Automatic Generation of Penrose Empires

by
Jason B Healy

A Thesis Submitted in partial fulfillment of the
requirements for the Degree of Bachelor of Arts
with Honors in Computer Science

Williams College
Williamstown, MA
May 19, 2000

Abstract

Penrose tilings are infinite tilings that cannot tile the plane in a periodic manner. They are of interest to researchers because they may model a new type of matter called *quasicrystals*. Being able to easily generate and manipulate Penrose tilings makes them easier to study, and that is the primary goal of this research. Although Penrose tilings are infinite, we would like to be able to store them in a finite device, such as a computer, so that they may be more easily studied. The primary aim of this research is to attempt to discover some of the underlying structure of Penrose tilings, so that the tilings can be reduced and stored in an efficient, finite manner.

In addition to being able to store tilings, this research also aims to learn more about constructing the tilings themselves. Because we are reducing an infinite amount of data to a finite amount of information, there will necessarily be some computation required to reconstruct the information that is not explicitly stored. This research will investigate methods of constructing Penrose tilings from a small amount of initial information.

The algorithms developed and discussed in this work allow us to represent Penrose tilings as a small set of finite variables and then compute any arbitrary information about the tiling on demand. In this way, we can reconstruct as much of any tiling that we desire, without needing to store the entire tiling. It is hoped that the methods used will allow more detailed study of Penrose tilings to take place in future research.

Acknowledgments

"All life is an experiment. The more experiments you make the better."
—Ralph w. Emerson

"If I have seen farther it is by standing on the shoulders of giants."
—Isaac Newton

This research would not have been possible without the help of several people. My advisor, Duane Bailey, knew that I wanted to do an honors thesis before I did (he seems to have a knack for this). Without his guidance and "motivational" talks, this work would have never been completed. He knew just when to congratulate me on work well done, and (more often) when to tell me all the things that still needed finishing.

I also owe a great thanks to Linden Minnick, who paved the way for my research by performing her own. Out of respect for her, one of my Java objects was given the name "Linden". To this day I cannot believe that she hand-colored so many pictures of Penrose Empires without losing her mind.

Finally, I thank Bill Lenhart, who helped me with revisions to this work. Not only did he help my research, but also my morale. He always greeted me with a smile and friendly conversation, and served as a reminder that I should should be having fun, no matter what I'm doing.

Contents

List of Figures

Chapter 1

Introduction

Before launching into the more specific details that relate to this work, we will provide a brief history of Penrose tilings so that the reader may become familiar with the general background of the problem. In the next chapter, we outline the more specific background of the empires of Penrose tilings, the primary concern of this work.

1.1 Tilings

A *tiling* is generated by a finite set of *prototiles* that are arranged so that copies of the prototiles cover the plane in an edge-to-edge adjacent manner. One of the simplest tilings is a square with copies laid out next to each other to cover a plane as in Figure 1.1. Here, the square is the only prototile.

This tiling exhibits various forms of repetition. For example, if an imaginary ant was to walk along the tiling a distance equal to the width of one tile, the tiling would appear, to the ant, to be the same. In the diagram, this is shown by translating the point of reference from **A** to **B**. The result of this translation suggests a nontrivial isomorphism of the tiling. It is nontrivial because not all translations of a unit distance will preserve the isomorphism of the tiling. This preservation of the tiling after translation is called *translational symmetry*. In a similar fashion, if the tiling were rotated 90 degrees about **A**, an isomorphism of the tiling would again be obtained. This is known as *rotational symmetry*, and **A** is the *center of rotational symmetry*. Finally, if a tiling exhibits translational symmetry, then the tiling is said to be *periodic*. The amount of translation required to preserve the isomorphism of the tiling defines what is called the *unit cell*. The unit cell is a finite portion of the tiling that can construct the entire tiling by being copied and translated. In our square tiling, a sample unit cell could be defined as a group of four squares, shown in Figure 1.1 (there are, of course, other possibilities). It is obvious that this unit cell could be used to tile the entire plane, and thus our square box tiling is periodic.

A tiling is *non-periodic* if there is no unit cell that can tile the plane using

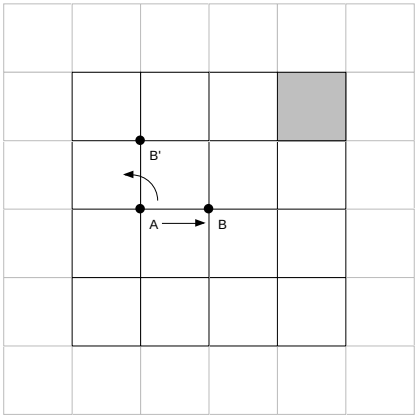


Figure 1.1: An example of a tiling using a square as the only prototile. The shaded tile shows an example of a prototile. The tiling has translational and rotational symmetry.

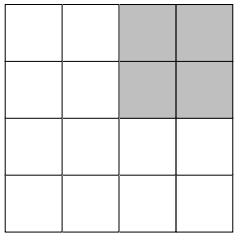


Figure 1.2: One possible unit cell in the box tiling. The shaded region shows how four tiles can be combined into a larger unit cell which tiles the plane.

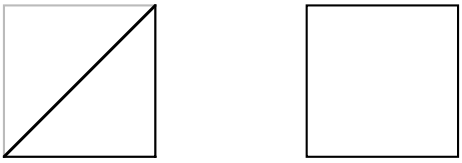


Figure 1.3: Prototiles that could generate a non-periodic tiling. Each triangle tile is the same as half of the square tile.

only translations. An example of a non-periodic tiling would be one where the prototiles were either a square or an isosceles triangle equal to half the square cut along its diagonal, as shown in Figure 1.1. Using these tiles, we could tile the plane as in Figure 1.1, but change one of the squares into two triangles. This would disrupt the periodicity of the tiling, because any unit cell that contained the triangles could not be used to tile portions of the plane without the triangles, and any unit cell that *didn't* have the triangles could couldn't account for the two triangles that exist in the tiling.

However, this non-periodic tiling is non-periodic only because we cleverly tiled the plane so that no nontrivial isomorphism could exist. Is there any set of prototiles such that all tilings *must* be non-periodic? Such sets do exist, and they are called *aperiodic*.

1.2 Aperiodic Tilings

The first set of aperiodic tiles was discovered in 1964 by Robert Berger. His set contained over 20,000 prototiles, which was a slightly unwieldy number to work with. Many people worked on reducing the number of tiles required for aperiodicity, and in 1974 Roger Penrose [Pen74] discovered a set of only two prototiles that tiled the plane aperiodically.

For a time, aperiodic tilings were considered only as an interesting problem in recreational mathematics, due to their strange properties. For example, any finite portion of an aperiodic tiling can be found infinitely many times in infinitely many tilings. Penrose tilings in particular exhibit icosahedral (five-fold) symmetry, which is not found in any kind of periodic tiling.

However, in 1984, a new form of matter was discovered that had icosahedral symmetry. Since icosahedral symmetry cannot be found in periodic tilings of space, this new form of matter could not be described as a crystal, because crystals were modeled purely on periodic structures. On the other had, the new material could not be like glass, which has no structure whatsoever, because the new matter appeared to possess an underlying structure. The new form was called a *quasicrystal* in an attempt to capture both the structured and non-crystalline properties of the matter.

Aperiodic tilings are a potential model for this new form of matter, since they exhibit similar properties. Danzer [Dan89] constructed a three-dimensional

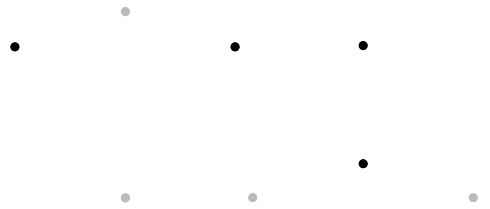


Figure 1.4: Penrose Kite (left) and Dart (right), with vertices colored to enforce matching rules.

analog to the two-dimensional Penrose tiles, thereby furthering the modeling of three-dimensional quasicrystals. Our research seeks to better understand the properties of two-dimensional tilings, in hopes that the concepts can be applied to their three-dimensional counterparts.

1.3 Penrose Tilings

Returning to two dimensions, we will examine Penrose tilings more carefully. Recall that Penrose discovered a set of only two tiles that tile a plane aperiodically [Pen74]. While he discovered more than one set of tiles that does this,[1] we will use the set composed of "Kites" and "Darts", which are names for the prototiles shown in Figure 1.3.[2] At first glance, the tiles do not appear to guarantee aperiodic tilings, as they can clearly be fit together to form a rhombus, which would tile the plane in a periodic fashion. Therefore, there are "matching rules" that dictate the ways that the Kites and Darts fit together. These rules prevent the formation of rhombi. The simplest way to enforce the matching rules is to decorate, or mark, the vertices of the prototiles as shown in Figure 1.3. We augment the earlier side-to-side requirement in tilings to also require the matching of these markings. Following these rules will ensure that a legal Penrose tiling is being constructed.

If these matching rules seem like an unnatural restriction, there is another way to think about them. One can imagine that instead of decorating the vertices, notches were cut out of the sides of the tiles to ensure that the tiles only fit together in certain ways. This too would force aperiodicity, and it would be based solely on the shapes of the tiles. However, since such a "jigsaw" approach makes our diagrams harder to read, we will use decorative matching rules to keep our prototiles as simple as possible.

