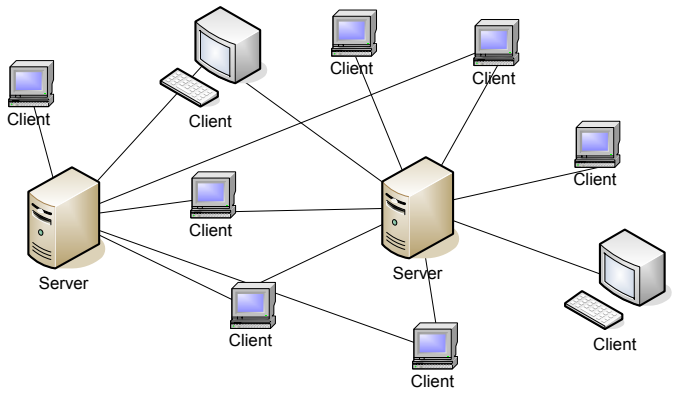


Figure 5.1: The client-server architectural style. Some components of a system are designated as clients and may query the components designated as servers. The servers answer those queries.

servers are charged with answering those queries. Figure 5.1 shows a schematic of the client-server architectural style. In this schematic, there are two servers and nine clients.

In its purest form, the client-server architectural style keeps no state about the clients at the servers. Each query must contain all the information necessary to be answered by a server. This way, the resulting architecture is most scalable because the servers do not have to remember the clients' state and can serve the most clients, one query at a time. Some systems may break this requirement and allow the servers to keep some state about its clients, for example, to minimize future network traffic (if the components happen to be distributed on a network). This architectural style allows the data to be distributed on a few components and the clients to be lightweight. All the data are kept at the servers, and the clients need only to be able to ask the servers and keep the current user's state, but do not need to be able to perform much computation on their own.

Note that the client-server style is limited by the number of queries the servers can handle. In large networks with few servers and many clients, the servers may become overloaded.

5.1.2 REST Architectural Style

The REST (Representational State Transfer) architectural style allows the transfer of client state between resources. Among the most successful systems that uses the REST architectural style is the World Wide Web [52]. Using the World Wide Web as an example, a client may access a resource, such as a URL, which returns a representation of that resource, such as a webpage. A client may follow a hyperlink on that webpage to another resource, thus transferring the state to a new state.

Styles can be combined. For example, components in the REST architectural style may be implemented as components from the client-server style. In such a view of the architecture, it is possible to describe heavy servers and light servers as well as heavy clients and light clients. A heavy component includes a lot of functionality, while a light one has only minimal functionality. Thus servers may deviate from the standard client-server style and store some or even a lot of state about their clients, becoming heavier and taking some of the load for remembering the state off the clients. Alternatively, a heavier client may reduce the load on the server, resulting in a more scalable system that allows the server to handle more clients.

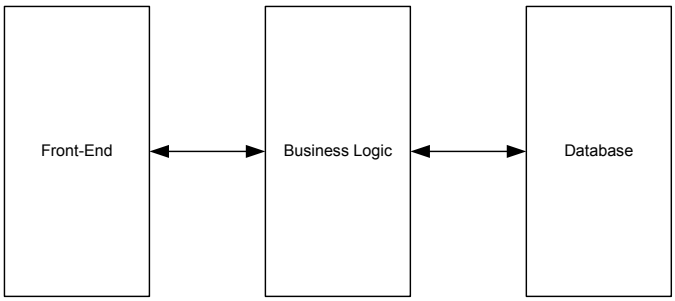


Figure 5.2: The three-tier architectural pattern. There are three major components of the system, the front-end, the business logic, and the database. Each component only talks to the component next to it, so the front-end cannot interact directly with the database. This architectural pattern is commonly used by Internet retailers.

5.1.3 Three-Tier Architectural Pattern

The three-tier architectural pattern is a popular pattern [95] among Internet retailers who use this architecture for business. Figure 5.2 shows a schematic of a three-tier architectural pattern. An example of a system that uses this patern is an airline ticket reservation system. This system has a front-end component, such as a web site that users may use to search for tickets or a travel agent interface, a business logic component that manages the ticket searches, and a database component that stores information of available and purchased tickets, past customers, etc.

Another example of combining styles, and in this case a pattern, is the airline ticket reservation system that may consider implementing the interaction between the business logic component and the database component using the client-server architectural style.

5.1.4 Peer-to-Peer Architectural Style

The peer-to-peer architectural style [102] is perhaps the most relevant to my work. This style allows for a highly decentralized network, which can be modeled as a swarm of nodes, none of which have more power on the network than others.

The peer-to-peer architectural style's main advantage over the client-server style is that it is even more scalable. This style diminishes some components' roles as servers, and makes all components roughly equal. The bandwidth and computational power scale nearly linearly with the number of components. In fact, early systems based on the peer-to-peer style were closely related to client-server-style systems, with a single "super node" component, with which each new component had to register [90] and from which components learn about other components [42]. Once the components know about each other, they interact directly with each other without using the super node component.

Peer-to-peer networks have been widely used by file-sharing systems (e.g., Napster, Gnutella [90], and Bittorent [42]). More recently, peer-to-peer telephony has emerged, led by Skype [15]. Skype, and some newer implementations of Bittorent, distribute the job of the super node over many components, using the same peer-to-peer style to organize them. Figure 5.4 shows a schematic of a system with distributed super nodes.

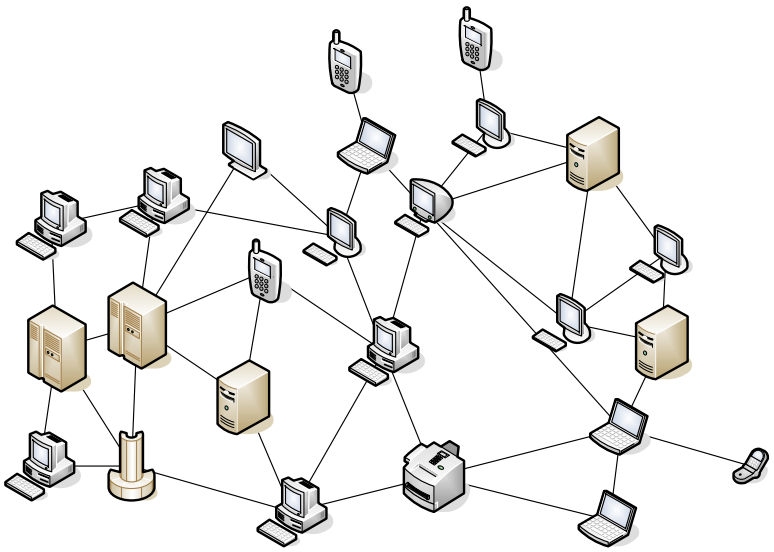


Figure 5.3: The peer-to-peer architectural style. In this architectural style, every component has roughly the same power, resulting in a highly decentralized system. Components may leave the network at any time, and new components may join the network.

5.1.5 Publish/Subscribe Architectural Style

The publish/subscribe architectural style allows the organization of components that have resources and components that need access to those resources [36]. Every component that needs access to a resource *subscribes* by letting the component that provides that resource know. When there is a change in that resource's status, such as becoming available, the component responsible for that resource *publishes* the status change and that information gets delivered to the subscribing components. Often, publish/subscribe architectures use intermediary agents called *brokers* to manage the subscriptions and publications.

Figure 5.5 shows an example of a publish/subscribe architectural style. There are three components, two of which subscribe to each of the two available resources. Brokers manage those subscriptions and forward the desired data to the subscribed components.

5.2 Grid Computing

Grid computing, or grid-based computing, is a paradigm with a goal similar to mine: to distribute computation over a large number of nodes. Grid computing usually does not concern itself with distributing the computation in a discreet manner, but only with distributing it efficiently and in a scalable manner. The work on grid systems in the realm of security has focused on authentication of the individual machines, and not discreetness. Grid computing can be divided into two areas: computational grids and data grids [76]. The former concerns itself with distributing computational load over a network, while the latter focuses on delivering large volumes of data to a distributed user base. The goals of the computational grids are closely related to those of my work. The overview of grid computing I present here follows closely a grid computing review from [75].

The most widely used solution in the area of computational grids is the Globus Toolkit [54]. Globus is an open-source middleware framework for constructing and deploying grid-based software systems. It combines a middleware transport layer (reified in the form of the Simple Object

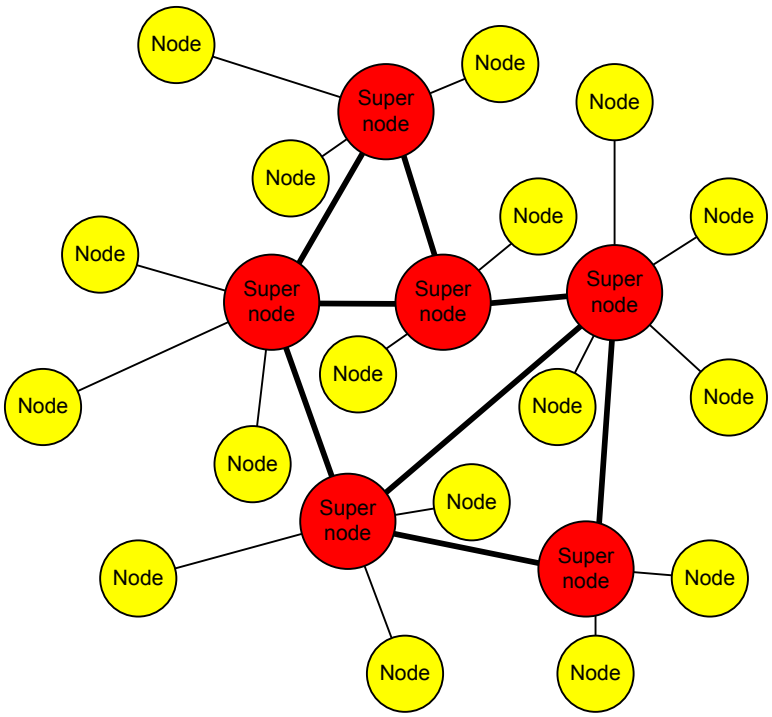


Figure 5.4: The peer-to-peer architectural style with distributed super node components. The super nodes, which components use to join the system, know about parts of the system but not the entire system. This architectural style is more scalable than one with a single log-in component.

Access Protocol (SOAP) [61]), a suite of grid-services and protocols (e.g., Grid Resource Allocation Management or GRAM [54], GridFTP [54], and so on) and a web services-based implementation infrastructure for constructing and deploying grid-based software systems using various programming languages, including Java, C++, and Perl. While Globus is primarily used by the academic and research communities, IBM, Sun, and Microsoft have begun to adopt Globus as well [56].

Globus organizes the sharing of computing, data, and resources among network nodes, which is the basic goal of grid computing. However, while Globus goes a long way in achieving the basic goal of grid computing: the establishment of virtual organizations sharing computing, data, metadata, and security resources, its adoption and use across a more widespread family of software systems and environments would likely be improved by the inclusion of some salient features; among them are (1) integration of architecture-based software development, which has been shown to facilitate and improve large-scale, distributed software construction, and (2) the decoupling of Globus protocols and services, such as GridFTP and GRAM, from their heavyweight origins, File Transfer Protocol (FTP) and Lightweight Directory Access Protocol (LDAP), respectively. The inclusion of these features would likely allow Globus to become applicable to larger problems. Particularly related to my work, Globus offers a number of security facilities, but does not explore the issues of distributing computation to its nodes in a discreet manner.

Alchemi [10], an alternate grid framework, is based on the Microsoft .NET platform and allows developers to aggregate the processing power of many computers into virtual computers. Alchemi is designed for deployment on personal computers, similar to the work I discuss in Section 5.3. The computation takes place only when the personal computer is idle, an approach that could be employed by my systems.

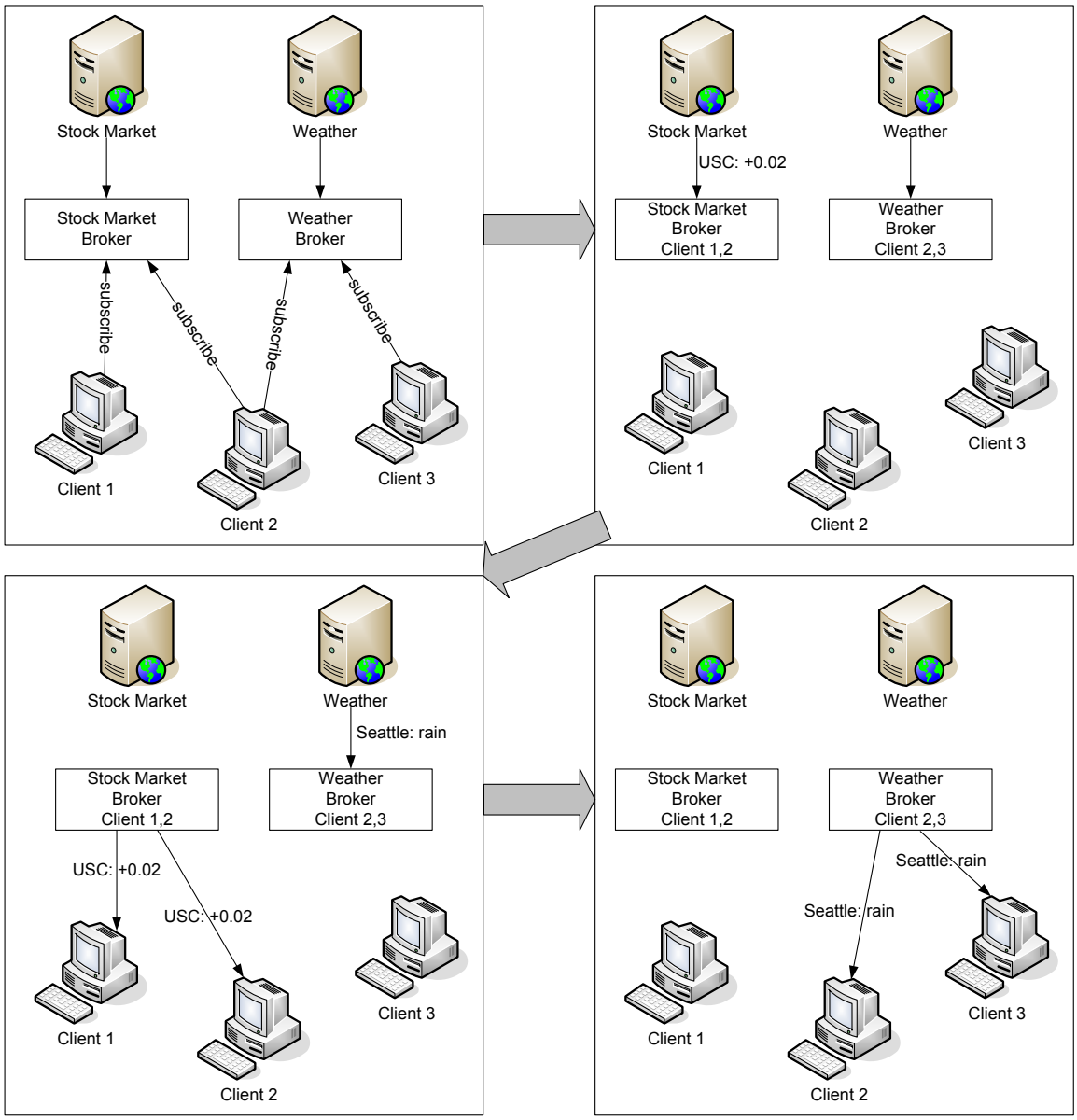


Figure 5.5: The publish/subscribe architectural style allows for components to subscribe to resources made available by other components through the help of brokers.