To prove that his tiles completely tile the plane, Penrose used two processes that Conway calls *inflation* and *deflation* [CL90]. Simply put, inflation is the process of combining prototiles in such a way that they form new tiles that are larger versions of the original prototiles. Deflation is just the opposite—

[1] "Thin" and "thick" rhombi form the other common set of prototiles.
[2] Unless otherwise noted, all figures are drawn to scale.

taking tiles and dividing them into smaller tiles. Put another way, inflation and deflation on a tiling is similar to iterating over a self-similar fractal. As one "zooms in" (or out), similar structures appear on different scales. Penrose uses this property of his tilings to show that any arbitrarily sized two-dimensional surface may be covered by a legal Penrose tiling.[3] Briefly, the proof works as follows: suppose we want to cover an arbitrary-sized disc in two-dimensional space. We simply select a Penrose tile that is large enough to cover the whole disc, forming a Penrose tiling composed of a single tile which covers the disc. Finally, we repeatedly inflate the tile until the sub-tiles are at the scale that we desire. Since we can cover a disc of any size with a Penrose tiling, if we allow the disc to become infinite in size we find that a Penrose tiling can tile the entire plane.

1.4 Legal Tilings

A legal Penrose tiling is one where the tiles have been fit together with no gaps or overlaps, and also one where no matching rules have been violated. A *legal set* of Penrose tiles is set of tiles that form a finite subset of a legal Penrose tiling. In any legal set of Penrose tiles, the lines that compose the tiles intersect at the *vertices* of the tiling. There are only a few configurations of tiles that can form around any vertex in a legal Penrose tiling. More specifically, Grünbaum [GS87] proves that there are only seven, and these seven configurations are shown in Figure 1.4 (the names are those used by Conway).

Because there are only certain legal vertex configurations, we can say that a tile is *forced* if there is a partial vertex configuration in a tiling that only one tile could possibly fill. Because a Penrose tiling covers the entire plane, if we are given any legal set of Penrose tiles it is always possible to add more tiles—one at a time—and continue forever to tile the plane. However, we cannot add tiles in a random or haphazard way; we must choose where and which tiles to add carefully to prevent problems from developing later in the tiling. The aforementioned forced tiles help us complete tilings, as we know that we can place tiles in those spots without affecting the legality of the tiling.

Socolar [Soc91] developed rules that would allow a defect-free tiling to always be generated, using only information available locally at the vertex one wished to add a tile. His method involved placing new tiles adjacent to an already existing patch of tiles (as a base case, one starts with a single tile). If there exists a space along the perimeter of this patch of tiles where a tile would be forced, we place the forced tile there. When there are no more forced tile positions, methods are provided to add tiles to the patch that will ensure consistency with the tiling rules. Once new tiles are added, more forced positions emerge, and the tiling can continue to be grown.

Socolar's methods were primarily motivated by a desire to relate Penrose tilings to quasicrystals. When physical crystals grow, we have no reason to believe that they encode any of the ultimate structure they are creating. Thus,

[3] For a rigorous proof, consult Grünbaum [GS87].

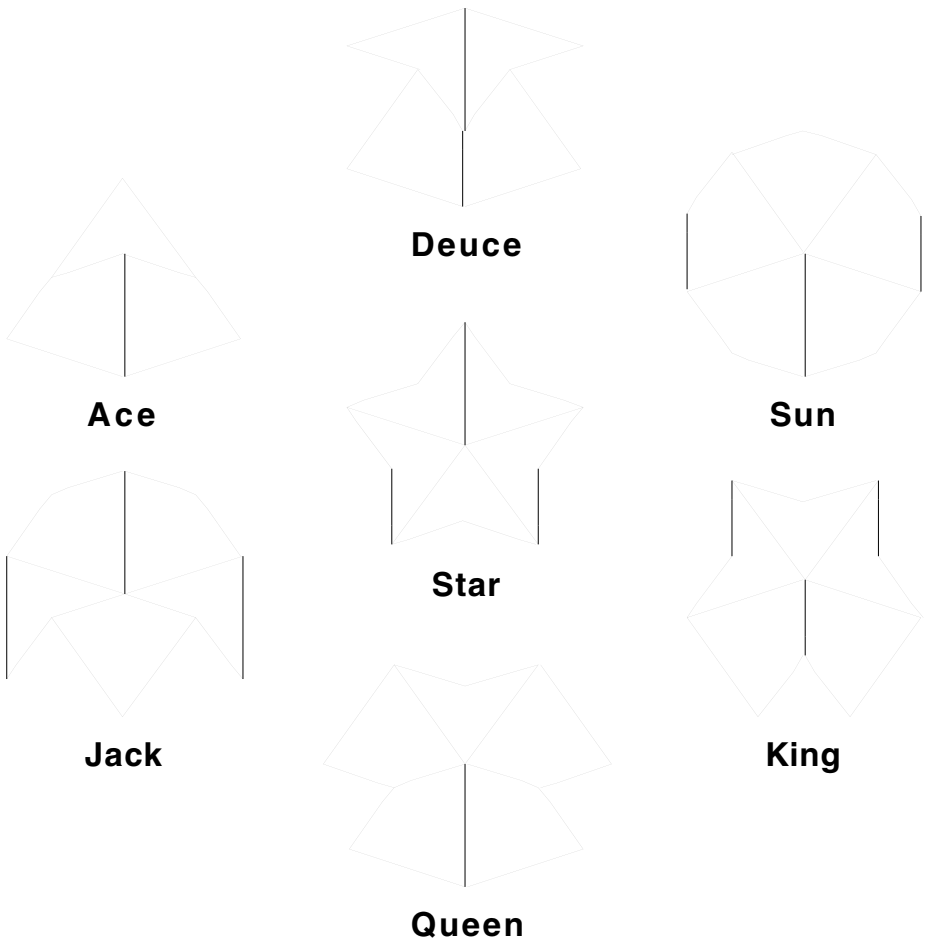


Figure 1.5: The seven legal vertex configurations in a Penrose tiling.

any new growth must be made given only the information that exists at the point where growth is occurring. For this reason, socolar restricted his rules so that they involved only information that could be found locally at a particular vertex. While his rules guarantee a legal tiling, they do not allow us to force a tiling to take any particular shape. If we wanted to grow a tiling with, say, a Kite forced at a specific point in the plane, we could not do so without first generating a tiling that forced the Kite and then recording all the "decisions" that forced that particular tile. Yet, if an analysis of tilings is performed, patterns clearly emerge that suggest an underlying structure to Penrose tilings that are not based solely on local rules.

It is these non-local rules which will allow us to analyze Penrose tilings on a much larger scale, and also to allow us to manipulate and study them with greater ease.

Chapter 2

Background for Research

Penrose tilings are interesting to researchers in several fields for several different reasons. Regardless of the field of study, however, being able to easily manipulate and study Penrose tilings is a universal concern. Although Penrose tilings are infinite in nature, we would like to be able to store them in a finite device, such as a computer. This research attempts to discover some of the underlying structure of Penrose tilings, so that the tilings can be stored and manipulated as efficiently as possible.

In addition to being able to store tilings, this research also aims to learn more about constructing the tilings themselves. Because we are reducing an infinite number of Penrose tiles to a finite amount of information, there will necessarily be some computation required to generate the information that is not explicitly stored, as is usually the case in such situations. This research will investigate methods of constructing Penrose tilings for study and research.

This work seeks to expand directly on the work of Linden Minnick [Min98] by searching for a general algorithm to find the tiles forced by arbitrary patches of tiles, and also a method to regenerate a complete tiling when given only the minimal information needed to represent the tiling. Algorithms will be developed and implemented in software to aid in this task.

2.1 Ammann Bars: A Non-Local Aid

Socolar noted that his rules were in agreement with what are called *Ammann bars* (or *Ammann lines*). Ammann bars are markings that can be superimposed on tiles and that verify the matching rules. A set of Penrose tiles with Ammann markings are shown in Figure 2.1. When tiles are placed adjacently, the adjacent Ammann bars must be collinear. The final result, in an infinite Penrose tiling, is that the Ammann bars form infinitely long parallel lines. Because of the different orientations of the tiles, there are actually five "sets" of Ammann bars, where a "set" is a group of lines that are all parallel to one another. The five sets in a Penrose tiling are rotationally separated by angles that are multiples

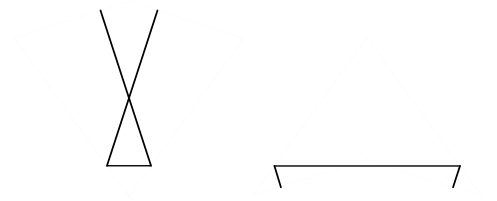


Figure 2.1: Penrose Kite (left) and Dart (right) with Ammann markings.

of 72°. This has the added benefit of reinforcing the idea of isohedral symmetry in the tiling.

Ammann bars can also be thought of as a way of placing Penrose tiles. We can imagine drawing the Ammann bars onto the plane *first*, and then putting down Penrose tiles that agree both with the other tiles and with the Ammann bars. If one has five complete sets of Ammann bars in a plane, then the intersection and spacing of those lines will force the Penrose tiles into specific locations, because the tiles must match the lines in the ways shown in Figure 2.1. In this way, the Ammann bars become the important part of the tiling, and the Penrose tiles become the decorations. If the Ammann bars can be generated so that they force legal tilings, then the Penrose tiles impart no new information.

In fact, the Ammann bars can be generated so that they force legal Penrose tilings. Each set of parallel Ammann bars in a legal Penrose tilings forms what Conway calls a *musical sequence*. A musical sequence is a set of parallel lines, where each line must be separated from its neighbor by one of two specific intervals. More precisely, the distance between lines can be either a *short* distance, or a *long* distance. The short and long distances themselves are related in the following way:

Let $|\mathbf{L}|$ and $|\mathbf{S}|$ stand for the length of a "long" ($\mathbf{L}$) and "short" ($\mathbf{S}$) interval, respectively. Let $\#\mathbf{L}$ and $\#\mathbf{S}$ stand for the number of "long" and "short" intervals, respectively, in a Musical sequence. Grünbaum [GS87] has shown that if n is the total number of intervals in the musical sequence, then

$$\lim_{n\to\infty} \frac{\#\mathbf{S}}{\#\mathbf{L}} \quad \approx \quad \frac{|\mathbf{S}|}{|\mathbf{L}|} \quad = \quad \frac{1}{\tau}$$

τ is the *Golden Mean*, which is an irrational number[1] whose precise mathematical definition is $\frac{1+\sqrt{5}}{2}$. The number has been known since at least the time of the ancient Greeks, and it is often found in nature [Dis59].

This "golden" relationship hints at a pattern in the sequence of intervals. In fact, the sequence of intervals between lines is also aperiodic, though in one dimension instead of two. And just like Penrose tilings, there are "legal" ways to arrange the intervals between bars. A sequence can be checked for legality

[1] 1.6180339887498948482045868343656381177203091798057628621354486227052604 62...

through several different means, many of which are outlined by Minnick (see below).

Assuming one knows how to generate legal sequences, one would find that the pattern of **L**'s and **S**'s does not repeat periodically. In generating musical sequences, certain choices of intervals cause certain bars to be forced at a specific distance away from the other bars in order to preserve the legality of the sequence. Thus, we also have a concept of forced and unforced bars, just as we do forced and unforced tiles in a Penrose tiling.

By generating Ammann bar sets and intersecting them in the plane, it should be possible to generate Penrose tilings. Given this approach, solving the problems of musical sequences must be completed before attempting to solve the problems of Penrose tilings.

2.2 Minnick's Thesis

Minnick [Min98] combined the techniques described above so that knowledge about the empire of tiles could be expanded. The *empire* of a tile or set of tiles is the group of tiles that are forced by the presence of the initial set. Socolar's rules were insufficient to describe the empire of tiles because they only considered local conditions. His rules would show all *locally* forced tiles at any given vertex, but did not attempt to show *nonlocal* tiles which might also be forced. Grünbaum and Shephard performed a brief analysis of the relationship between Ammann bars and forced tiles in an attempt to find tiles that were forced non-locally. They noted that certain shapes formed by the intersections of Ammann bars always forced certain tile configurations. They also published figures showing some of the non-local tiles that were forced. While the known empire had been expanded, it still was not complete.

Minnick expanded on these empire-finding efforts. First, she described several ways to generate legal musical sequences, and provided proofs for the methods. The method we rely on in this paper relies on her geometric method to generate and verify musical sequences. The geometric method always allows for the storage of a musical sequence (which is infinite) in a finite amount of space. Additionally, it is easy to force arbitrary bars in the sequence, as well as check arbitrary bars to see if they are forced.

Minnick also performed a close analysis of the Penrose tilings' relationship to the Ammann bars. She constructed a set of distances that related points on the prototiles to the Ammann bars, and used these distances to compute how certain sets of tiles would force sets of Ammann bars. She constructed diagrams that showed how the Ammann bars would be forced around all seven of the legal Penrose vertex configurations, and also found partial empires of these vertex configurations. Finally, she demonstrated that certain configurations of tiles directly forced Ammann bars that did not touch the tiles. This result was important because it demonstrated that there was a relationship between Ammann bars and Penrose tiles that worked over a distance, rather than the tiles only forcing bars that passed through the tiles themselves. Her results also

suggest that there exist more rules for Penrose tilings that may depend on even larger groups of tiles.

Minnick constructed a high-level algorithm for finding part of the empire of an arbitrary patch‘of tiles. She implemented the algorithm by hand for certain vertex configurations to demonstrate that her methods would find forced tiles that had not yet been discovered. She did not come up‘with a general implementation of her algorithm, however, and so could not generate arbitrary empires. Additionally, while her algorithm should generate the empire of a set of tiles, it assumes that we know all the forcing rules that might exist for Penrose tilings. We do not know if we possess all the forcing rules, so Minnick's algorithm does not necessarily produce complete empires yet.

2.3 Proposed work

The efforts described in this work seek to build directly on Minnick's work. We will attempt to implement her general algorithm in a systematic way, so that we can find the empire of any arbitrary patch‘of tiles. While Minnick has provided an excellent starting point, her algorithms are described at a very high-level. Much work must be done to actually implement them in a programming language.

Specifically, the basic functions that need to be written in software are these:

1. Representation of a musical sequence, with the ability to find and force arbitrary bars,
2. Interaction of five musical sequences in a plane, as they would appear in a Penrose tiling,
3. The ability to find intersections of forced musical sequence bars,
4. The ability to find patterns in a field of intersection points,
5. The ability to analyze the patterns and fill in missing or incomplete information, and
6. The ability to draw the forced Penrose tiles on top of the musical sequences.

With the implementation of the algorithms, we should be able to generate partial empires of arbitrary sets of tiles (partial because we do not know all the forcing rules yet). Using this as a tool, we can generate empires that would be too difficult to construct by hand, and study these empires for new patterns and rules. We hope to better understand Penrose tilings, and find even better ways to construct and store them using computers.

Chapter 3

Musical Sequences

Musical sequences are one-dimensional aperiodic structures. In many ways they are the one-dimensional analog to Penrose tilings because they contain similar properties of similarity, inflation, deflation, and, of course, aperiodicity. Musical sequences can be represented in a number of different ways, depending on the context. to prevent confusion, we will use a consistent notation for all musical sequences, and introduce the meaning of the notation at the appropriate times.

3.1 Theory

One possible representation of a musical sequence is a string over the two-symbol alphabet {**L**, **S**}. The symbols in a musical sequence cannot be arranged in an arbitrary fashion; there are certain orderings of these symbols that are not allowed.

3.1.1 Inflation

Using this symbolic representation of a musical sequence, the simplest musical sequence is the empty string ϵ. A more useful trivial string is the single symbol **S**, which is a valid musical sequence by definition. Starting with this single **S**, a new musical sequence can be built by using a process called *inflation*. Inflation means rewriting all the symbols in a musical sequence using a parallel rewriting scheme called an *L-system*. The rewrite creates a longer string with all of the symbols having been rewritten simultaneously. The rewriting rules for inflation are:

$$L \rightarrow LS$$

$$S \rightarrow L$$

Note that the rules must be applied *in parallel* to all of the symbols in the string. A single **L** cannot become **LL** during one rewrite, but must rather go from **L** to **LS** in one step, and then from **LS** to **LSL** in the next step. Another way to think of the rewriting rules is by imagining that the symbols represent

S
L
LS
LSL
LSLLS
LSLLSLSL

Figure 3.1: Inflation of a musical sequences, beginning with **S**.

rabbits: **S** is a baby rabbit, and **L** is an adult rabbit. In every unit of time, all of the adult rabbits each create a single new baby rabbit ($L \to LS$), while at the same time any baby rabbits mature into adults ($S \to L$). All the rabbits change simultaneously, or "in parallel."

It is always possible to inflate a musical sequence to create a longer musical sequence because the validity of a musical sequence is not changed by inflation [GS87]. Therefore, if we start with a valid musical sequence, we can repeatedly apply the inflation rules to obtain a musical sequence of arbitrary length. Because the single symbol **S** is a trivially valid musical sequence, we may begin with it and inflate repeatedly to obtain other valid musical sequences. It is important to note that not all finite portions of musical sequences can be generated in this way unless the sequence generated has infinite length. Only an infinite musical sequence will contain all finite substrings, much in the same way that only an infinite Penrose tiling will contain all finite patches of Penrose tiles.

Figure 3.1.1 shows a sequence that is inflated from the single symbol **S**. Note that after the first rewriting, this sequence is exactly the same as one beginning with the single symbol **L**.

3.1.2 Musical Sequence Rules

After a moment of careful study we see that there are never any consecutive **S** symbols in a musical sequence, nor are there ever more than two consecutive **L** symbols. These are rules of any musical sequence, which will be referred to as *consecutive symbol* rules. They may be stated as follows:

Theorem 1. *No valid musical sequence may have more than one consecutive **S** symbol.*

Theorem 2. *No valid musical sequence may have more than two consecutive **L** symbols.*

Proof. to show that there can never be consecutive **S** symbols, we can think of the **L** symbols as "generators" for the sequence. Only **L** symbols create new symbols, and they do so in a very specific manner. **L** symbols create a single new **S** symbol to the right of the **L** symbol during every rewrite of the string. Additionally, all extant **S** symbols turn into **L** symbols during every rewrite.