The peer-to-peer architectural style, some aspects of which I use in developing my systems, influenced a grid framework called JXTA [73]. JXTA has a layered architecture that abstracts away low-level protocols and services, such as host discovery, data sharing, and security. Here, just as with Globus, the computation is not distributed across the network in a discreet way.

5.3 Nondiscreet Distributed Computational Systems

The growth of the Internet in the 1990s has made it possible for researchers to use public computers to distribute computation to willing hosts. A barrage of software designed to solve computationally intensive problems has emerged to take advantage of this phenomenon, enticing users to devote their computers' otherwise idle cycles to some academically or otherwise worthy cause. Three related systems that concentrate on distributed computation are SETI@home, Folding@Home, and Rosetta@home.

SETI@home (SETI stands for Search for Extraterrestrial Intelligence, a scientific area with the goal of detecting intelligent life outside Earth) is a distributed system, run out of the University of California at Berkeley, that processes radio signals headed toward Earth from outer space. Radio telescopes collect these radio signals from outer space. Some believe that finding a radio signal that is not known to occur naturally would indicate the existence of intelligent life on other planets. Originally, the radio signals were processed on special-purpose supercomputers, but in 1995, David Gedye proposed distributing the computation over the Internet [65, 92].

The SETI problem is to scan the radio signals received by a radio telescope to identify if there are any narrow-bandwidth signals. Claiming that if we were to try to communicate with other planets, we would use narrow-bandwidth signals (also our own loose communication signals are narrow-bandwidth signals), it makes sense to search for such signals in the broad-range signals captured by the telescopes. Searching for signals in a data stream is an easily distributed task: the data are broken down into frequency bands by using fast Fourier transforms and individual computers can examine the perhaps slightly overlapping bands independently. Further, data from different telescopes, and thus from different parts of the sky can be examined virtually independently. The SETI@home system distributes a small portion of the signal to each of its volunteer computers around the Internet, and each computer executes the exact same algorithm on the entire input it receives and reports the results back to the central node. As of October 2000, the project had processed nearly 39 terabytes of data by 2.4 million users, with up to a million of them working at any given time [65].

Figure 5.6 shows a schematic of the SETI@home network, indicating one possible break up of the data over the nodes. Apart from their inability to decide on a common capitalization scheme for the project names, the creators of the systems I describe below (Folding@Home and Rosetta@home) use almost identical network setups to the SETI@home setup. Of course the data, and thus the types of data ranges, are different for these different projects.

Folding@Home [53, 70], run out of Stanford University, and Rosetta@home [86], run out of the University of Washington, are similar distributed systems to SETI@home, but they attempt to solve instances of the protein-folding problem. The protein-folding problem is, given the sequence of amino acids that make up a given protein, to determine that protein's secondary structure. This problem, like the SETI problem, is easily parallelizable, and while it only takes microseconds in our cells for a protein to fold, we know of no fast (polynomial-time) algorithms to determine the secondary structure. In fact, the protein-folding problem has been shown to be NP-complete [20]. Each possible fold of the protein can be examined for stability using the known properties of hydrophilicity and hydrophobicity, and its free energy can be determined in polynomial time,

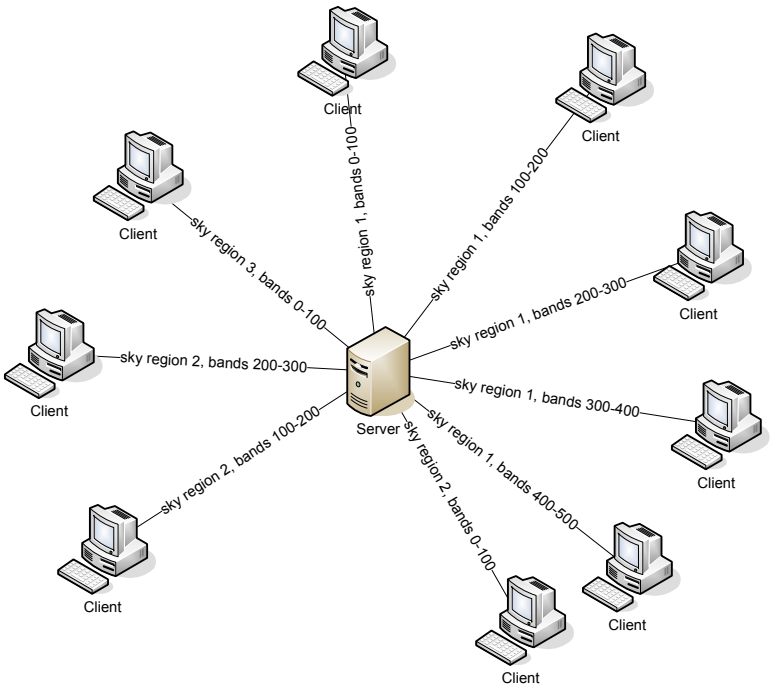


Figure 5.6: SETI@home distributes computation over a large network. There is a single log-in server that decides which data to send to each node. The nodes perform the necessary computations and send their results back to the server.

independently of such analysis of other potential folds. Thus it is possible to distribute such computation over independent nodes. The Folding@Home and Rosetta@home projects have in some way been a success, resulting in numerous scientific publications regarding protein secondary structures.

A number of other similar projects exist, including ones that work to factor numbers. Distributing computation, for some algorithms, is quite powerful, and for some special algorithms, one can even execute those algorithms more than m times faster on m machines than on a single machine [96] (due to memory sharing and caching). The problems I have called highly parallelizable (e.g., NP-complete problems) are of the class of problems for which we know algorithms, most of which can achieve linear speed up via distribution.

While the work presented in this section displays the ability to distribute certain types of computation over large networks, that computation is neither discreet nor scalable. The individual nodes know the entire algorithm and portions of the data large enough to be used as an input to that algorithm (e.g., entire narrow radio bands as oppose to small portions of individual frequencies). Further, a single client computer distributes all the work to the individual nodes, so as the network grows, the client computer would have to communicate with more computers. Discreetness and scalability are two properties my distributed systems will posses.

5.4 Discreet Nondistributed Computational Systems

Some research [1, 50, 51] has gone into devising strategies for getting computational help without disclosing the input to the computation. This work has focused on asking a single other computer for help, and thus does not directly extend to distributed systems. However, the theoretical results

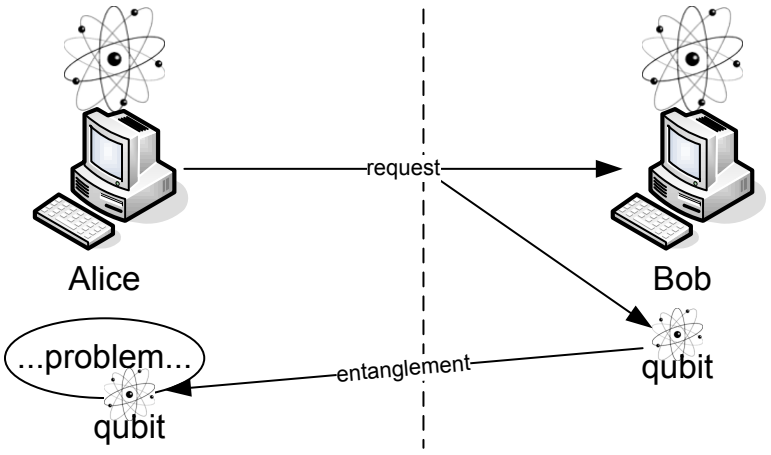


Figure 5.7: In the quantum computing setting, Alice, who cannot or chooses not to do universal quantum computation, may get help from Bob, who can do universal quantum computation, without revealing the algorithm or the input to Bob via the mechanism of entanglement. In the classic computing setting, getting such help is not possible [40].

related to the amount of information one can hide when asking a single computer for help is directly related to the amount of information one can hide from an entire network when asking it for help.

Childs [40] has shown that in classical computing, for NP-complete problems (as well as many others), it is not possible to ask a single entity for help and receive useful feedback without disclosing at least some information about the entire input and the problem one is trying to solve. He also points out that it may be possible to disguise some fairly insignificant information. For example, in getting help in solving 3-SAT, one could randomly negate every occurrence of some of the variables in a Boolean formula and get help solving the modified formula without disclosing the original. However, except for the final Boolean assignment, the rest of the information, such as the relationship and frequency of the variables, is preserved. Childs [40] has also shown that using quantum computing, one can get help solving computationally intensive problems without sharing specific input or problem data by using entanglement. Figure 5.7 illustrates, on a high level, the approach in getting help in a quantum setting. Alice, who cannot perform universal quantum computation, asks Bob, who can, for help.

My work lies in the middle of these two findings. I stick to classical computing and do not rely on quantum computing, which, to date, is not a viable technique in practice. I also accept the finding that it is impossible to get help without sharing your data with the entire network, but devise a mechanism that allows distributing the data and algorithm around the network such that every sufficiently small group of nodes knows neither the algorithm nor the input.

Some proposals for distributing computational problems over the Internet point out that solving NP-complete problems has the extra advantage that a small certificate could be provided with the answer allowing the client to check the correctness of the computation quickly [18]. My approach can take advantage of this observation to help the client ensure that the solution is correct. What I will show, however, is that such a certificate may be an overkill because the fault tolerance properties of my systems will allow the user to set an arbitrarily high probability that no malicious or faulty node may cause the outcome of the computation to be incorrect.

5.5 Secure Multi-party Computation

After Diffie and Hellman posed the problem of secure multi-party computation [48], Yao suggested and solved the millionaire problem: two millionaires wish to learn who of them is the richest without sharing their worth [110]. In [111], Yao generalized the problem to what is now known as multi-party computation protocols. In these protocols, there are n parties, each with a piece of private information. Their goal is for every party to gain access to the value of some function of those pieces of information without sharing the private data.

Yao's garbled circuit protocol is a well-known result in multi-party computation protocols. It states that for every polynomial-time computable function on two inputs, there exists a polynomial-time protocol that allows two parties, each with access to one of the inputs, to compute the value of that function on those inputs without either party learning the other's input. The result follows from the fact that for every polynomial-time computable function, there exists a polynomial-size circuit, and each party can compute a garbled form of this circuit [110].

The first party, for every wire w in the circuit, chooses two random strings $S_{w,0}$ and $S_{w,1}$. These random strings represent the values of 0 and 1, on the wire w. The first party then computes a garbled truth table for each gate, using a symmetric encryption scheme E and each wire's random string values. For example, if wires 1 and 2 are the inputs to an *AND* gate and wire 3 is its output, the truth table's entry for the value 0 on wire 1 and value 1 on wire 2 would be $E_{S_{1,0}}\left(E_{S_{2,1}}\left(S_{3,0}\right)\right)$. To prevent the second party from learning the logic of the truth table, the first party permutes the table entries and then sends these garbled truth tables to the second party, along with its garbled inputs via oblivious transfer. Oblivious transfer is a primitive operation of sending information while remaining oblivious as to what is being received and is critical to the correctness of almost all multi-party computation protocols. The second party can then use its own inputs and the garbled truth tables to compute the garbled value of the circuit.

If both parties are honest, Yao's garbled circuit protocol is guaranteed to compute the correct answer without sharing the private data [71]. In a model with malicious parties, Yao's protocol is not secure; however, it can be made secure with the use of zero-knowledge compilers [59, 60].

In the generalized problem, with n parties, Ben-Or et al. [19] have shown that a function on n inputs, each of which is on a different host, can be computed by a complete network on n processors such that if no fault occurs, no set of size $t < \frac{n}{2}$ processors gets any additional information, other than the function value, and that if Byzantine faults may occur, no set of size $t < \frac{n}{3}$ can either disrupt the computation or get additional information.

While the secure multi-party computation protocols try to solve a problem that in many ways is similar to mine, it is also in many ways orthogonal. These protocols must deal with data being distributed on different nodes, whereas inputs to my computations are known on a single node. Further, as these protocols require complete communication networks on n nodes, they are unlikely to scale to Internet-sized networks — a primary goal of my work. At times, however, my systems will need pairs of nodes to compute simple functions of their private data without sharing that data. In those cases, Yao's garbled circuit will provide the necessary infrastructure.