These two rules work together to prevent two **S** symbols from ever being next to each other. Because **S** symbols are always created to the right of a **L** symbol, there is no way for two **L** symbols to produce adjacent **S** symbols during the *same* rewrite; one of the **L** symbols must necessarily interrupt the **S** symbols. Because all **S** symbols become **L** symbols during every rewrite, there is no way for two **S** symbols created during *different* rewrites to be next to each other; one would have been changed to a **L** before (or during) the other's creation.

Once we have shown that there cannot be consecutive **S** symbols, showing that there cannot be more than two consecutive **L** symbols becomes quite easy. As demonstrated above, **L** symbols always generate a new **S** symbol, so **L** symbols will always insert a **S** between themselves after a rewrite. The only reason why there exist consecutive **L** symbols at all is because the **S** symbols become **L** during on rewrite, but do not produce a new **S**. During this single step, a **L** can exist without a **S** next to it. However, in the next step, this new **L** will begin producing **S** symbols, thus preventing further accumulation of consecutive **L** symbols. Since there cannot be consecutive **S** symbols (shown above), we can have at most two consecutive **L** symbols: one which just changed from a **S** to a **L** during this rewrite, and one to it's right which has been a **L** prior to this rewrite. Any other **L** would produce a **S** (thus interrupting the consecutive **L** symbols), and we know that another **S** symbol is impossible. Thus, there cannot be more than two consecutive **L** symbols in a valid musical sequence. □

The inflation rules, along with the consecutive symbol rules, can be used to show the relative frequency of the two symbols. We know that for every **S** there must be at least one **L**, so we would expect the ratio of the number of **S** symbols (—nS) to the number of **L** symbols (#**L**) to be less than one. Additionally, we know that there cannot be more than two consecutive **L** symbols, so we would expect the same ratio to be greater than $\frac{1}{2}$. In fact, Grünbaum and Shephard have shown that for any infinite musical sequence

$$\frac{\#\mathbf{S}}{\#\mathbf{L}} \quad = \quad \frac{1}{\tau} \quad = \quad (1-\tau)$$

These relationships hold true in the limit: when the sequence has infinite length we say the sequence is *prototile balanced*[1] and these relationships hold. In finite portions of musical sequences, these relationships are approximated, but not perfect.

These rules, when combined with our notion of inflation, give us a tool that can confirm the validity of musical sequences. By looking at a string and verifying that it does not violate the rules of consecutive symbols, we know that the string is possibly musical. Another step is required, however, to verify that the string is truly musical.

[1] prototile balanced implies that the limit of the ratio is finite and nonzero

LSLLSLSL
LSLL
LS
L
ϵ

Figure 3.2: Deflation of the musical sequences from Figure 3.1.1.

3.1.3 Deflation

Because all musical sequences can be constructed from a single symbol, we should be able to trace the path that created any valid musical sequence. If we can, in a sense, "run inflation backwards," then we can verify a musical sequence. To do so, we use rules which define *deflation*, the opposite of inflation:

$$LL \to S$$

$$S \to L$$

$$L \to \epsilon$$

Deflation is exactly like inflation in that the rewriting rules must be applied in parallel to the entire string, and only one rule may be applied to a group of symbols in a single rewrite. In other words, we cannot rewrite **LL** to **S** and then rewrite **S** to **L** all in a single rewrite.

Deflation always preserves the validity of a musical sequence [GS87], and this allows us to verify musical sequences. Given an arbitrary string, if we can repeatedly deflate it until we are left with the empty string ϵ, then the original string must be a valid musical sequence. At no time during the deflations can the string violate our consecutive symbol rules; if it does, then the string is not a musical sequence. If we did not make sure that the string always obeyed our consecutive symbol rules, then we could accidentally deflate an invalid string into a valid one. For example, **LLL** could be deflated to **S**, thus transforming an invalid string to a valid musical sequence. We must therefore check the validity of the musical sequence after each step of the deflation.

To illustrate this point, we will verify the musical sequence constructed in Figure 3.1.1. Because the string was built using only inflations from the trivial (valid) case, the string must be a valid musical sequence. Figure 3.1.3 clearly shows that the string does deflate to the empty string ϵ, and the reader may readily verify that none of the intermediate strings violate our consecutive symbol rules.

As for invalid strings, consider the two invalid strings in Figure 3.1.3. The one on the left is clearly invalid because it violates our rule of having no more than two consecutive **L** symbols. The one on the right, however, is more troublesome. The string deflates several times before our consecutive symbol rule is violated. This demonstrates that verifying a string requires deflating it until all that remains is the empty string; if we stop before that point then there may

LSLLSLLL
(More than two consecutive
L symbols)

LLSLLSLL
SLSLS
LLL
(More than two consecutive
L symbols)

Figure 3.3: Examples of invalid musical sequences. The left example is trivially invalid, but the right example requires several deflations before the flaw becomes apparent.

be an inconsistency in the sequence that will not become apparent until further deflation.

3.1.4 Forcings

The consecutive symbol rules place restrictions on which patterns of symbols may exist in a musical sequence. Certain sequences cannot exist because they cannot be deflated. The sequences that can be deflated completely appear to contain patterns of symbols, with the patterns being the result of certain symbol orderings being disallowed. For example, in a musical sequence we might see a lot of **LL** pairings, but no **SS** pairings, because the sequence rules allow for the former and disallow the latter. These patterns give the appearance of *forcings* in the musical sequence, because certain symbols *always* follow others. Consider the sequence below:

$$\ldots \mathbf{LSLL} \ldots$$

If we were to add a new symbol to the right-hand side of this musical sequence, the only symbol we can use is a **S**, because adding a **L** would violate the rule preventing more than two consecutive **L** symbols. Because only one symbol can possibly be used at this point in the musical sequence, we say that this symbol is *forced*. Essentially, the existing symbols combined with the sequence rules demand the appearance of an **S**.

Forcings are much more than trivial rule-following. As Figure 3.1.3 showed, flaws in a musical sequence may not become apparent until several deflations have been performed. Because of this, placing one symbol may affect the placement of another distant symbol on the sequence, even though they do not appear to have any relationship to each other. As deflation s are performed, symbols that are far apart are brought closer together until they are eventually near enough to affect each other through the consecutive symbol rules. Because of this, placing a symbol in an infinite sequence forces not only adjacent symbols, but also an infinite number of non-local symbols.

Figure 3.4: The musical sequence **LSLLSLSL** represented using Ammann bars. The intervals between bars are marked with their symbolic equivalents to show the relationship between Ammann bars and pure symbolic musical sequences.

3.2 Ammann Bars

Now that we have defined the basic properties of musical sequences, we will remove one layer of abstraction and discuss *Ammann bars*. Ammann bars are musical sequences expressed as sets of parallel lines rather than as symbolic strings. Instead of using **L** and **S** to represent the two possible states in a musical sequence, we use the space between adjacent parallel lines to encode this information. In a sequence of Ammann bars, the space between any two adjacent lines can only be one of two lengths. One length is called *long* and the other is called *short*. As one would expect, *long* intervals are analogous to the **L** symbol the string representation, and *short* intervals represent **S** symbols.

As an example, —figfig:base-ammann shows the musical sequence from Figure 3.1.1 as a sequence of Ammann bars. It now becomes convenient to speak of *intervals* between bars, rather than symbols. For consistency, however, we will refer to these intervals using the familiar symbols **L** and **S**. Note that sequences of Ammann bars *are* musical sequences; the bars simply represent the sequences in a graphical fashion. Throughout this work, we will refer to "a sequence of Ammann bars," to denote that it is different from our symbolic representation used earlier. The reader should simply be aware that a valid arrangement of Ammann bars always implies a musical structure.

Ammann bars are particularly useful when we visualize the forcing concept discussed earlier. When a forcing occurs in a sequence of Ammann bars, two things are forced. First, a particular Ammann bar is forced to be in a certain position relative to all the other forced Ammann bars. Secondly, the interval between all these forced bars is forced to be a certain length: either **L** or **S**. While these are really two different ways of describing the same phenomenon, it is sometimes more convenient to speak of forcing *bars* or forcing *intervals*, and we will freely switch between the two different terms. We will say that an *interval* is forced when we are speaking about the relative spacing of bars, but the absolute positions of the particular bars are not important. We will say that a *bar* is forced when we are forcing a specific bar into a position, but the interval it forms is not important.

Figure 3.2 illustrates these two ways of forcing. We can force bars in any of the five configurations shown at the top of the figure if we desire specific bars

to be in specific places. The last portion of the figure shows that we can also just force the two bars on the left and right sides of the figure, thus forcing the interval between them to be a certain distance. In this case, we get an interval of a specific length, and the actual ordering of bars in between does not matter.

3.3 Five-Fold Interactions

Ammann bars are two-dimensional; they are sequences of parallel line segments with space in between them. Additionally, the *long* and *short* intervals between the bars have a relationship between them. In a sequence of Ammann bars, a *long* interval is defined as being τ times as long as a *short* interval. Once we have picked an arbitrary length for either a long or short interval, all other distances in a sequence of Ammann bars are known. If we use $|\mathbf{L}|$ to denote the length of a long interval, $|\mathbf{S}|$ to denote the length of a short interval and n to denote the total number of intervals in the sequence, then we can sum up‘what we know about Ammann bars as follows:

$$\lim_{n\to\infty} \frac{\#\mathbf{S}}{\#\mathbf{L}} \quad \approx \quad \frac{|\mathbf{S}|}{|\mathbf{L}|} \quad = \quad \frac{1}{\tau} \quad = \quad (1-\tau)$$

Ammann bars are more than just graphical representations of musical sequences. They are the basis of an underlying pattern for Penrose tilings. In any Penrose tiling, five sequences of Ammann bars interact in the plane, dictating where the tiles should be placed. Figure 3.3 shows how the tiles and bars interact. The five Ammann bar sequences are arranged at 72° angles. The intersections of the bars in the sequences define where the tiles in the Penrose tiling must go.

The direct relationship of Ammann bar position to Penrose tile position also links forced Ammann bars to forced Penrose tiles. If we force certain bars in the five sequences of Ammann bars that comprise a Penrose tiling, then we have forced the positions of actual tiles in the tiling as well. This property is what our research is based on: by exploiting the underlying structure that the Ammann bars provide, we can determine the positions of forced tiles, and thus the empires of Penrose tiles.

3.4 Extended Theory

In the next chapter we will discuss in depth the relationship of Ammann bar formations to Penrose tile placement. For the moment, assume that some method of placing Penrose tiles on top of Ammann bars exists. If this is the case, then finding the empires of tiles can be outlined as follows:

1. Find the Ammann bars forced by our initial set of tiles.
2. Find the Ammann bars that are in turn forced by these initial Ammann bars.

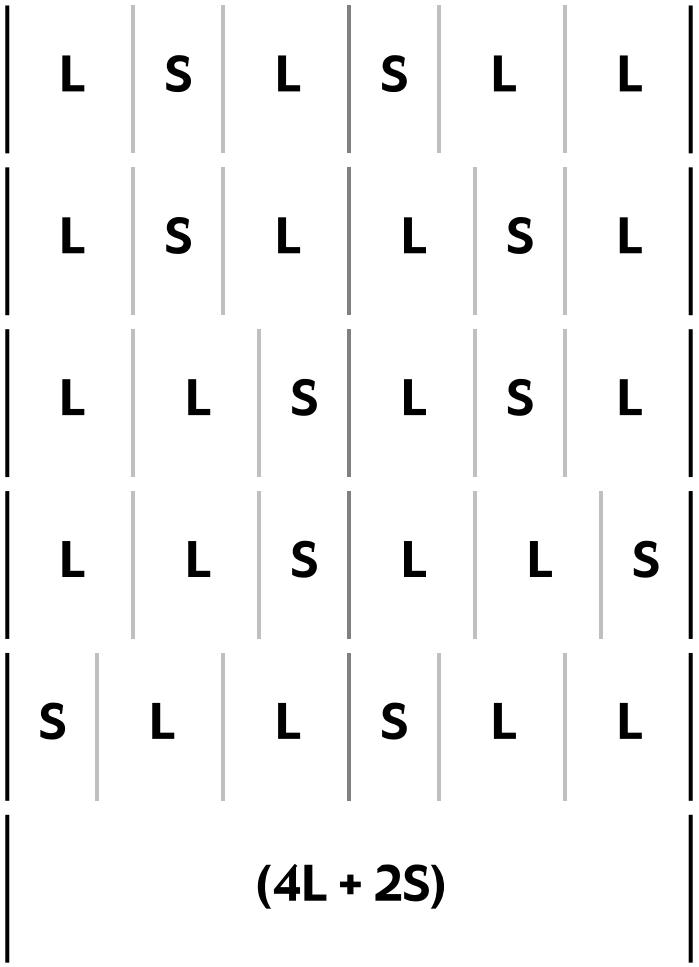


Figure 3.5: The distance between two Ammann bars, represented in terms of **L** and **S**. Above, all the possible orderings of symbols that fit between two bars forced at a distance (4**L**+2**S**) from each other. Notice that the middle bar (dark grey) always appears in the same place, even though we did not explicitly force its position. This bar is forced.

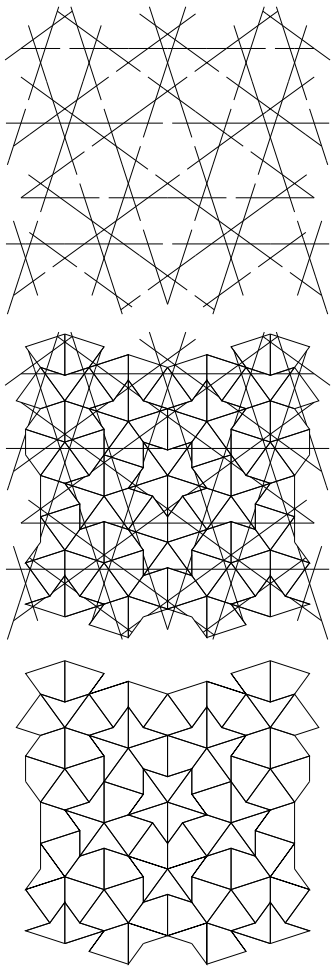


Figure 3.6: An example of how Ammann bars and Penrose tilings interact. On the top, five sequences of Ammann bars. On the bottom, a Penrose tiling. The center shows the two superimposed.

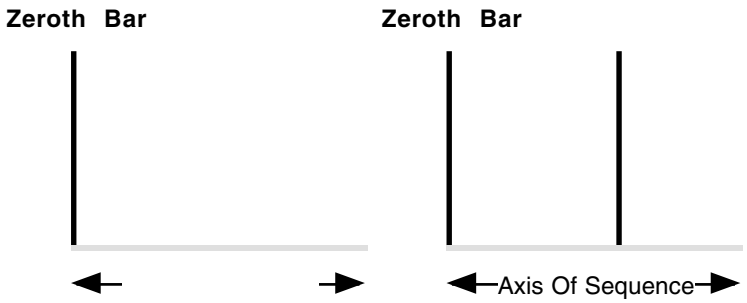


Figure 3.7: An unforced musical sequence represented as Ammann bars, and a musical sequence with a single interval forced.

3. Place the deducible tiles on top of these newly forced Ammann bars.

At a very high level, that is what our research deals with. In the next chapter we will discuss the first and third s outlined above, which relate the Ammann bars to the tiles. Right now, what interests us is finding all the Ammann bars that are forced by other Ammann bars.

3.4.1 Forcing Arbitrary Ammann bars

A good way to imagine a musical sequence represented by Ammann bars is by thinking of the sequence as initially unforced, as shown in Figure 3.4.1. In this state, there is a single Ammann bar, called the *zeroth bar*. This bar is used as a point of reference only; it does not force anything by itself because forcing requires two bars (two bars enclose an interval, and intervals are what represent the musical sequence). This zeroth bar can be thought of as the center of the sequence, with the remainder of the sequence continuing infinitely in either direction from the center.

Suppose that we now force a new bar in the sequence. By adding a bar, we have forced an interval somewhere in the set to be a certain length. As demonstrated earlier, a forcing in a musical sequence causes an infinite number of other forcings to occur. But how do we know which intervals are forced? Forced intervals are not trivial, because long-range defects in the musical sequence may not become apparent until several deflations have been performed. so far, we have only described a method of determining whether a sequence is valid or not. We have not described a way to force arbitrary intervals, because the methods we have discussed so far (inflation, deflation, and consecutive symbols) are all *local*, and do not adequately deal with long-range effects.

3.4.2 Minnick's Work

Minnick [Min98] researched arbitrary forcings in musical sequences, and refined one algorithm in particular. The algorithm uses a geometric representation

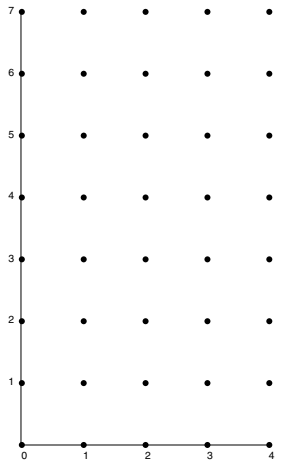


Figure 3.8: An empty integer lattice.