Multi-party computation protocols have led to solutions in the fields of zero-knowledge proofs, distributed voting, private bidding and auctions, sharing of signature or decryption functions, private information retrieval, secure poker play, and many others [21, 45, 93, 97]. Goldreich provides an excellent in-depth discussion of secure multi-party computation protocols in [58].

Chapter 6

The Tile Architectural Style

In this chapter, I will explain how the tile architectural style allows distributing computation over a network. The work presented in this chapter appears in my papers [25,31–33]. To distinguish tile systems and software systems based on the tile style, I will refer to the former as "tile assemblies" in this and the next chapters.

A tile-style architecture is based on a tile assembly. The components of the architecture are instantiations of the tile types. While a system based on such an architecture will have a large number of components, there is a comparatively smaller number of different *types* of components (e.g., 8 types for adding and 64 types for solving 3-SAT). Nodes on the network will contain these components, and components that are adjacent in a crystal can recruit other components to attach by sampling nodes until they find one whose side labels, or interfaces, match. Note that many tile components can run on a single physical node, as I will further elaborate below.

In addition to defining the tile types, a tile assembly also directs the architecture how to encode the input to the computation. The input consists of a seed, which is a small connected collection of tiles, such as the clear tiles along the right and bottom edges in Figure 4.39. The seed replicates on the network to create many copies.

For exposition purposes, I will first summarize the tile style using the characterization proposed by Mikic-Rakic et al. [77]. I will then elaborate on the key facets of the style in the remainder of this chapter. Each component's externally visible *structure* comprises the four *interfaces*, i.e., side labels (shown on the sides of each tile). The *topology* is a 2-D grid of components that allows neighbors on the grid to interact. The components exhibit four *behaviors*: identifying other nodes on the network that deploy particular types of tiles (as described in Section 6.2.2), replicating to create copies of themselves on other nodes (as described in Section 6.2.3), cooperating with neighbors to recruit suitable new components to attach (as described in Section 6.2.4), and reporting the solution to the user. The *interaction* consists of exchanging data about a component's sides in order to recruit. Finally, the *data flow* is limited to the components' interfaces, allowing components to inform their neighbors of their side interfaces, but provide no other information.

6.1 Using the Tile Style

A user who wishes to solve a computationally intensive and easily parallelizable problem, e.g., an NP problem, and has access to a large unreliable network, may use the tile style to design a system to solve her problem. The user has two options: (1) use the tile style to design her own architecture based on the tile assembly that solves her particular problem, or (2) reduce her problem to 3-SAT, using a standard polynomial-time reduction [98], and use the 3-SAT tile assembly (or other such tile assembly, e.g., the *SubsetSum* assembly).

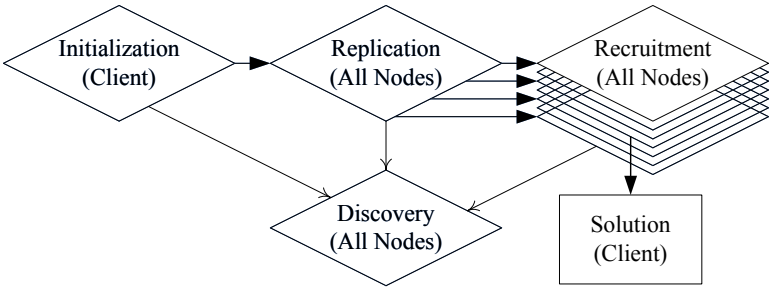


Figure 6.1: Overview of tile style node operations.

The two options have clear analogs in the traditional computing realm. When presented with a computational problem, one can either (1) write a program to solve that problem or (2) relate the problem to one for which an established program already exists. Option 1 can be expensive and error-prone and will require intricate design and verification, though may result in the most efficient program possible. Option 2 can save significant time and effort and provides the user with more confidence in the final solution. For these reasons, I advocate the second option. At the same time, I acknowledge that using the same tile assembly for all computation may compromise discreetness, because an adversary may discover that the user always uses the 3-SAT tile-style system. A compromise is the design of several tile assemblies, each to solve a different NP-complete problem, and selecting among these assembly-based systems at random every time the user is presented with an NP problem to solve. This approach allows the user to reuse existing systems while retaining some discreetness of the algorithm. Only the algorithm's discreetness can be compromised by reusing systems and not the input's; a user who is more concerned with the discreetness of the algorithm may choose option 1. For simplicity, I restrict my discussion to 3-SAT in this chapter.

Whichever tile assembly the user chooses will serve as the template for the architecture, with the assembly's tile types defining the types of components. Part of a tile assembly is the description of seeds that encode inputs (e.g., the clear tiles in Figure 4.39 are the seed for that 3-SAT computation). The user sets up a seed to encode her input and assigns computers, or nodes, on the network to deploy the seed tile components. Once the initialization is complete, starting with the seed tile components, adjacent components deploy on other network nodes to represent fitting components and eventually produce the solution. The solution tile components (e.g., the ✓ component for the 3-SAT assembly) then report their state to the user. Figure 6.1 summarizes the steps nodes take to perform the computation. I will now elaborate on those steps.

6.2 Node Operations

Systems built using the tile style are self-assembling software systems. That is, the nodes participating in the computation are in some ways self-sufficient and require no central controlling entity. In this section, I describe four operations performed by the nodes that allow self-assembly: initiation, discovery, replication, and recruitment.

6.2.1 Initiating Computation

When a client computer wishes to initiate a computation, it creates a tile type map. A tile type map is a mapping from a large set of numbers (e.g., all 128-bit IP addresses) to tile types. It determines the type of tile components a computer with a given IP address deploys. The tile type

map breaks up the set of numbers into k roughly equal-sized regions, where k is the number of types of tiles in the tile assembly. For the 3-*SAT* example from Section 4.4, there are 64 different tile types, so the tile type map would divide the set of all 128-bit numbers into 64 regions of size 2^{122}. The exact mapping is not important, but it may be convenient to simply map the first 2^{122} numbers to tile type 1, the next 2^{122} to tile type 2, and so on. Thus the tile type map maps each IP address (or other unique identifier), passed through a hash function, to a tile type. The hash function ensures an even distribution of tile types. The size of the tile type map, which will later be sent to all the nodes on the network, is small. For a tile assembly with k tile types, the tile type map is k 128-bit numbers. Below, I describe how nodes learn to what tile types their IPs map.

The tile type map will uniformly distribute the tile types over the network, and also ensure that unless a computer can control more than a single IP address, it will not be able to learn of more than one tile type. Note that in some networks, it may not be prohibitively difficult for an adversary to have a single machine control a large number of IP addresses. If that is a concern, it is possible to use other unique identifiers, such as network card MAC addresses, that are more expensive to control in bulk.

I assume that the network is such that each computer is connected to a constant number of other computers (say p computers), distributed roughly randomly. This is a first-order approximation of the Internet, but my analysis will extend to more accurate models. Every computer may contact its neighbors directly, via their IP addresses, and may also query its neighbors for their lists of neighbors, thus discovering more nodes. A number of my algorithms are designed specifically to work on such a distributed network, on which no single node knows a large portion of the network. For completeness, I will discuss how these algorithms will perform on more highly connected networks, and how the algorithms could be improved for such networks.

For each tile type, the client computer sends the tile type map and that tile type's description to at least one node whose IP maps to that tile type. A tile type's description consists of the four tile component interfaces, which can be described using just a few bits. For discussion purposes, I will overestimate and use 32 bits to describe the four interfaces. To accomplish this initial task, the client first contacts its neighbors and sends them the tile type map and their appropriate tile type descriptions. If no neighbors map to a certain tile type, the client requests from the neighbors their neighbor lists using a small request (a few bits), and receives back the IPs of more neighbors, and so on, until at least one node of every tile type knows its type. For a system with k tile types, the client will have to "collect" at least one of each of the k types, which means that it will have to sample $\Theta(k \log k)$ computers. Specifically, after sampling $ck \log k$ computers, where $c < 2$, the probability that a node has not found all k tile types is $\frac{1}{2}$. After sampling $2ck \log k$ nodes, that probability is $\left(\frac{1}{2}\right)^2$. The probability continues to drop exponentially, so after sampling $20ck \log k$, the probability of not having located a node of each needed type is only $\left(\frac{1}{2}\right)^{20} < 10^{-6}$. For the 3-*SAT* example, $20ck \log k$ will be fewer than 15,000 packets, which for typical UDP packets amounts to only 750 kilobytes. This analysis, and the value of c, come from the solution to a well-known *coupon collector* problem [80].

The nodes that learn their types from the client computer propagate the information to their neighbors whose IPs map to the same tile types, and so on, until every computer on the network learns the type of tile component that computer will deploy. Thus every computer receives the tile type map and the description of its own tile type. Each computer might receive its tile type information and the tile type map several times, up to as many times as it has neighbors. I assumed earlier that every computer in my network of N nodes has only a few (p) neighbors. Thus, the total amount of data sent by each node is $\Theta(p)$, because roughly $\frac{1}{k}$ of a node's p neighbors will have to be sent the $128k$ bits $\left(\frac{128kp}{k} = \Theta(p)\right)$. If the network is a highly connected one, e.g., every node

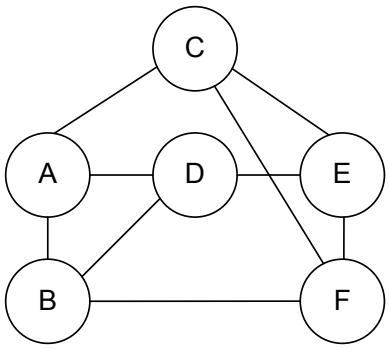


Figure 6.2: A network with six nodes. I assume that every node in my underlying network has p neighbors (here $p = 3$).

is connected to every other node, the amount of data sent by each node may be a concern, as it would be $\Theta(N)$, where N is the number of nodes on the network. However, the problem on such a network becomes far simpler. Each computer could send the information to only p of its neighbors (resulting in $\Theta(p)$ data sent from each node), or the client computer could send all the information itself (resulting in $\Theta(Nk)$ data sent from a single node and no data from others). The diameter of a network of N nodes with randomly distributed connections is $\Theta(\log N)$ [80], so the tile type map and the tile types will propagate through the network in $\Theta(\log N)$ time.

I now help clarify the procedure for delivering the tile type map and tile types to each node on the network with the use of an example. Figure 6.2 shows an example network on six nodes, A, B, C, D, E, and F. Note that in this network, $p = 3$, meaning that every node is connected to 3 other nodes. I assumed earlier that the underlying network is such that every node has p neighbors, but should note that it is straightforward to convert most networks into these special networks: nodes that have too few neighbors can discover more via their neighbors, and nodes that have too many can simply ignore those. I assume such a regular network only for ease of analysis; my algorithms work on more typical networks with nodes of various degrees.

If node A is the client, it decides on the tile type map, and then tells its neighbors (B, C, and D) the tile type map and descriptions of their particular tile types. Suppose that in this small example there are four tile types, and the tile type map chosen by A maps A itself to tile type 1, B and D to tile type 2, C to tile type 3, and E and F to tile type 4. A informs B, C, and D of their tile types, but knows of no nodes that map to tile type 4. Then A requests from B its neighbors, and B returns A, D, and F. A recognizes that F is a newly discovered node and records it on the list of its neighbors. It also uses the tile type map to identify F as a tile type 4 node and informs it of the tile type map and the description of tile type 4. Node A is now satisfied, and each of the nodes B, C, D, and F check their neighbors for ones of the same tile type as themselves. Nodes B and D recognize that they are both of type 2 and send each other the tile type map and the description of tile type 2. Note that this information is redundant because both nodes already know it. Node F recognizes that it and E are both of type 4 and sends E the tile type map and the description of tile type 4. At this point, all nodes are finished sending information and know the tile type map and the descriptions of their particular tile types.

Until now, I have ignored the case of a network with fewer nodes than the number of types of tiles. If the network is that small, it is possible to create multiple virtual nodes on each machine and proceed as before, though a single physical node will have knowledge of more than one tile type, compromising discreetness. In the limit, for a network with a single node, it has been analytically shown that discreetness is not possible [40]. The tile style is intended to be used on

large networks, and while it can be made to work on small ones as well, the discreetness properties may be compromised. I will return to this issue in the discussion of my empirical evaluation in Section 7.2.

6.2.2 Discovery

After the computation has been initiated, nodes will perform a *discovery* operation, which will be used in replication and recruitment, discussed below. The discovery operation, given a tile type, returns a *uniformly-random* IP of some computer deploying tile components of that type, meaning that if a node performs this operation repeatedly, the frequencies of the IP addresses it returns asymptotically approach the uniform distribution. Thus, every suitable computer has an equal chance of being returned, in the long run. My algorithm for discovery will guarantee uniform randomness, which in turn will guarantee that all nodes on the network perform a similar amount of computation. The algorithm will use a property of random walks to ensure uniform randomness.

In order to quickly return the IP address of a computer that deploys tile components of a certain type, each node will keep a table, called the node table, of three IP addresses of each component type. I explain the reason for this below. For 3-SAT, the size of this table will be $64 \times 3 = 192$ IPs. The table contains only an identifier for each tile type, and not the details about the side labels, thus preserving the discreetness of the algorithm. The preprocessing necessary to create the node table is simple: first a node fills in the table with all its neighbors and then gets help from neighbors (by requesting their neighbor lists). The analysis of this procedure is identical to the analysis of the client computer finding nodes that deploy tile components of each type in Section 6.2.1; this preprocessing procedure will take $\Theta(k \log k)$ time per node (happening in parallel for each node), for k different tile types. The amount of data sent by each node is limited to $\Theta(k \log k)$ packets. For 3-SAT's $k = 64$, that is fewer than 300 packets, which for typical UDP packets amounts to only 15 kilobytes.