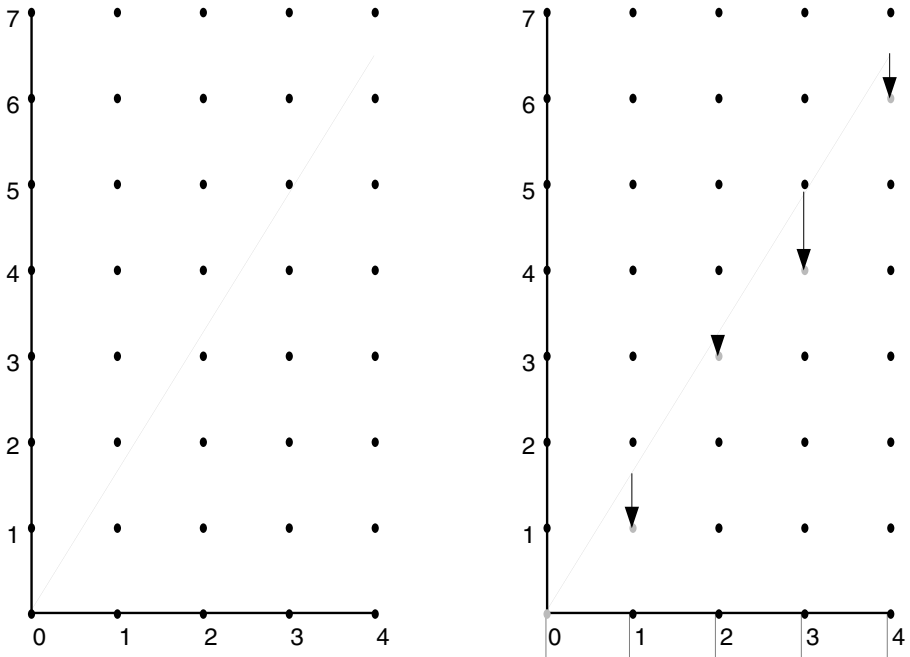
of the musical sequence, and allows for the location and creation of arbitrary forcings in a constant amount of time.

To visualize the algorithm, we begin with an *integer lattice*. This lattice is comprised of a Cartesian coordinate system, with a point placed at every integer pair. A lattice is shown in Figure 3.4.2. The x-axis integers represent bar numbers, with the zeroth bar at the x value 0. The space between x-axis integers represents the intervals between bars. Thus, the bar immediately to the right of the zeroth bar is at $(x = 1)$, and the first interval of the sequence is bounded by the bars at $(x = 0)$ and $(x = 1)$.

Now we draw a line ℓ through the integer lattice, with slope τ and y-intercept between 0 and 1, as shown in Figure 3.4.2. For every x value on the lattice, take the y value of the line ℓ, and truncate it to an integer value. Because the slope of ℓ is τ, which is between 1 and 2, the difference between truncated y values will be either 1 or 2. A difference of 1 represents a *short* interval, and a difference of 2 represents a *long* interval. Figure 3.4.2 shows this translation from a geometric line to a symbolic musical sequence.

Therefore, a single line of slope τ defines an entire musical sequence in this geometric system. If we want to find a particular interval, we need only to determine the difference between two y values on the integer lattice. For example, if we wanted to find the interval distance between the 3rd and 4th bars in —figfig:lattice-conversion, we would compute the truncated y values at bar 3 (4) and at bar 4 (6) and take the difference (2). The difference tells us that the interval is *long*. Note that this calculation depends only on the y-intercept of the line; no other information about surrounding intervals is required.

The example described above shows a completely forced musical sequence. All of the values are determined by the line ℓ, and all of these values are forced. In order to represent an unforced sequence, we must introduce some kind of



y-values:	(1-0)	(3-1)	(4-3)	(6-4)
y-values:	1	2	1	2
Interval Length:	**S**	**L**	**S**	**L**

Figure 3.9: Left: An integer lattice with a line of slope τ through it. Right: The line with the y values truncated to integers. Bottom: the integer values converted to interval symbols

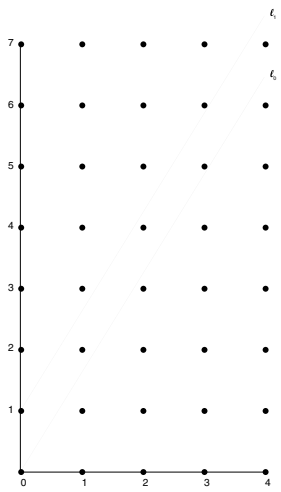


Figure 3.10: A lattice with two lines with slope τ ℓ_0 and ℓ_1, with y-intercepts 0 and 1.

uncertainty.

Minnick accomplishes this by using two lines instead of one. Returning to our integer lattice, suppose we have two lines ℓ_0 and ℓ_1, both with slope τ. Let us also suppose that the y-intercept of these two lines initially differ by exactly 1: ℓ_0 has a y-intercept of 0 and ℓ_1 has a y-intercept 1. This is shown in Figure 3.4.2. These two lines effectively bound the set of all possible musical sequences.

Under this representation, in order for a bar to be forced at any particular x value, the truncated y value at that point *must be the same for both lines* ℓ_0 *and* ℓ_1. In this initial configuration, no bars are forced because any y value for ℓ_0 will always be one less than the y value for ℓ_1.

To force a bar under this system, the y-intercept of one of these lines must be changed. Changing the y-intercept causes the entire line to move, and moving the line allows us to change the y value at any point. Only two changes are possible: ℓ_0 can have its y-intercept *raised*, or ℓ_1 can have its y-intercept *lowered*. Because of this, the y-intercepts of the lines are always moving closer together, and in the limit they will become a single line with one y-intercept. When this happens, we have a single line just like the one in Figure 3.4.2, and this single line defines a completely forced musical sequence. But so long as the two lines are different in some way, there are some bars in the musical sequence that are not forced.

When we force a bar, the intercept of one of the lines is moved just enough to force the y value at that bar to agree with that of the other line. Consider Figure 3.4.2: after truncation, ℓ_0 has a y value of 4 and ℓ_1 has a y value of 5. In order to force this bar, we must either raise ℓ_0 until its truncated y value

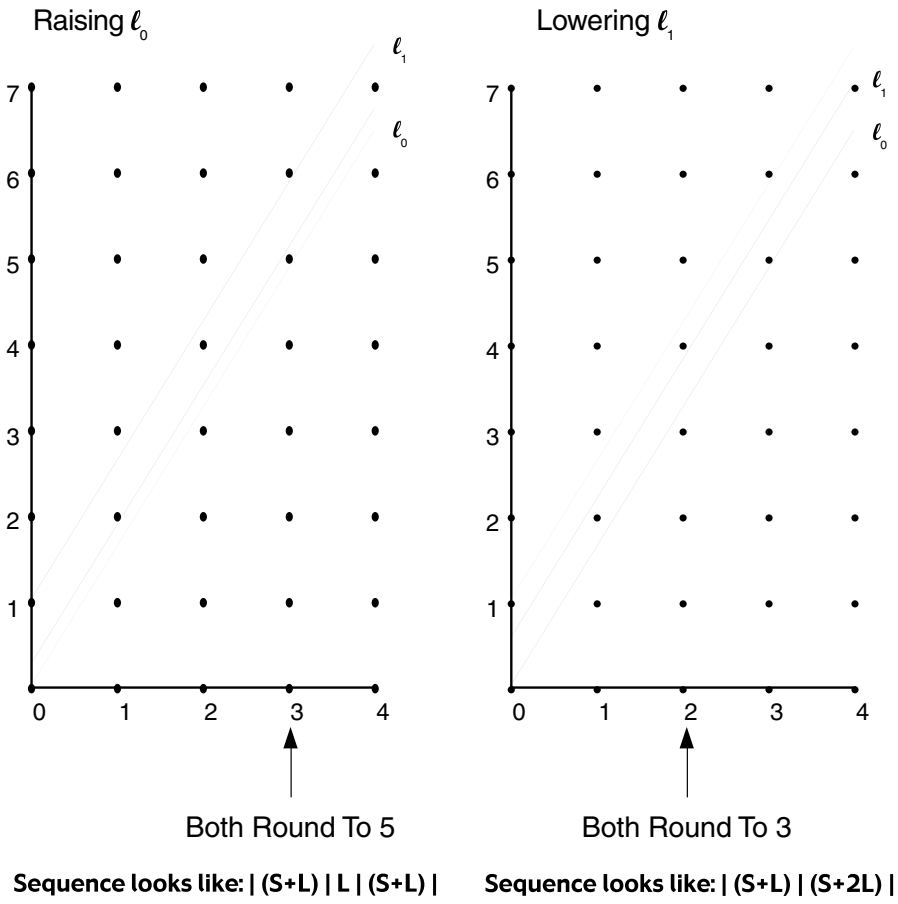


Figure 3.11: Forcing a bar in the integer lattice.

becomes 5, or we must lower ℓ_1 until its truncated y value becomes 4.

Raising ℓ_0 has the effect of making the interval between the bar we are forcing and the bar preceding it *longer*. Long intervals are created when two truncated y values differ by two instead of one. In order to differ by two, the latter of the two y values must be as high as possible, so that when it is truncated it is two more than the truncated y value of the bar before it. Thus, to make the y value as high as possible, we should *raise* ℓ_0.

Using similar reasoning, lowering ℓ_1 has the effect of making the interval between the current bar and the bar preceding it *shorter*. Short intervals are created when two truncated y values differ by one, and this is achieved by making the y value of the current bar as low as possible. Thus, we should *lower* ℓ_1 to make a certain interval shorter.

Using this method, we can force an arbitrary bar to be a relative distance

away from other bars. Note that we are not guaranteed to have an interval of the desired length merely by moving ℓ_0 or ℓ_1. Moving a particular line only causes the sequence to expand with a greater number of short or long intervals; the legality of the sequence cannot be violated by moving one line or another.

Minnick's system gives us the power to deal with forcings in a simple manner, as well as giving us the ability to represent musical sequences in a compact form. The geometric representation uses only two lines to represent an entire musical sequence, yet these lines are sufficient to tell us where any, and all, forcings occur, as well as allowing us to define new forcings of our choosing. Because Penrose tilings can be represented as musical sequences in the form of Ammann bars, Minnick's algorithms form the base of our work to find the empires of Penrose tiles.

3.5 Pseudo-Code Implementation

Minnick's geometric method is fairly straightforward, though actually implementing it requires some work. The two major problems in the implementation phase are efficiency and precision. We do not want to lose the efficiency of the algorithm; as described above the forcing operations appear to take only constant time. Since our work will rely heavily on these operations, we need to make them as efficient as possible. Precision is important because the computer is a finite device, whereas the conceptual algorithms assume infinite precision. In the algorithm, the lines ℓ_0 and ℓ_1 have y-intercepts between 0 and 1. Even small errors in these intercept values (caused by rounding in the computer) could cause certain bars to be forced when they should not be, or bars that should be forced to no longer be forced. For this reason, we want to find some way of storing the positions of lines ℓ_0 and ℓ_1 using integer values.

To help with our implementation, we can modify the algorithm slightly so that it more readily adapts for use in a computer program. Our first step is finding a representation of the lines ℓ_0 and ℓ_1 that are as accurate as possible. Instead of storing the y-intercepts of the lines (which are real numbers), we can store the lines in a *point-slope* form. Point-slope form requires that we know the slope of a line and one point that the line travels through. Because we always know the slope of the lines is (τ), we can write an equation to find an arbitrary y value y_i for any arbitrary x value x_i:

$$y_i = y_{int} - ((x_i - x_{int}) * \tau)$$

We always know one point (x_{int}, y_{int}) because the lines are always raised or lowered so that they truncate to a certain value. The best way to make a line truncate to a particular integer value is to simply force the line to *pass through* that integer value. In Figure 3.4.2, raising ℓ_0 to intersect $(3, 5)$ is the smallest amount we can move ℓ_0 and still force the bar. To raise ℓ_0, instead of raising the y-intercept, we can just say that the line now goes through the point $(3, 5)$.

By storing a point that the line intercepts, we get to use integer values for our line representation, which is preferable to using real coordinates to store the

truncateClosed Given a value v, returns a new integer i such that $i \leq v$.

truncateOpen Given a value v, returns a new integer i such that $i < v$.

Figure 3.12: Definition of two truncation functions.

y-intercepts. Our line representation is now as accurate as our choice of τ for the slope of the line; all other values are perfectly precise. The only problem with this system is dealing with the line ℓ_1. When we raised the line ℓ_0 to an integer, we knew that it would truncate to that same integer (which was the property we wanted). However, when we lower ℓ_1, we want to set its value so that it truncates to the *next-lowest integer*. Returning again to Figure 3.4.2, if we chose to lower ℓ_1 to force it to truncate to 4, we would want to lower ℓ_1 to be *just below* the value 5 ($4.999\ldots$). That way, the next time we computed the y value for ℓ_1, it would truncate to 4, which is what we want. However, the value $4.999\ldots$ is not an integer, and we additionally cannot represent "the number that is closest to 5 but still less than 5" in a finite computer system. How then, do we represent the known intersection point of line ℓ_1?

We can overcome this problem by redefining how we truncate the y values of the two lines. We always know which y value we are truncating, so the easiest solution would be to treat truncations from the two lines ℓ_0 and ℓ_1 differently. Let's create two functions *truncateOpen* and *truncateClosed* which both truncate y values. The qualifiers "open" and "closed" refer to the mathematical terms used when describing intervals. *truncateClosed* truncates a value simply by removing the non-integer component of the value. *truncateOpen* also truncates a value in this way, *unless the value is already an integer*. If the value is an integer, it returns $(value - 1)$. The two *truncate* functions are defined in Figure 3.5.

Having defined these two functions, we now have a precise method of representing the two lines. We store the lines in point-slope form, using integer coordinates for the points. When we want to determine if a bar is forced, we get the y values for the lines ℓ_0 and ℓ_1, which we'll call y_0 and y_1. To compare these values, we first truncate them, using *truncateClosed* for y_0 and *truncateOpen* for y_1. Whenever we want to force a bar, we set the y value of the intersection point to the desired value for ℓ_0, or to *one more* than the desired value for ℓ_1. This will ensure that when the value is truncated, it will fall to the correct value.

Figure 3.13 shows an implementation of the geometric representation of a musical sequence. The functions **ForceBar**, which forces an arbitrary bar in a musical sequence, and **IsBarForced**, which returns **true** if a particular bar is forced in the musical sequence, are shown. These functions will be used often when we begin to query sequences of Ammann bars to find the empires of Penrose tiles.

Start of Algorithm **ForceBar**

Input:

M, a musical sequence
x_i**,** the x value of the bar to force, and
longer/shorter, whether the bar should be made *longer* or *shorter*

Output:

M, with bar x_i forced.

Begin:

Store the y value of ℓ_0 at x_i in y_0
Store the y value of ℓ_1 x_i in y_1
Store *truncateClosed(*y_0*)* in $trunc_0$
Store *truncateOpen(*y_1*)* in $trunc_1$
if we are making a long interval
then raise ℓ_0 so that it intersects $(x_i, trunc_1)$
else lower ℓ_1 so that it intersects $(x_i, (trunc_0 + 1))$

End of **ForceBar**

Start of Algorithm **IsBarForced**

Input:

M, a musical sequence
x_i**,** the x value of the bar to check

Output:

Returns boolean; true if the bar is forced, **false** if it is not

Begin:

Store the y value of ℓ_0 in y_0
Store the y value of ℓ_1 in y_1
Store *truncateClosed(*y_0*)* in $trunc_0$
Store *truncateOpen(*y_1*)* in $trunc_1$
return ($trunc_0$ **equal to** $trunc_1$)

End of **IsBarForced**

Figure 3.13: A pseudo-code implementation of the functions **ForceBar** and **IsBarForced**.

These algorithms give us the ability to quickly and easily deal with Ammann bar forcings. The first part of our implementation is now complete. What remains for us is to find the interactions between sets of Ammann bars in a plane, and then use these interactions to find Penrose tiles.

Chapter 4

Forcing Rules

In a Penrose tiling, a *forced tile* is one that must be placed in a certain position in order to preserve a legal tiling. Given an initial set, or *patch* of Penrose tiles, the set of tiles that are forced by this patch are known as the *empire* of the patch.

A patch can have both *local* and *non-local* tiles in its empire. Locally forced tiles share an edge with the patch, or with another locally forced tile, such that all the locally forced tiles are contiguous with the initial patch. These tiles can be found by studying a patch and discovering all the tiles that cannot be changed without affecting the legality of the tiling. Figure 4 shows the simplest example of a local tile forcing. The tiles shown form what is called the *Ace* configuration. It consists of a single Dart and two Kites adjoining it. There are no other tiles that legally fit into the concave portion of the Dart, and so those two Kites are always forced by the Dart.

We cannot realistically use an approach of trying all possible neighborhoods in order to find non-local forced tiles. As we move farther and farther away from the initial patch, the number of tiles whose positions would have to be tried would increase geometrically. Finding a forced tile at a large distance from the initial patch‘would take too long if the methods used were exponential in nature.

Fortunately, we have a tool that will help us solve the non-local tile problem.

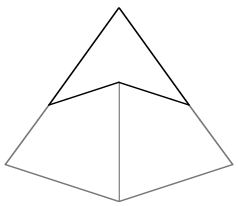


Figure 4.1: The Ace configuration: an example of local tile forcing.

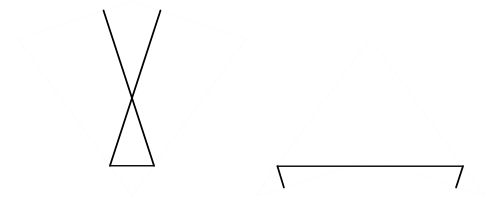


Figure 4.2: Penrose Kite (left) and Dart (right) with Ammann bars superimposed.

We know that Penrose tilings can be decorated by sequences of Ammann bars, as in Figure 4. These decorations serve as an alternative to the matching rules in a tiling.

We can think of the Ammann bars as conveyors of information. Because the Ammann bars are infinitely long, they affect the infinitely long "strip" of the tiling that they stretch through. When we force a new Ammann bar, this forces an infinite number of Ammann bars elsewhere in the plane. Our problem with the Penrose tiles was that we would have had to try all possible tiles to find the forced ones. But we can find forced Ammann bars without needing to try all the possible bar orderings, as we saw in the previous chapter. Because Penrose tiles can be thought of as mere decorations on top of the Ammann bars, we can convert Penrose tiles to Ammann bars, compute the forcings, and then convert the Ammann bars back to Penrose tiles. An overview of this process is shown in Figure 4. The initial set of tiles in (A) can be represented as the Ammann bars in (B). But these Ammann bars force new bars, in accordance to the rules discussed in Chapter 3. These new bars are shown in (C). If we now mark the Ammann bars with Penrose tiles, we have new tiles that were not in our initial set. These are the tiles that are forced by the initial patch, and are shown in (D).