After the preprocessing, when queried for the IP of a computer that deploys tile components of a given type, the node performs two steps: (1) it selects one of the three entries in the node table for that tile type, at random, and (2) it replaces its list of three entries in the table with the selected node's corresponding three entries. The reason for the replacement is that the selection of IPs should emulate a random walk on the node graph [80]. The request packet only needs to contain the tile type (e.g., a 32-bit number) and the answer packet must contain three IPs (three 128-bit numbers). This entire procedure takes $\Theta(1)$ time.

I now help clarify the preprocessing and discovery operations with the use of an example. Suppose the small six node network in Figure 6.2 is part of a larger network, but happens to be the connectivity of six nodes that all map to the same tile type. In creating its node table, A first checks its neighbors B, C, and D, and records them in the three slots for that tile type. A's node table (for that tile type) is now complete, but had A not found three valid nodes to fill its table, it would expand its neighbor list by querying one of its neighbors for its neighbors, until it discovered a sufficiently large portion of the network. B follows the same procedure as A and creates a node table and records its neighbors A, D, and F as the three nodes deploying the same tile type. When A needs a node of that type later (for reasons discussed below), it selects a random node from its three entries. Suppose it selects B. A then replaces its node table entries with B's entries (A, D, F). Note that it is possible for a node to store itself on its node table.

Theorem 6.2.1 *On a network on N nodes, after filling only $\Theta(\log N)$ requests for an IP of a computer that deploys a certain tile type using the above-outlined procedure, the probability of each valid IP being returned is uniformly distributed.*

Proof: Because the node table keeps independent lists of three nodes of each type, it is sufficient to prove the theorem for a single tile component type. Consider the directed graph G formed by representing every node as a vertex with three outgoing edges to the vertices representing the nodes on the node table. Now consider a sequence of nodes derived by the above-outlined procedure of picking a random node from the three entries, and replacing those three entries with that node's entries. That sequence corresponds to a random walk on G. From [80], a random walk on G mixes rapidly, which means that if selecting nodes via this random walk after $\Theta(\log N)$ steps, the probability of getting the IP of each node becomes proportional to that node's in degree. Thus on a uniform graph, every IP is equally likely to be returned. ■

I have discussed how to convert a random network into one such that each node has exactly three neighbors. Again I emphasize that this simplification is made to aid my analysis, and in fact the random walk theorem from [80] holds for all graphs with nodes having three or more neighbors, so this result is directly applicable to all reasonable distributed networks.

I again make a note of the implications of using the tile style on small networks. A small network may not have enough nodes to deploy every tile component type on at least three machines, thus making the creation of the node table impossible, unless I allow initial duplication of nodes. The reason for the more intricate algorithm, such as ours, is to ensure that every node that deploys each tile component type is used uniformly. If there are two or fewer nodes that deploy a given tile component type, returning those with uniform probability is trivial. Thus for small networks, discovering the entire network does not pose computational difficulty, and selecting nodes uniformly randomly is trivial.

6.2.3 Replication

After the computation initiation step, the client sets up a single seed on the network, as described in Section 6.1. Each tile component knows of its neighbors. The seed tile components then replicate twice, to create two additional copies of the seed on the network. The reason for replicating twice is that after t time steps, the number of seeds on the network is $\Theta\left(2^t\right)$. Note that that are no known algorithms to solve NP-complete problems without requiring an exponential number of parallel executions, thus every fixed-size network can be overwhelmed by a large enough input. In the example from Section 4.4 with 3 variables, the algorithm would need to explore at least $2^3 = 8$ possible scenarios. A goal of the tile architectural style is to distribute the computation across many physical nodes to execute in parallel, but for a given network size and input size, one can set bounds on the number of components deployed on each physical node to prevent overloading those nodes.

To replicate, each node X uses its node table, as described in Section 6.2.2, to find another node Y on the network that deploys the same type components as itself, and sends it a replication request. A replication request consists of up to four IP addresses (four 128-bit numbers) of X's neighbors. X lets its neighbors know that Y is X's replica (by sending Y's IP to its neighbors). Those neighbors, when they replicate using this exact mechanism, will send their replicas' IPs to Y. Thus, the entire seed replicates.

After creating two copies of the seed, the tile components begin the recruitment process described in Section 6.2.4. The newly created seeds will also each replicate twice, thus creating a number of seeds exponential in time. The seeds continue to replicate and self-assemble until one of the assemblies finds the solution, at which time the client broadcasts a signal to cease computation by sending a small "STOP" packet to all its neighbors, and they forward that packet to their neighbors, and so on. As discussed above, the diameter of a large connected network of N nodes

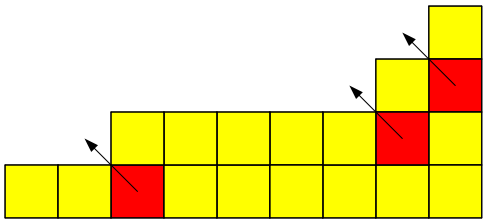


Figure 6.3: Tile components that have both a north and a west neighbor (highlighted in the diagram) can recruit new components to attach to their northwest.

with randomly distributed connections is $\Theta(\log N)$ [80], so the "STOP" message will propagate in $\Theta(\log N)$ time.

6.2.4 Recruitment

In a temperature two computational tile assembly (such as the assembly described in Section 4.4 that solves 3-SAT), a tile that has both a north and a west neighbor recruits a new tile to attach to its northwest. Figure 6.3 indicates several places in a sample crystal where tile components are ready to recruit new tiles.

A recruiting tile component X (highlighted in Figure 6.3), for each tile type, picks a node Y of that type from its node table, as described in Section 6.2.2, and sends it an attachment request. An attachment request consists of X's north neighbor's west interface and X's west neighbor's north interface. If those interfaces match Y's east and south interfaces, respectively, then Y can attach. At that point, X informs Y of the IPs of its two new neighbors, and those neighbors of Y's IP. Note that X can perform this operation without ever learning its neighbors' interfaces by using Yao's garbled protocol [111], which is crucial for discreetness.

In the example 3-SAT system, the successful crystal recruits 310 tile components (non-clear tiles in Figure 4.39). An unsuccessful crystal, which I discuss further in Sections 6.3 and 7.1.2 can recruit fewer, but no more than 310 tiles.

6.3 Answering *3-SAT* in the Negative

A crystal that finds the proper truth assignment that satisfies the Boolean formula reports the success to the client computer. Since for NP-complete problems the answer is always "yes" or "no," the notification is only a few bits. Deciding that there is no satisfying assignment is more difficult. No crystal can claim to have found the proof that no such assignment exists. Rather, the absence of assemblies that have found such an assignment stands to provide some certainty that it does not exist. Because for an input on n variables there are 2^n possible assignments, if 2^n assemblies find no suitable assignment, then the client knows there does not exist such an assignment with probability at least $\left(1 - e^{-1}\right)$. After exploring $m \cdot 2^n$ assemblies, the probability grows to at least $(1 - e^{-m})$. Thus as time grows linearly, the probability of error diminishes exponentially. Given the network size and bandwidth, it is possible to determine how long one must wait to get the probability of an error arbitrarily low.

In the example from Section 4.4 with 3 variables, the probability of exploring $2^3 = 8$ assemblies and not finding the solution is no more than e^{-1}. After exploring 80 assemblies, that probability drops to $e^{-10} < 0.00005$. Note that no crystal can be larger than 310 tiles, so 80 assemblies would require fewer than 25 thousand tile components. Because the tile components are lightweight (each

one is far smaller than 1 KiB), there is little reason why even a single computer could not deploy that many components.

Chapter 7

Analysis and Evaluation of the Tile Architectural Style

In this chapter, I will present my analysis of discreetness, scalability, and fault and adversary tolerance of tile-style-based systems. The work presented in this chapter appears in my papers [25, 31–33]. Section 7.1 will cover the theoretical analysis, and Section 7.2 will cover the empirical analysis.

7.1 Theoretical Analysis of the Tile Architectural Style

Underneath the tile architectural style lies a formal mathematical model of self-assembly, the tile assembly model. That formal foundation allows me to reason formally about systems built using the tile style. In particular, I can evaluate the systems' discreetness, scalability or efficiency, and fault and adversary tolerance.

7.1.1 Discreetness

I call a distributed system *discreet* if, with high probability, for all time, for all nodes on the network, each node can discover neither the algorithm the network is executing nor the entire input to that algorithm.

I make five arguments in showing that tile style-based systems are discreet. The first three deal with the discreetness of the algorithm: (1) given one tile type of a tile assembly one cannot determine any information about the function that assembly computes, (2) it is difficult to control enough computers to learn all the tile types, and (3) even if an adversary controls enough computers to learn all the tile types, that adversary cannot determine the algorithm the assembly is executing, in the general case. The last two arguments deal with the discreetness of the input: (4) given a single tile in a crystal, it is not possible to learn any information about the input and (5) controlling enough computers to learn the entire input is prohibitively hard on a large public network. I will then discuss what kind of information one may be able to learn about the input and argue that it can be greatly limited by properly encoding the input.

1. First, I show that given a single tile type in an assembly, it is not possible to determine what function that assembly computes.

   Theorem 7.1.1 *Let $\mathbb{S}$ be a tile assembly with the tileset T. For all $t \in T$, knowing t gives no information about the function $\mathbb{S}$ computes.*

   Proof: The labels of the sides of a tile t are elements of a finite alphabet. The particular labels themselves have no meaning other than matching or not matching labels on other tiles'

sides. Thus, given a bijection from labels to labels, it is possible to relabel the sides of all the tiles according to that bijection without changing the function the assembly computes. Thus given a tile t from assembly $\mathbb{S}$, for all computable functions f, there exists a tile assembly $\mathbb{S}'$ such that $\mathbb{S}'$ computes f and t is in the tileset of both $\mathbb{S}$ and $\mathbb{S}'$. Therefore, for every computable function, there exists a tile assembly with t in its tileset and it is not possible to deduce any information about the function given only t. ■

Since the tile architectural style is designed so that each node is only aware of a single tile type, it follows that no single node on the network may know the algorithm the network is computing. Note that at the start of the computation, the client computer creates a tile type map that maps each computer, using its IP address or another unique identifier, to a single tile type and that nodes only disclose the description of the tile types to other nodes that map to the same tile type. In both recruiting and replicating, the nodes never learn their neighbors' interfaces, and thus cannot learn other tile types.

2. I now investigate how many nodes an adversary must control on a network in order to learn all the tile types. Since the algorithm the client uses at the start of the computation to assign one tile type to each node employs a hash function to ensure that this assignment is made randomly, with a uniform probability distribution over all tile types, no node can trick its way into being a specific tile type. Thus, based on the coupon collector problem, one must control at least $\Theta(k \log k)$ unique identifiers to be able to learn the k tiles types [80].

   Thus in order to learn all 64 tile types of a 3-SAT solving tile assembly, an adversary must control several hundred computers on the network.

3. In general, given a computer program, it is not possible to tell what that program does, or even if it will halt. A similar statement can be made about tile assemblies, as it is not possible to determine what function a tile assembly computes, in the general case.

   Theorem 7.1.2 *Let $\mathbb{S}$ be a tile assembly and f be a function. Then in the general case, one cannot determine whether $\mathbb{S}$ computes f.*

   Proof: Winfree has shown that temperature two tile assemblies (such as the ones I use to compute functions) are Turing universal, and further, how to design a tile assembly that simulates a given Turing machine. Assume that for each tile assembly, one can determine whether that tile assembly computes some function f. Then, given a Turing machine, one could create a tile assembly to simulate that Turing machine, thus gaining the ability to learn whether the Turing machine computes f. But Rice's Theorem [98] says that given a Turing machine, one cannot tell whether it computes some function f. That is a contradiction, so it is not possible to know whether a tile assembly computes any given function, in the general case. ■

4. For a tile assembly, such as the one solving 3-SAT, each tile type encodes no more than one bit of the input. A special tile encodes the solution, but has no knowledge of the input. If every tile component in the crystal were deployed by a different node on the network, it would be trivial to argue that the computation was discreet. However, since a single node on the network may deploy several tile components of the same type, the argument relies on the fact that each component is unaware of its location in the crystal, and thus does not know the location of the bits of the input. Thus every node on the network may be aware of either some bits of the input or the solution, but not both, and a node cannot use the partial information

it has about the bits of the input to recompose that input in its entirety. That is, the nodes can learn information such as "there is at least one 0 bit in the input," but no more.

5. It is clear that if an adversary controls or can see the internal data of the entire network, that adversary can learn the input to the problem. However, the likelihood of such a scenario on a very large public network is exceptionally low. An interesting question becomes how much of the network an adversary must control in order to learn the n-bit input.

   Theorem 7.1.3 *Let c be the fraction of the network that an adversary has compromised, let s be the number of seeds deployed during a computation, and let n be the number of bits (tiles) in an input. Then the probability that the compromised computers contain an entire input seed to a tile-style system is $1-(1-c^n)^s$.*

   Proof: If an adversary controls a c fraction of the network nodes, then for each tile in a seed, the adversary has a probability c of controlling it. Thus for a given n-bit seed, distributed independently on the nodes, the adversary has probability c^n of controlling all the nodes that deploy the tiles in the seed, and thus the probability that the seed is not entirely controlled is $1-c^n$. Since there are s independent seeds deployed, the probability that none of them are entirely controlled is $(1-c^n)^s$. Finally, the probability that the adversary controls at least one seed is $1-(1-c^n)^s$. ■

   Let me examine a sample scenario. Suppose I deploy a tile-style system on a network of $2^{17} \approx$ 100,000 machines to solve a 17-variable 3-SAT problem. Let me also suppose a powerful adversary has gained control of 12.5% of that network. In order to solve this problem, the system will need to deploy no more than 2^{17} seeds, thus the adversary will be able to reconstruct the seed with probability $1-\left(1-2^{-51}\right)^{2^{17}} < 10^{-10}$. Note that as the input size increases, this probability decreases. The probabilty decays exponentially for all $c < \frac{1}{2}$ (that is, as long as the adversary controls less than one half of the network). In the above example, control of 25% of the network gives the adversary a probability of reconstructing the input below 10^{-5}, and control of 33% of the network yields a probability no greater than 10^{-3}. An adversary who controls exactly half the network has a $\frac{1}{e} \approx 37\%$ chance of learning the input, and one who controls more than half the network is very likely to be able to learn the input, which is why my technique is geared towards large public networks.

   One possible challenge to discreetness on large public networks is botnets [46]. The Internet is home to several 20,000-machine botnets, and botnets as large as 100,000 machines exist, although they are extremely expensive to maintain [46]. A randomly selected portion of the Internet is highly unlikely to be dominated by a botnet (e.g., have more than 1% of its computers be part of any one particular botnet). However, if a network is composed of self-selecting machines, one must take care to ensure the network is large enough that no one botnet controls a large fraction of the nodes. A likely scenario on a network such as the Internet is that no adversary will be able to control a constant fraction of that network. If c is replaced with a function inversely proportional to the size of the network (consistant with an adversary controlling a constant number of machines regardless of the size of the network), the probability of learning the input will decay exponentially as the network grows.

The analysis of Theorem 7.1.3 dictates that the probability of reconstructing the entire input is low. It is interesting to ask how much information about the input one can discover. While an adversary who controls some portion of the network is unlikely to reconstruct the whole input,

that adversary may be able to collect enough information to determine, with some certainty, the frequency with which 0 and 1 bits occur in the input. Similarly, that adversary may be able to collect pairs (and triplets, etc.) of tiles deployed on compromised nodes to determine the frequencies of 00, 01, 10, and 11 sequences in the input. The analysis of Theorem 7.1.3 also applies to reconstructing constant fractions of the input (such as half or 10%) and thus the probability of learning a contant fraction of the input also decays exponentially. It is somewhat easier to reconstruct constant-sized tuples such as pairs and triplets. However, the amount of information one can learn from the frequencies of these tuples is limited by the informational entropy [44] within the frequencies. Thus one can encode inputs to maximize that entropy by making sure that the numbers of small equal-length subsequences (e.g., 00, 01, 10, and 11) in the input are equal, thus greatly limiting the information that an adversary can obtain from such subsequences. The field of DNA sequencing contains some work on the related problem of reconstructing long sequences from short overlapping subsequences. Typically, these reconstruction algorithms require the subsequences to be of length around 100 [37,82], much longer than the sequences likely to be recovered by an adversary who has compromised less than half of the network.

Each tile component in the 3-SAT system handles at most a single bit of the input. Theoretically, this is sufficient for solving NP-complete problems; however, practically, handling more than a single bit of data at a time would amortize some of the cost of communication. Thus each tile component can be made to represent several bits. This transformation would result in a trade-off between discreetness and efficiency, as faster computation would reveal larger segments of the input to each node.

7.1.2 Efficiency and Scalability

The tile style is aimed at large networks and for solving computationally intensive problems. For small inputs, the overhead of using the tile style may dominate the benefit of parallelization.

When a client wishes to solve a highly parallelizable problem and needs discreetness, she may choose to do so on her own single computer, perhaps on a small private network of trustworthy computers, or using the tile style on a large insecure network. The disadvantage of computing on a network is that, by various estimates, remote communication can be 100 to 1000 times slower than local communication. The computation is further slowed down by the fact that tiles may have to perform more basic operations than a program that is not restricted by the components of the tile style. For instance, the example in Figure 4.39 uses 352 tiles, whereas a simpler program could simply try all possible truth assignments to the three variables and check whether at least one of the three literals in each of the three clauses is $TRUE$, using $2^3 \cdot 3 \cdot 3 = 72$ operations. The upside is that the tile style distributes the work over the network, parallelizing computation. For small inputs, there is only so much parallelization of which the tile style can take advantage; however, for large inputs, the style can significantly speed up computation. For a 3-SAT problem with seventeen variables and 100 clauses, a single computer would have to perform $2^{17} \cdot 100 \cdot 3 \approx 3.9 \cdot 10^7$ operations. On the other hand, a network of 1.3×10^5 nodes, such as an existing TeraGrid computational grid network, would deploy no more than 352 components on each node, resulting in a system that takes between $1000 \cdot \frac{352}{2^{17} \cdot 100 \cdot 3} \approx 9.0 \times 10^{-3}$ as much time as a single computer (assuming network communication is 1000 times slower than local communication) and 9.0×10^{-4} as much time (assuming it is 100 times slower). In other words, the resulting tile style-based distributed system computes over a hundred to over a thousand times faster than the custom-built single-computer system.

Suppose now that a user needs to solve a 3-SAT problem with 47 variables and 180 clauses. A single $2.5GHz$ computer would require over one year to solve that problem, whereas a tile-style

system deployed on TeraGrid would take between eight hours and four days. Note that this larger problem does not benefit from any more potential parallelism than the 17-variable problem because TeraGrid has roughly 2^{17} nodes. If the underlying network were larger, as the networks I target are likely to be, the tile-style system would perform even faster. In Section 7.1.1, I discussed a way to further bring down the overhead of the tile style significantly, by having each tile represent more than just a single bit of data. One could represent 32-bit or even 1024-bit words, speeding up the system by 32 and 1024 times, respectively, assuming that sending a 1-bit packet on the network takes roughly as much time as a 1024-bit packet, which is a reasonable assumption for small packet sizes.

Based on such analysis, I assert that systems built using the tile style will be highly scalable. Every tile component in a crystal requires a constant amount of communication to attach. Once attached, it can only participate in recruiting of two other tiles. Thus, the communication associated with each tile is bounded. The recruiting process cannot take more than k steps, for a system with k computational tile types, as described in Section 6.2.4. For the example 3-SAT tile system, a tile has to sample no more than 64 other nodes to either find a tile that can attach or know that no such tile exists. Thus, no node representing a component will need more than a constant amount of communication originating from it to recruit an attachment.

7.1.3 Fault and Adversary Tolerance

The tile style abstracts the properties of fault and adversary tolerance away from the computational aspects of the system. My goal is a system that does not require the designer to put effort into making the system fault- or adversary-tolerant. Once designed using the tile style, a system's architecture allows the designer to specify two parameters: the fraction of nodes on the network that may be faulty or malicious, and the acceptable rate of system failure. For example, a designer may say that $\frac{1}{4}$ of the network is malicious but the client can only accept a failure rate of 2^{-10}. The system then automatically self-adapts to produce the proper failure rates, without the designer, or anyone else, having to write code. This approach requires an estimate of an upper bound on the fraction of faulty or malicious nodes. Calculating such a bound depends on the nature of the system, as the faults may be related to hardware or environmental factors. While the bound needs only to be an estimate, the guarantees provided by the tile style will only be as accurate as this estimate.

First, I will give the definition of fault and adversary tolerance in Section 7.1.3.1. I will then provide an intuitive explanation of how the tile style achieves the fault and adversary tolerance properties in Section 7.1.3.2. The intuition is not entirely accurate, but goes a long way toward making the technique more understandable. Then, I will more formally explain the tile style's emergent tolerance and how it leverages work on error correction within the tile assembly model in Section 7.1.3.3. Finally, I will discuss the types of faults and attacks the tile style tolerates in Section 7.1.3.4.

7.1.3.1 Definition of Tolerance

I call a distributed system *fault-tolerant* (*adversary-tolerant*) if, given a fraction of the network nodes failing (acting in a malicious fashion), the probability of successful computation can still be bounded arbitrarily close to 1 without paying an exponential cost in speed. In particular, systems designed using the tile style allow the architect to slow down the system linearly while lowering the failure rates exponentially.

7.1.3.2 Intuition Behind Tolerance

Computer science has long used redundancy to induce fault tolerance in systems. The basic idea is that if a component has a probability p of failure, if the failures are independent, two components performing the same task have only a p^2 probability of failure. The above calculation depends heavily on the assumption of independence of failures. In my realm, this assumption implies that a node's failure must have no dependence on another node's failure, which can be achieved by ensuring that either the implementation of the components or the attacks on them differ.

If the failure is a *crash* failure (component either returns the correct answer or crashes returning no answer), then k components performing the same tasks reduces the probability of failure of the entire system to p^k. A similar result can be achieved for *Byzantine* failures (component, possibly colluding with other components, either returns the correct answer or an incorrect answer but no indication that it has failed) by employing a system of voting. In this case, $\Theta(k)$ components are necessary to perform the same task to reduce the probability of failure of the entire system to p^k. In essence, the tile style can employ a redundancy approach. In the "basic" tile style, each tile component of the assembly is deployed on some network node. If that node is malicious, or faulty, it may attach incorrectly or attempt to recruit other tile components incorrectly and break the entire computation. It is possible to have multiple nodes be responsible for deploying each tile, checking each others' computations, and thus correcting crash errors and voting to avoid Byzantine errors.

7.1.3.3 Error Correction for Tolerance

While direct redundancy is one way to accomplish the goal of fault tolerance, it is possible to use nodes more wisely to correct each other by exploring the field of error correction in self-assembly. I have discussed some of this work in Section 2.6, and I will in particular refer to work by Winfree et al. [108] described in Figures 2.8 and 2.9. They have shown that given a tile assembly and a certain fraction of malicious tiles, one can increase the number of tiles (e.g., break each tile into a 2×2 grid and represent it with four tiles), and bring the probability of error exponentially close to 0. For example, increasing the number of tiles by a constant factor c, from k to ck in [108], (or from k to ck^2 in [83]) would bring the previous error probability of ϵ to $\epsilon^{\Theta(c)}$. In some sense, they have developed "smart redundancy."

By applying tile assembly model error correction techniques to the tile style, I have begun a new exploration of these techniques because software faults differ from the faults traditionally examined in the study of self-assembly. This exploration is leading to ways of classifying the techniques, as well as modifying them to be robust to a larger class of faults.

Several other researchers have done work on error correction in the tile assembly model [39,83,99, 107], looking at increasing accuracy without increasing the assembly size, and healing catastrophic errors such as the death of a large portion of the assembly at once. This work may be useful for errors such as large portions of the network failing and massive attacks or failures that target geographically close tiles, but the exact exploration of that area is beyond the scope of my work.

7.1.3.4 Fault Scenario Analysis

The error correction work proposed in [83, 108] applies directly to tile assemblies. As the tile assembly model is a model of biological systems, these error correction techniques safeguard against errors commonly found in biology (e.g., what are called nucleation and growth errors). A tile assembly that employs such error correction techniques may be resilient to mismatched tiles attaching in the wrong places, which is an accurate model of incorrect molecular attachment. However, the

tile architectural style allows computers to represent tiles, and the types of errors that may occur are likely to differ from the biological variety. While it is not too difficult to show that these error correction techniques, applied to the tile architectural style, would correct some particular types of faults, the key aspect of this work is showing that neither faulty nor malicious nodes on a network can break these particular error correction techniques. (It may be worthwhile to note that some other tile assembly model error correction techniques described in [39, 99, 107] would not be as helpful for the tile style as the ones I have selected.)

By applying the error correction mechanism from [83,108] to the tile style, I foresee countering such attacks as nodes pretending to host foreign tile components, attempting to report incorrect computation results, and overwhelming the client or other nodes on the network with traffic. Theorem 7.1.4 summarizes that result.

Theorem 7.1.4 *Let $\mathbb{T}$ be a computational tile assembly that computes the function f, and let $\mathbb{S}$ be the software system with the architecture derived by the tile style from $\mathbb{T}$. Let ρ be the probability of $\mathbb{S}$ failing on network N with some faulty and some malicious nodes. Then for all $j \geq 2$, it is possible to design a tile assembly $\mathbb{T}'$ that also computes f such that the software system with the architecture derived by the tile style from $\mathbb{T}'$ has the probability of failing on network N of ρ^j.*

The proof of Theorem 7.1.4 is a combination of proofs of theorems in [83,108], the fact that each tile component does not know its location in the assembly, and the observation that when a tile is represented by a $k \times k$ block of tiles, as long as there are fewer than k improper tiles, that block either assembles completely correctly or will not complete. Theorem 7.1.4 is a stronger statement than the theorems in [83, 108] because those theorems assume only the presence of tiles allowed by the tile assembly, whereas I assume all possible tiles, as well as the presence of tiles conspiring together to break the assembly. While I outline the proof here, it remains part of my future work to formally prove Theorem 7.1.4, as well as empirically verify this result.

The implication of Theorem 7.1.4 is that given a software system designed using the tile style, and a network with faulty or malicious nodes, if the system has a failure rate of 25%, by using smart redundancy as described in Figure 2.9(b) with $k = 5$, the probability of the system failing decreases to 2^{-10}, while slowing down the execution speed only by a factor of 5, because $\left(\frac{1}{4}\right)^5 = 2^{-10}$.

While this discussion appears promising for implementing fault and adversary tolerance in tile-style systems, the details of the implementation remain future work and I discuss then briefly in Chapter 8.

7.2 Empirical Analysis of the Tile Architectural Style

To empirically demonstrate the utility, efficiency, scalability, and discreetness of the tile style, I created an implementation of a distributed software system whose architectural style is faithful to the descriptions of the algorithms of the tile style. To do this, I have leveraged Prism-MW [74], a Java-based middleware platform intended specifically for style-driven implementation of software architectures in highly-distributed and resource-constrained environments. Prism-MW is an accurate fit for style-driven design, but it was neither intended nor previously applied to solving computationally complex, such as NP-complete, problems. It provided explicit implementation-level constructs for declaring components, interfaces, interactions, network communication, etc., as well as the ability to encode stylistic constraints as first-class middleware-level constructs.

I do not focus on the details of the implementation because it closely reflects the design outlined in Chapter 6. The implementation is lightweight, contains under 3000 lines of code, and the compiled binaries are only 78KiB. A host may deploy multiple Prism-MW Architecture objects,

each of which represents a virtual network node, thus allowing the simulation of large logical networks on comparatively smaller physical networks; when deploying a tile style-based system on a large network, each physical node is intended to run a single Prism-MW Architecture object. The Prism-MW Architecture forms a "sandbox," or a virtual machine, within which all of the code deployed on a remote node executes. The use of system resources on each participating hardware host is hence restricted and can be released at any time. The tile components are deployed inside the Architecture objects and perform their functionality outlined in Chapter 6 via their interfaces, which are implemented as Prism-MW Ports (which can, in turn, be either local or distribution-enabled).

The user who is interested in solving an NP-complete problem only has to provide a description of the set of tiles for that problem and the input to the computation (the tiles for solving two NP-complete problems, *SubsetSum* and 3-*SAT*, are included). The implementation takes this information and automates the remaining steps of building a distributed tile style-based system.

I have performed a number of empirical measurements of my tile-style implementation in solving *SubsetSum* and 3-*SAT* problems. My evaluation was challenged by the number of networked computers under my complete control. It would have been relatively easy for me to install the tile style implementation on, and "borrow" execution cycles from, a large number of computers, such as those available in the USC Center for Systems and Software Engineering Laboratory. However, while the tile style implementation is in fact intended to be used on computers busy with unrelated tasks, variations in computer load would have prevented me from distinguishing the impact of that load on my evaluation results. Note that when demonstrating the tile style's functionality on small problems and small networks, I am penalized by the network communication cost, but do not benefit from the high parallelization available on large networks. Thus it makes sense to look at the relative speeds of execution as the network grows, rather than the absolute speeds.

I have used a heterogeneous network of 11 Pentium 4 1.5GHz nodes with 512MiB of RAM, some running Windows XP and others Windows 2000. Some had fresh operating systems installed for my experiments, while others had been in service for years with no antivirus protection, thus more accurately representing some nodes on the Internet. One node had a wired connection to the Internet while the others were on wireless networks. As an example of using my tile-style implementation, Figure 7.1 shows the mean completion times for solving a 21- and a 32-bit *SubsetSum* problem on networks comprising between 2 and 11 nodes. Because the executions are nondeterministic, I performed each computation five times and then calculated the mean completion time.

These preliminary empirical results verify that the tile architectural style can be used to solve NP-complete problems, that my algorithms result in correct computations, and that as the underlying network grows, the computation time decreases roughly inversely proportionally to the network size, which agrees with intuition as more computation can happen in parallel on larger networks. For example, execution on 6 nodes was roughly 1.6 times faster than execution on 3 nodes ($\frac{52}{36}$ for 21-bit input and $\frac{110}{60}$ for 32-bit input), and execution on 10 nodes was roughly 1.6 times faster than execution on 5 nodes ($\frac{43}{27}$ for 21-bit input and $\frac{71}{48}$ for 32-bit input). To further verify the system's correctness, I executed the system with inputs that returned a negative answer. As expected, the system executed indefinitely. It is possible to implement a progress bar to report the confidence in that negative answer; that confidence would grow exponentially quickly, as described in Section 6.3.

In my largest experiments to date, I have deployed the tile-style system on an 186-node subset of USC's Pentium 4 Xeon 3GHz High Performance Computing and Communications cluster. For example, when using half the cluster (93 nodes), solving a 20-variable 20-clause 3-*SAT* problem took 220 minutes, whereas the full cluster completed the same job in 116 minutes (i.e., 1.9 times faster). Again, this confirmed my estimated expected running times and provided confidence in my theoretical analysis. It remains future work to demonstrate the efficiency and robustness of the

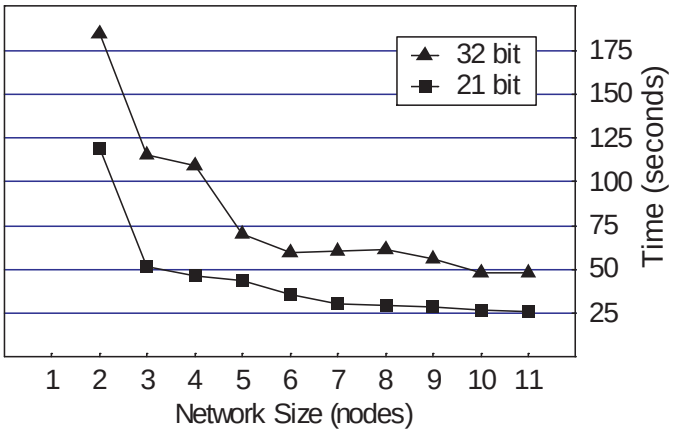


Figure 7.1: The execution time of a tile-style program solving two *SubsetSum* problems on networks of varying sizes. Each point represents the average of five executions.

tile style on systems distributed on large public networks and solving computationally intensive problems.

Chapter 8

Contributions and Future Work

In this chapter, I will summarize the contributions of my work presented in this dissertation, briefly describe several additional attempts at using self-assembly to solve interesting problems I have explored, and present future directions and variants of my research, some of which I have begun or intend to follow.

8.1 Contributions

When engineers compare biological and software systems, the former come out ahead in the majority of dimensions. For example, the human body is far more complex, better suited to deal with faulty components, more resistant to malicious agents such as viruses, and more adaptive to environmental changes than your favorite operating system. Thus it follows that we, the engineers, may be able to build better software systems than the ones we build today by borrowing technologies from nature and injecting them into our system design process.

In this dissertation, I have developed an architectural style for building large distributed software systems that allow large networks, such as the Internet, to solve computationally intensive problems. This architectural style, the tile style, is based on nature's system of crystal growth, and thus inherits some of nature's dependability, fault and adversary tolerance, scalability, and security. The tile style allows one to distribute computation onto a large network in a way that guarantees that unless someone controls the majority of that network, they cannot learn the private data within the computation or force the computation to fail. The resulting software systems allow a large number of processors to come together to help solve NP-complete problems by parallelizing the computation. These systems are highly scalable, capable of dealing with faulty and malicious nodes, and are discreet since every sufficiently small group of nodes knows neither the problem nor the data.

As the tile style is based on a formal mathematical model, I have been able to verify and prove the properties of discreetness, fault and adversary tolerance, and scalability for software systems built using the tile style. I have further created an implementation of such a system and distributed computations on a network to empirically demonstrate the tile style's utility.

Along the way to developing the tile style, I explored the tile assembly model, a formal mathematical model of self-assembly. I contributed to that field by defining a notion of computation within that model and developing systems that compute functions such as adding, multiplying, factoring, and solving the NP-complete problems *SubsetSum* and *SAT*. For each system, I proved that the system computes its intended function, analyzed the running time and tileset size (two measures previously identified as important in tile systems [4]) and probability of successful computation (a measure I identified as important for nondeterministic computation). I have also developed a set

of techniques for designing tile systems to solve computational and other problems that can be applied by others.

8.2 Connecting Self-Assembly and System Design

Along the path to defining my particular area of contribution, I explored several subfields of self-assembly and molecular computation and made minor but important additions to those subfields. These contributions are directly related to self-assembly and molecular computation. I include here a brief summary of these contributions and references to publications describing them in more detail.

8.2.1 DNA Complexes

The theory of self-assembly is driven largely by the scientists who have developed molecular implementations of the tile assembly model. One of the original DNA complexes was the double crossover [55], and it has been used in several implementations of the tile assembly model [12, 88]. Together with collaborators, I worked on designing two other DNA complexes, a triangle complex [38], and a double double crossover complex [84], as well as generalizing the ideas in these complexes to create paradigms for assembling other complexes [30]. This work goes toward implementing tile systems using DNA, thus possibly being able to solve much larger problems within a single test tube than we can solve on computers today.

8.2.2 Amorphous Computing

Artificial intelligence researchers have explored a notion of amorphous computing [2], known by many other names, for example, swarm robotics, active self-assembly [11], and paintable computing [35]. Most of these researchers strive to show that simple components can come together to exhibit complex behavior. My work in this area, in collaboration with Dustin Reishus, includes tile assembly systems with components that are in some ways as simple as they can be, coming together to perform many of the jobs that the more complex components have been shown to do, such as finding and repairing paths on graphs and forming shapes [34]. This work may lead to biologically inspired algorithms for robotics or sensor networks that are more powerful or are cheaper and easier to implement than existing algorithms.

8.2.3 DNA Logic Gates

I have explored models other than the tile assembly model that allow DNA to compute complex functions. Together with Manoj Gopalkrisnan, I have developed a series of binary logic gates that use exclusively DNA. Theoretically, such gates can be used in diagnosis and treatment of cancer cell-by-cell, allowing one to treat only those cells affected by cancer, and not the healthy ones [29]. This approach to computation can be described as a silicon computing-based biological system, and is somewhat opposite to the approach of the tile style. Implementing logic gates using DNA is largely less efficient, less reliable, and less accurate than doing so in silicon; however, the power of this approach lies in being able to deliver these small and partially reliable machines to places silicon computers would have a hard time reaching, such as individual cells within living human tissues.

8.3 Future Work

In this section, I will describe the work that remains to be done in the areas of the tile assembly model and the tile architectural style, as well as some future directions for biologically inspired software.

8.3.1 Implementing Fast Algorithms

I have described how the tile style, based on tile assembly model systems such as the ones from Chapter 4, can implement the most direct algorithms for solving NP-complete problems. These algorithms execute via $\Theta(2^n)$ parallel assemblies, for inputs of size n. The implication of that fact is that if the size of the problem is far larger than the network, the system will require $\Theta(2^n)$ time to find the solution. While we are unaware of subexponential-time algorithms to solve NP-complete problems, there are algorithms that perform in exponential time but with a base smaller than 2. Woeginger [109] provides a fairly complete survey of such algorithms, organized by the different techniques they employ. I will briefly describe a few of these algorithms for *SAT* and *SubsetSum* and argue that they can be implemented using the tile assembly model, and then converted into distributed software systems either directly via the tile style or with minor modifications to the tile style. While my arguments here will be intuitive, brief, and incomplete, it remains future work to develop the tile systems and tile style modifications necessary to implement such algorithms into large discreet distributed software systems.

For the discussion of the algorithms, I will need to define the $O^\star$ notation, which is similar to the O notation but ignores not only constant factors but also polynomial factors. Thus I will say $O^\star(m(x))$ for a complexity of the form $O(m(x) \cdot poly(x))$. The justification for this notation is that the exponential growth of $m(x)$ will dominate all polynomial factors for large x. For example, if f is a function such that $f(x) = O(1.4142^x x^4)$, then I write $f(x) = O^\star(1.4142^x)$. Note that the exponential term dominates and one could say $f(x) = O(1.4143^x)$ and forgo the $O^\star$ notation altogether; however, that would not most accurately describe the functions.

8.3.1.1 Pruning the Search Tree

One technique used to improve the time complexity of algorithms is the pruning technique. For example, the algorithm described in Section 4.4 for the 3-*SAT* problem explores each of the possible 2^n truth assignments to the n variables. A more intricate algorithm can explore a subset of those assignments by noting the following fact: if the Boolean formula contains a clause $(x_1 \vee \neg x_2 \vee x_3)$, then the algorithm need not explore any of the $2^{(n-3)}$ assignments with $x_1 = x_3 = FALSE$ and $x_2 = TRUE$ because this clause would not be satisfied by any of those assignments. This type of approach results in an $O^\star(1.8393^n)$ algorithm [109]. Slight improvements in the branching step can result in an $O^\star(1.6181^n)$ algorithm [78]. Using quantitative analysis of the number of resulting 2-clauses from such branching improves the time complexity to $O^\star(1.5783^n)$ [91]. The champion algorithm using this technique achieves a time complexity of $O^\star(1.4963^n)$ [67, 68]. Note that there are somewhat faster algorithms for *SAT* that use other techniques, but I now wish to briefly argue that it is possible to implement the tree pruning ideas in the tile assembly model, and thus in the tile style.

The system described in Section 4.4 nondeterminically branches on each variable's assignment. Thus it creates 2^n distinct assemblies. The seed encodes the Boolean formula in a row and the variables in a column. Each distinct assembly encodes one possible assignment in a column next to the variables and then "sweeps" that assignment across the formula to check if each clause

is satisfied. Instead of trying each possible assignment for each variable, one could make the assignments based on the clauses. For example, if the first clause is of the form $(A \vee B \vee C)$, where A, B, and C are literals, then the assignments the system should explore are:

- $A = 1$,
- $A = 0$ and $B = 1$, and
- $A = B = 0$ and $C = 1$.

It is reasonable to imagine a tile system that examines the first clause and nondeterministically chooses one of the above-outlined three options. The system then sweeps the assignment across the formula and simplifies it, abandoning the computation if any clause contains three false literals. Repeating these steps recursively would solve 3-SAT. Figure 8.1 demonstrates the preliminary ideas of how a hypothetical tile system assembly implementing this algorithm might look. While I have worked out some of the details of such a system and have ideas of how it would work, it remains future research to formally define the system and prove its correctness. One deduction that should be true is that the number of distinct nondeterministic assemblies should fall to $O^\star(1.8393^n)$ for this algorithm, and to $O^\star(1.4963^n)$ for the more intricate algorithms, reducing the time complexity of the tile-style software systems.

8.3.1.2 Data Preprocessing

The tile system described in Section 4.3 explores each possible subset of the input numbers to solve *SubsetSum*. As there are 2^n possible subsets, the tile system creates 2^n distinct assemblies. Suppose I were to split the n numbers into two equal-sized subsets and then compute the $2^{\left(\frac{n}{2}\right)}$ sums of the elements of each set. The problem would then become to check whether the sum of one element from each set of computed sums equals the target number. This algorithm performs in $O^\star\left(\left(\sqrt{2}\right)^n\right) \approx O^\star(1.4142^n)$ time. It is possible to use this technique to improve the algorithm for a similar NP-complete problem, the Exact-Hitting-Set problem, to $O^\star(1.2494^n)$ time complexity [49].

This type of preprocessing and algorithm may seem to pose a greater challenge to implement in a tile system than the search tree pruning idea. However, since the tile assembly model is universal, it should be possible to implement this algorithm in it, and my intuition says that one can do it without using too many tiles. One possibility is to nondeterministically select a subset of the first half of the numbers and add them, and then to check if there exists the proper complement in the second half. Another possibility is to abandon the notion of nondeterministic computation, and rather have a single large assembly compute all the sums of the two subsets and check if proper complements exist. If one selects the second path, the tile style will likely need to be adjusted slightly because the resulting tile system will not be a temperature two system; at the very least it will need to have some strength 2 binding domains to grow a seed that is polynomially large in the input size into an assembly that is exponentially large.

More work is needed to develop these tile systems and to adjust the tile style to work with these new systems, but my experience indicates that it is possible to implement these algorithms using a close variant of the tile style.

8.3.1.3 Other Techniques

There are numerous other techniques for solving NP-complete problems, such as local search and dynamic programming. The former technique produces an algorithm for SAT that runs in

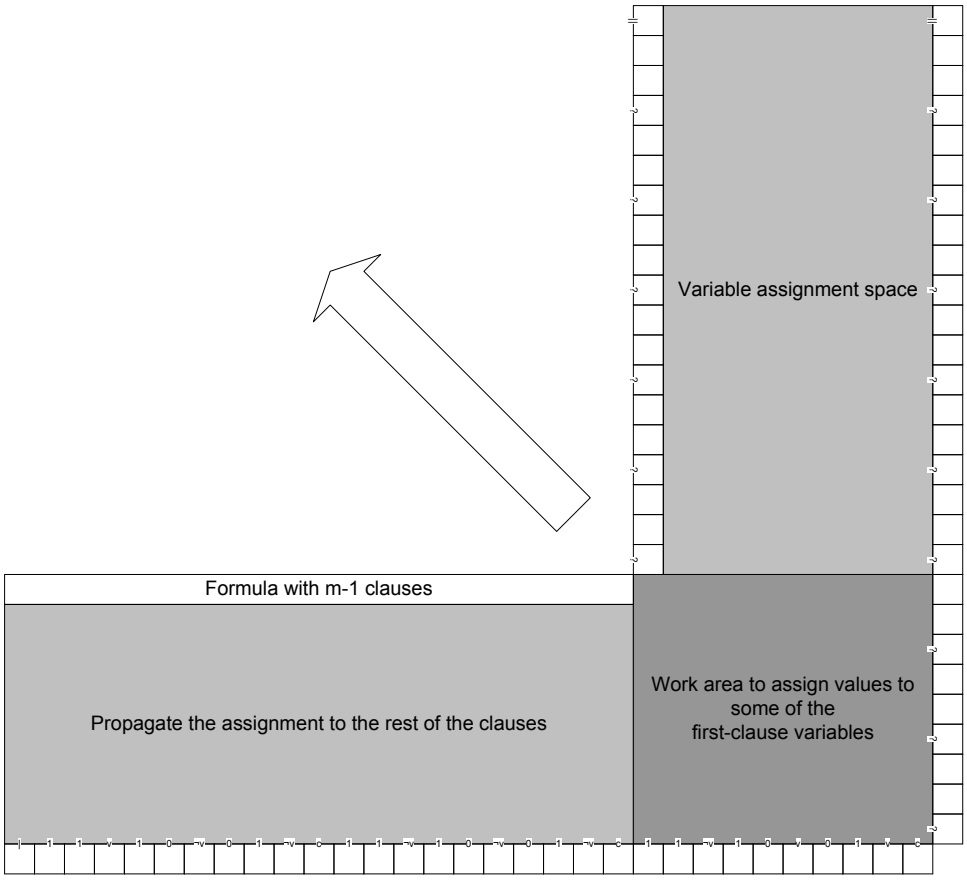


Figure 8.1: Outline of a tile system implementing a fast algorithm for 3-SAT. Here, in deciding whether the Boolean formula $\phi = (x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee x_1 \vee x_0)$ might be satisfiable, the algorithm chooses at least one of the literals in the first clause to be true and simplifies the rest of the formula before moving on to the second and later clauses. This hypothetical system could decide 3-SAT using $O^\star(1.8393^n)$ assemblies, and these ideas can be used to reduce the number of assemblies even further.

$O^\star(1.3302^n)$ [63]. The tile assembly model should be able to implement all such algorithms but the question remains as to just how efficient in terms of the size of the tileset these implementations can be. Answering that question will go a long way toward making the tile style a fierce competitor to other parallelization techniques.

8.3.2 Fault Tolerance

In Chapter 7, I spoke about fault tolerance and began to argue that the tile architectural style can be used to create fault-tolerant software systems. However, I have only scratched the surface of exploring fault tolerance in these systems. In order to implement some of the ideas presented in Chapter 7, the tile style has to be modified to allow reversible attachment of tiles (tiles are allowed to attach with a probability proportional to the attachment strength and detach with probability inversely proportional to the attachment strength). This modification will require slight changes to the algorithms involved in the tile style, and thus discreetness and scalability must be argued with respect to the new algorithms, though I envision those arguments would not change much from their current state.

Slightly alternate versions of fault tolerance mechanisms may prove to be more powerful than those I have described. In particular, the most basic 2×2 proofreading mechanism converts each tile type into four distinct tile types; however, it may be possible to simply require three tiles of a type to attach to every location, thus allowing the detection of a single error (five attachments allowing detection of two errors, and $2n + 1$ attachments allowing detection of n errors). This type of a mechanism may be implementable in a 3-D version of the tile assembly model but can be much more directly implemented in the tile style. The exact implications of this mechanism remain future work.

8.3.3 Exploring Underlying Computational Models

In creating the tile style, I have used the tile assembly model as the underlying computational model. It is possible to create similar architectural styles on top of cellular automata, Turing machines, or other universal computational models. Each computer on the network could represent a single cell in the automaton or on the tape and communicate with neighbors to compute. Such architectural styles would also likely result in scalable, fault-tolerant, and discreet software systems. To me, the tile assembly model seems to have some benefits over Turing machines because Winfree's proof that the tile assembly model is Turing-universal [105] uses tiles to simulate 1-D cellular automata that are capable of simulating Turing machines. The resulting tile systems, in effect, compute the states of the Turing machine's tape at every step in the computation. Thus at the very least, the tiles can compute functions in the same number of steps as Turing machines (and for the same reasons, 1-D cellular automata). While Turing machines, cellular automata, and tile systems are all polynomially related with respect to their running times, it is not at all clear (though entirely possible) that Turing machines can simulate tile systems in the same number of steps. I have demonstrated some tile systems that compute functions more efficiently than tile systems simulating Turing machines computing the same functions. Thus, there may be an advantage to using the tile assembly model as the underlying computational model of the tile style. However, it may prove fruitful to explore other computational models to create related but new architectural styles.

8.3.4 Other Biologically Inspired Software

Biological systems are far more complex than systems we design and build today. The human body alone has orders of magnitude more complexity than our most intricate designed systems. Further, biological systems are decentralized in such a way that allows them to benefit from built-in error correction, fault tolerance, and scalability. Despite added complexity, human beings are more resilient to failures of individual components and injections of malicious bacteria and viruses than engineered software systems are to component failure and computer virus infection. Other biological systems, for example worms and sea stars, are capable of recovering from such serious hardware failures as being cut in half (both worms and sea stars are capable of regrowing the missing pieces to form two nearly identical organisms), yet we envision neither a functioning desktop, half of which was crushed by a car, nor a machine that can recover from being installed with only half of an operating system. It follows that if we can extract certain properties of biological systems and inject them into our software design process, we may be able to build complex self-adaptive software systems. Work outlined here has developed software systems by looking to biology for inspiration. I plan to involve biological inspiration to an even greater degree and it is my long-term goal to develop an understanding of how to build software systems that function the way biological

systems do, and to design appropriate architectures, design tools, and programming tools to create such systems.

While studying biological systems is likely to have a positive effect on many different types of software systems, one of the areas that I believe can benefit most by borrowing nature's techniques is distributed Internet-sized systems. While we have some experience and knowledge in how to build software to be executed on a single processor, the notions of "programming the Internet" or "running an operating system or a virtual machine in a distributed fashion on a large network" are fairly new. Such distributed systems will likely require a great deal of collaboration, while the large size of the network is likely to require that collaboration to scale well. Further, since no single entity controls the Internet, and the nodes may join or leave the network at any time, the collaboration must be fault-tolerant and resilient to dynamic node addition, removal, and failure. Finally, if these systems are to perform important computations, they must be resilient to malicious attacks from the network's nodes. In nature, systems often deal with constant component birth, death, and failure, as well as attacks from malicious components within the system, while allowing well-scaling collaboration between the components. The need for the development of design and implementation tools for these large distributed software systems together with the apparent similarities in requirements between these systems and large-scale natural systems make large distributed software systems an ideal target for future research. Thus, I believe our goal for the next decade should be to concentrate on developing a toolkit of techniques, architectures, and design tools for Internet-sized decentralized distributed software systems for (1) computation, (2) security, and (3) data storage and retrieval, and the tile architectural style presented in this dissertation is a first step in that direction.

References

[1] Martin Abadi, Joan Feigenbaum, and Joe Kilian. On hiding information from an oracle. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC87)*, pages 195–203, New York, NY, USA, May 1987.

[2] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight, Jr., Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss. Amorphous computing. *Communications of the ACM*, 43(5):74–82, May 2000.

[3] Leonard Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, 1994.

[4] Leonard Adleman. Towards a mathematical theory of self-assembly. Technical Report 00-722, Department of Computer Science, University of Southern California, Los Angleles, CA, 2000.

[5] Leonard Adleman, Qi Cheng, Ahish Goel, Ming-Deh Huang, and Hal Wasserman. Linear self-assemblies: Equilibria, entropy, and convergence rates. In *Proceedings of the 6th International Conference on Difference Equations and Applications (ICDEA01)*, Augsburg, Germany, June 2001.

[6] Leonard Adleman, Qi Cheng, Ashish Goel, Ming-Deh Huang, David Kempe, Pablo Moisset de Espanés, and Paul W. K. Rothemund. Combinatorial optimization problems in self-assembly. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC02)*, pages 23–32, Montreal, Quebec, Canada, May 2002.

[7] Leonard Adleman, Ashish Goel, Ming-Deh Huang, and Pablo Moisset de Espanés. Running time and program size for self-assembled squares. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC02)*, pages 740–748, Montreal, Quebec, Canada, May 2002.

[8] Leonard Adleman, Jarkko Kari, Lila Kari, and Dustin Dale Reishus. On the decidability of self-assembly of infinite ribbons. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS02)*, pages 530–537, Ottawa, Ontario, Canada, November 2002.

[9] Gagan Aggarwal, Qi Cheng, Michael H. Goldwasser, Ming-Yang Kao, Pablo Moisset de Espanés, and Robert T. Schweller. Complexities for generalized models of self-assembly. *SIAM Journal on Computing*, 34(6):1493–1515, 2005.

[10] Alchemi .NET grid computing framework. `http://www.alchemi.net/doc/0_6_1`, 2008.

[11] Daniel J. Arbuckle and Aristides A. G. Requicha. Active self-assembly. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA04)*, pages 896–901, New Orleans, LA, USA, April 2004.

[12] Robert Barish, Paul W. K. Rothemund, and Erik Winfree. Two computational primitives for algorithmic self-assembly: Copying and counting. *Nano Letters*, 5(12):2586–2592, 2005.

[13] Yuliy Baryshnikov, Ed G. Coffman, and Petar Momcilovic. DNA-based computation times. *Springer Lecture Notes in Computer Science*, 3384:14–23, 2005.

[14] Yuliy Baryshnikov, Ed G. Coffman, Nadrian Seeman, and Teddy Yimwadsana. Self correcting self assembly: Growth models and the hammersley process. In *Proceedings of the 11th International Meeting on DNA Computing (DNA05)*, London, Ontario, June 2005.

[15] Salman A. Baset and Henning Schulzrinne. An analysis of the skype peer-to-peer Internet telephony protocol. In *Proceedings of the 25th Conference on Computer Communications (IEEE Infocom06)*, Barcelona, Spain, April 2006.

[16] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[17] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transaction on Software Engineering Methodology*, 1(4):355–398, 1992.

[18] Micah Beck, Jack Dongarra, Victor Eijkhout, Mike Langston, Terry Moore, and Jim Plank. Scalable, trustworthy network computing using untrusted intermediaries. In *DOE/NSF Workshop on New Directions in Cyber-Security in Large-Scale Networks: Development Obstacles*, 2003.

[19] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC88)*, pages 1–10, Chicago, IL, USA, May 1988.

[20] Bonnie Berger and Tom Leighton. Protein folding in the hydrophobic-hydrophilic (HP) is NP-complete. In *Proceedings of the 2nd Annual International Conference on Computational Molecular Biology (RECOMB98)*, pages 30–39, New York, NY, USA, March 1998.

[21] Manuel Blum. Coin flipping by telephone. A protocol for solving impossible problems. *SIGACT News*, 15(1):23–27, 1983.

[22] Ravinderjit Braich, Nickolas Chelyapov, Cliff R. Johnson, Paul W. K. Rothemund, and Leonard Adleman. Solution of a 20-variable 3-SAT problem on a DNA computer. *Science*, 296(5567):499–502, 2002.

[23] Ravinderjit Braich, Cliff R. Johnson, Paul W. K. Rothemund, Darryl Hwang, Nickolas Chelyapov, and Leonard Adleman. Solution of a satisfiability problem on a gel-based DNA computer. In *Proceedings of DNA Computing: 6th International Workshop on DNA-Based Computers (DNA00)*, pages 27–38, Leiden, The Netherlands, June 2000.

[24] Yuriy Brun. Arithmetic computation in the tile assembly model: Addition and multiplication. *Theoretical Computer Science*, 378(1):17–31, June 2007.

[25] Yuriy Brun. A discreet, fault-tolerant, and scalable software architectural style for internet-sized networks. In *Proceedings of the Doctoral Symposium at the 29th International Conference on Software Engineering (ICSE07)*, pages 83–84, Minneapolis, MN, USA, May 2007.

[26] Yuriy Brun. Nondeterministic polynomial time factoring in the tile assembly model. *Theoretical Computer Science*, 395(1):3–23, 2008.

[27] Yuriy Brun. Solving NP-complete problems in the tile assembly model. *Theoretical Computer Science*, 395(1):31–46, 2008.

[28] Yuriy Brun. Solving satisfiability in the tile assembly model with a constant-size tileset. Technical Report USC-CSSE-2008-801, Center for Software Engineering, University of Southern California, 2008.

[29] Yuriy Brun and Manoj Gopalkrishnan. Toward in vivo disease diagnosis and treatment using DNA. In *Proceedings of the 2006 International Conference on Bioinformatics & Computational Biology (BIOCOMP06)*, pages 182–186, Las Vegas, NV, USA, June 2006.

[30] Yuriy Brun, Manoj Gopalkrishnan, Dustin Dale Reishus, Bilal Shaw, Nickolas Chelyapov, and Leonard Adleman. Building blocks for DNA self-assembly. In *Proceedings of the 1st Foundations of Nanoscience: Self-Assembled Architectures and Devices (FNANO04)*, pages 2–15, Snowbird, UT, USA, April 2004.

[31] Yuriy Brun and Nenad Medvidovic. An architectural style for solving computationally intensive problems on large networks. In *Proceedings of Software Engineering for Adaptive and Self-Managing Systems (SEAMS07)*, Minneapolis, MN, USA, May 2007.

[32] Yuriy Brun and Nenad Medvidovic. Discreetly distributing computation via self-assembly. Technical Report USC-CSSE-2007-714, Center for Software Engineering, University of Southern California, 2007.

[33] Yuriy Brun and Nenad Medvidovic. Fault and adversary tolerance as an emergent property of distributed systems' software architectures. In *Proceedings of the 2nd International Workshop on Engineering Fault Tolerant Systems (EFTS07)*, pages 38–43, Dubrovnik, Croatia, September 2007.

[34] Yuriy Brun and Dustin Dale Reishus. Path finding in the tile assembly model. Technical Report USC-CSSE-2008-802, Center for Software Engineering, University of Southern California, 2008.

[35] William Joseph Butera. *Programming a Paintable Computer*. PhD thesis, Massachussetts Institute of Technology, Cambridge, MA, USA, February 2002.

[36] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[37] Mark Chaisson, Pavel Pevzner, and Haixu Tang. Fragment assembly with short reads. *Bioinformatics*, 20(13):2067–2074, 2004.

[38] Nickolas Chelyapov, Yuriy Brun, Manoj Gopalkrishnan, Dustin Dale Reishus, Bilal Shaw, and Leonard Adleman. DNA triangles and self-assembled hexagonal tilings. *Journal of American Chemical Society (JACS)*, 126(43):13924–13925, October 2004.

[39] Ho-Lin Chen and Ashish Goel. Error free self-assembly with error prone tiles. In *Proceedings of the 10th International Meeting on DNA Based Computers (DNA04)*, Milan, Italy, June 2004.

[40] Andrew M. Childs. Secure assisted quantum computation. *Quantum Information and Computation*, 5(456), 2005.

[41] Paul Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[42] Bram Cohen. Incentives build robustness in bittorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, June 2003.

[43] Matthew Cook, Paul W. K. Rothemund, and Erik Winfree. Self-assembled circuit patterns. In *Proceedings of the 9th International Meeting on DNA Based Computers (DNA03)*, pages 91–107, Madison, WI, USA, June 2003.

[44] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 2 edition, 2006.

[45] Ronald Cramer, Ivan Damgård, and Stefan Dziembowski. On the complexity of verifiable secret sharing and multiparty computation. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC00)*, pages 325–334, Portland, OR, USA, May 2000.

[46] David Dagon, Guofei Gu, Chris Lee, and Wenke Lee. A taxonomy of botnet structures. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC07)*, pages 325–339, Miami Beach, FL, USA, December 2007.

[47] Premkumar T. Devanbu and Stuart Stubblebine. *Software Engineering for Security: A Roadmap*, pages 225–239. ACM Press, 2000.

[48] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[49] Limor Drori and David Peleg. Faster exact solutions for some NP-hard problems. *Theoretical Computer Science*, 287(2):473–499, 2002.

[50] Joan Feigenbaum. Encrypting problem instances: Or ... can you take advantage of someone without having to trust him? *Lecture Notes in Computer Science on Advances in Cryptology (CRYPTO85)*, 218:477–488, 1985.

[51] Joan Feigenbaum and Lance Fortnow. On the random self-reducibility of complete sets. *SIAM Journal of Computing*, 22(5):994–1005, 1993.

[52] Roy T. Fielding. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, University of California Irvine, Irvine, CA, USA, 2000.

[53] Folding@Home. `http://folding.stanford.edu`, 2007.

[54] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[55] Tsu Ju Fu and Nadrian C. Seeman. DNA double-crossover molecules. *Biochemistry*, 32(13):3211–3220, 1993.

[56] The Globus alliance. `http://www.globus.org`, 2005.

[57] Ashish Goel and Pablo Moisset de Espanés. Toward minimum size self-assembled counters. *Lecture Notes on Computer Science*, 4848/2008:46–53, 2008.

[58] Oded Goldreich. *The Foundations of Cryptography*, volume 2. Cambridge University Press, 2004.

[59] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play ANY mental game. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC87)*, pages 218–229, New York, NY, USA, May 1987.

[60] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(3):690–728, 1991.

[61] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. Simple object access protocol (SOAP) version 1.2. W3C Working Draft, 2002.

[62] Manfred Hauswirth and Mehdi Jazayeri. A component and communication model for push systems. In *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7)*, pages 20–38, Toulouse, France, September 1999.

[63] Thomas Hofmeister, Uwe Schöning, Rainer Schuler, and Osamu Watanabe. A probabilistic 3-SAT algorithm further improved. In *Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science (STACS02)*, pages 192–202, Antibes Juan-les-Pins, France, March 2002.

[64] Ming-Yang Kao and Robert Schweller. Reducing tile complexity for self-assembly through temperature programming. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA06)*, pages 571–580, Miami, FL, USA, January 2006.

[65] Eric Korpela, Dan Werthimer, David Anderson, Jeff Cobb, and Matt Lebofsky. SETI@home — massively distributed computing for SETI. *IEEE MultiMedia*, 3(1):78–83, 1996.

[66] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[67] Oliver Kullmann. Worst-case analysis, 3-SAT decision and lower bounds: Approaches for improved SAT algorithms. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 35:261–313, 1997.

[68] Oliver Kullmann. New methods for 3-SAT decisions and worst-case analysis. *Theoretical Computer Science*, 223:1–72, 1999.

[69] Michail G. Lagoudakis and Thomas H. LaBean. 2D DNA self-assembly for satisfiability. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 54:141–154, 1999.

[70] Stefan M. Larson, Christopher D. Snow, Michael R. Shirts, and Vijay S. Pande. *Folding@Home and Genome@Home: Using Distributed Computing to Tackle Previously Intractable Problems in Computational Biology.* Horizon Press, 2002.

[71] Yehuda Lindell and Benny Pinkas. A proof of Yao's protocol for secure two-party computation. *Cryptology ePrint Archive*, Report 2004/175, 2004.

[72] Anthony MacDonald and David Carrington. Guiding object-oriented design. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS98)*, pages 88–100, Washington, DC, USA, 1998.

[73] Nico Maibaum and Thomas Mundt. JXTA: A technology facilitating mobile peer-to-peer networks. In *Proceedings of the International Workshop on Mobility and Wireless Access (MobiWac02)*, page 7, Fort Worth, TX, USA, October 2002.

[74] Sam Malek, Marija Mikic-Rakic, and Nenad Medvidovic. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Transactions on Software Engineering*, 31(3):256–272, 2005.

[75] Chris A. Mattmann and Nenad Medvidovic. The gridlite DREAM: Bringing the grid to your pocket. In *Proceedings of the 2006 Monterey Workshop on Software Engineering for Embedded Systems*, volume 4322, pages 70–87, Paris, France, October 2007.

[76] Chris A. Mattmann, Nenad Medvidovic, Paul M. Ramirez, and Vladimir Jakobac. Unlocking the grid. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE05)*, St. Louis, MO, USA, May 2005.

[77] Marija Mikic-Rakic, Nikunj R. Mehta, and Nenad Medvidovic. Architectural style requirements for self-healing systems. In *Proceedings of 1st Workshop on Self-Healing Systems*, Charleston, SC, USA, November 2002.

[78] Burkhard Monien and Ewald Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10(3):287–296, 1985.

[79] Robert T. Monroe and David Garlan. Style-based reuse for software architectures. In *Proceedings of the 4th International Conference on Software Reuse (ICSR96)*, page 84, Orlando, FL, USA, April 1996.

[80] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, New York, NY, USA, 1995.

[81] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[82] Mihai Pop, Steven L. Salzberg, and Martin Shumway. Genome sequence assembly: Algorithms and issues. *Computer*, 35(7):47–54, 2002.

[83] John H. Reif, Sadheer Sahu, and Peng Yin. Compact error-resilient computational DNA tiling assemblies. In *Proceedings of the 10th International Meeting on DNA Based Computers (DNA04)*, Milan, Italy, June 2004.

[84] Dustin Dale Reishus, Bilal Shaw, Yuriy Brun, Nickolas Chelyapov, and Leonard Adleman. Self-assembly of DNA double-double crossover complexes into high-density, doubly connected, planar structures. *Journal of American Chemical Society (JACS)*, 127(50):17590–17591, November 2005.

[85] Raphael M. Robinson. Undecidability and nonperiodicity for tilings of the plane. *Inventiones Mathematicae*, 12(3):177–209, 1971.

[86] Rosetta@home. `http://boinc.bakerlab.org/rosetta`, 2007.

[87] Paul W. K. Rothemund. Scaffolded DNA origami: from generalized multicrossovers to polygonal networks. *Nanotechnology: Science and Computation*, pages 3–21, 2006.

[88] Paul W. K. Rothemund, Nick Papadakis, and Erik Winfree. Algorithmic self-assembly of DNA Sierpinski triangles. *PLoS Biology*, 2(12):e424, 2004.

[89] Paul W. K. Rothemund and Erik Winfree. The program-size complexity of self-assembled squares. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC00)*, pages 459–468, Portland, OR, USA, May 2000.

[90] Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia Systems*, 9:170–184, 2003.

[91] Ingo Schiermeyer. Solving 3-satisfiability in less than 1.579^n steps. *Computer Science Logic*, 702/1993:379–394, 1993.

[92] SETI@home. `http://setiathome.berkeley.edu`, 2007.

[93] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[94] Mary Shaw and Paul C. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of the 21st International Computer Software and Applications Conference (COMPSAC97)*, pages 6–13, Washington, DC, USA, August 1997.

[95] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[96] Michael R. Shirts and Vijay S. Pande. Mathematical analysis of coupled parallel simulations. *Physical Review Letters*, 86:4983–4987, May 2001.

[97] Victor Shoup. On formal models for secure key exchange. Technical Report RZ 3120 (#93166), IBM Zurich Research Lab, 1999.

[98] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[99] David Soloveichik and Erik Winfree. Complexity of compact proofreading for self-assembled patterns. *Lecture Notes in Computer Science*, 3892:305–324, 2006.

[100] David Soloveichik and Erik Winfree. Complexity of self-assembled shapes. *SIAM Journal on Computing*, 36(6):1544–1569, 2007.

[101] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 22(6):390–406, 1996.

[102] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice. In print.* John Wiley & Sons, 2008.

[103] Hao Wang. Proving theorems by pattern recognition. *II. Bell System Technical Journal*, 40:1–42, 1961.

[104] Erik Winfree. On the computational power of DNA annealing and ligation. *DNA Based Computers*, pages 199–221, 1996.

[105] Erik Winfree. *Algorithmic Self-Assembly of DNA.* PhD thesis, California Institute of Technology, Pasadena, CA, USA, June 1998.

[106] Erik Winfree. Simulations of computing by self-assembly of DNA. Technical Report CS-TR:1998:22, California Institute of Technology, Pasadena, CA, USA, 1998.

[107] Erik Winfree. Self-healing tile sets. *Nanotechnology: Science and Computation*, pages 55–78, 2006.

[108] Erik Winfree and Renat Bekbolatov. Proofreading tile sets: Error correction for algorithmic self-assembly. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS02)*, volume 2943, pages 126–144, Madison, WI, USA, June 2003.

[109] Gerhard J. Woeginger. Exact algorithms for NP-hard problems: a survey. *Combinatorial Optimization - Eureka, You Shrink!*, pages 185–207, 2003.

[110] Andrew Chi-Chih Yao. Protocols for secure computations. In *Proceedings of the 23rd annual IEEE Symposium on Foundations of Computer Science (FOCS82)*, pages 160–164, Chicago, IL, USA, November 1982.

[111] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science (FOCS86)*, pages 162–167, Toronto, Canada, October 1986.