Our strategy now is to find exactly how the Ammann bars relate to the Penrose tiles, so that we can begin with a patch of tiles, translate them to Ammann bars, find new forced bars, and then draw any new tiles that may be forced.

4.1 Finding Forced Shapes

Given a sequence of Ammann bars, we must be able to find portions of the Ammann bars that have features that we are looking for. For example, a certain arrangement of Ammann bars might represent a Kite, and we want to be able to find objects like Kites so that we can draw them.

Any arrangement of Ammann bars that has a feature we are looking for is called a *pattern*. We scan through the sequences of Ammann bars looking for patterns, and when we have found them we can take appropriate action, such

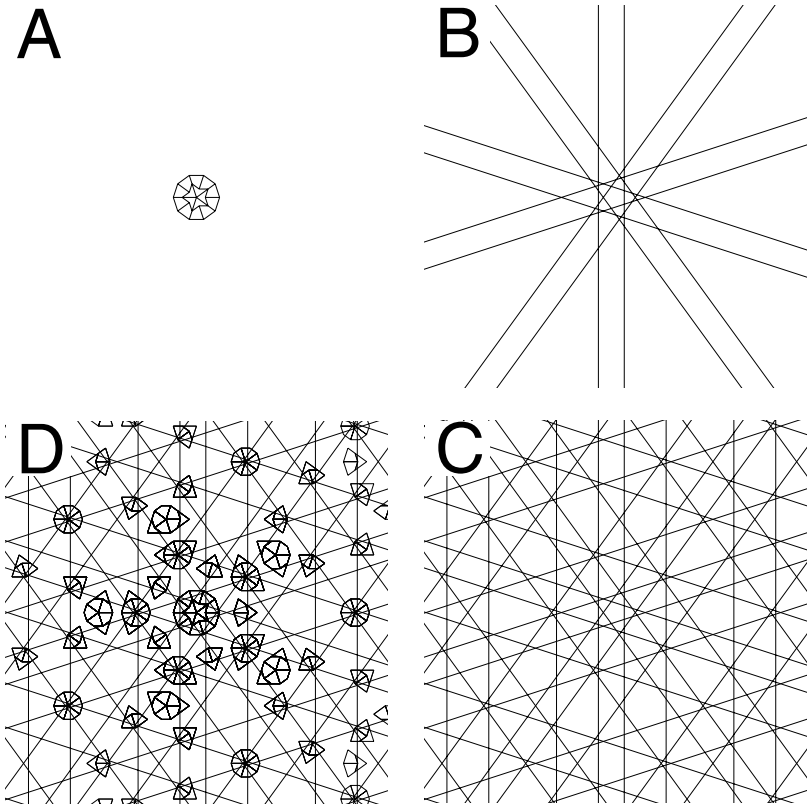


Figure 4.3: The four s in finding the empire of a patch of Penrose tiles, clockwise from upper left: (A) The initial patch of Penrose tiles. (B) The Ammann bars forced by the initial tiles. (C) The new Ammann bars forced by the initial bars. (D) The new Penrose tiles overlayed on the new Ammann bars.

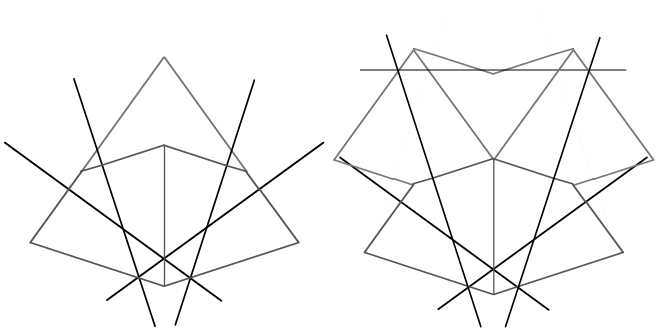


Figure 4.4: An example of a false match, using insufficient information. On the left, the Ace. On the right, the Queen. The bars in bold are shared by the two configurations, and so are not sufficient to match a unique pattern.

as drawing an object or forcing a new bar.

4.1.1 Finding Unique Patterns

In order to classify sections of the Ammann bars correctly, we must ensure that the patterns we are searching for are *unique*. Figure 4.1.1 shows two possible tile configurations that have the same Ammann bar configuration. If we were to match‘on this pattern, assuming it was an Ace, we might cause problems later on if it turns out that the pattern was actually part of a Queen.

The easiest way to define unique patterns of Ammann bars is to look at the *intersections* of the Ammann bars. This gives us a clear way to find distances, as well as angles between certain Ammann bars. The intersections form shapes that are similar to constellations in the night sky. In order to find a pattern, we must look for points that are certain distances away from each other, and that have lines through them that make certain angles with respect to each other.

To assist us with the pattern matching, it is helpful to think of the patterns as being formed of points where Ammann bars intersect. All the sets of Ammann bars have an orientation in the plane, so we can define the intersection points as being between Ammann bars that have certain orientations relative to each other. This will cut down on false matches because the additional information provided by the Ammann bar orientation prevents us from matching patterns that have the right shape but involve the wrong Ammann bars.

To create a pattern, we look at all the Ammann bars that are involved in the feature we want to match. In the case of Penrose tiles we look at all the Ammann bars that are involved with the tiles we are examining. We can then find the smallest unique set of intersections of the Ammann bars and use this as our pattern.

4.2 Matching Patterns

Once we have constructed a library of patterns that we want to find, we must devise an algorithm to actually locate the patterns for us. Because the Ammann bars can be rotated through angles that are multiples of 72°, the search algorithm must be able to find patterns that are rotated and translated, but *not scaled*. Because of the self-similarity of Penrose tilings, patterns may exist that have the right proportions but incorrect scale.

When we have found a match, we must also be able to accurately describe where it occurs. We have to be able to say where the pattern was found, what rotation it might have, as well as which Ammann bars are involved in the match. The reason for the last condition is that we may need to force other bars that are nearby, depending on what pattern we have matched. Knowing which bars are involved in the match will help in finding the nearby bars that may also be affected.

4.3 Forcing New Information

Some patterns may help us to add more information to a Penrose tiling. Suppose that every pattern we match has some amount of *unique* information associated with it that no other pattern possesses. Also suppose that not all of this information is required to uniquely identify the pattern, and that the pattern can exist without all of the information needing to be present. When this is the case, we can match on the smallest amount of information possible, and then fill in the missing information that we know must exist. Doing so augments the tiling and allows us to possibly find more forced information without needing more initial information.

As an abstract example, consider an animal-classification scheme. Suppose that we know an animal is a human if we know it has any two of the following properties: it uses language, it is a mammal, it stands on two feet, it uses tools. If you are told that a certain animal stands on two feet and uses tools, then you can uniquely classify it as human, because you have enough information to do so. Now, if you are asked whether the animal can talk, you can incorporate the other information associated with humans and answer "yes." This same idea carries over to our Ammann bars. Once we know enough to uniquely identify a pattern, we can rightfully add any missing information to the sequences of Ammann bars, possibly forcing new bars that will help us later.

More concretely, there are two patterns that we currently employ to force more information in a Penrose tiling. There may be other patterns that have not yet been discovered which force even more information. We hope that this research will allow more of these rules to be discovered.

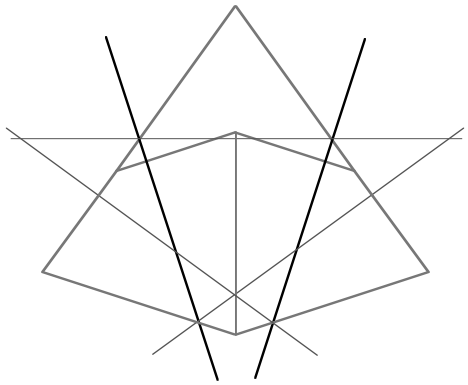


Figure 4.5: The two bars (shown in bold) from the Dart (Ace) that are not required to be forced. Only one of these bars must be forced for the Dart to be forced.

4.3.1 Darts

A Dart always forces the Ace configuration, as mentioned earlier. There are five Ammann bars that are forced in an Ace, but Minnick has shown that of these five, one of two special bars may remain unforced. The two special bars are shown in Figure 4.3.1. If one (but not both) of these bars is unforced, the entire Dart configuration is still forced. When one of these bars is unforced, and we find the Dart configuration, we can force the missing bar and add information to the tiling.

4.3.2 Minnick's Double Kite

Minnick discovered what we call the "Double Kite" forcing pattern. When two Kites are aligned so that their short edges coincide, a single Ammann bar runs directly through the two Kites. However, another Ammann bar is forced at a distance equivalent to a *short* interval away, as in Figure 4.3.2. This line is forced because all possible legal tile configurations that contain the Double Kite shape all have this Ammann bar forced in this location. By exhaustive proof, Minnick showed that the Double Kite formation always forces this Ammann bar. We include it as one of the patterns we search for.

4.4 Drawing Tiles

Some of the patterns that we search for will simply be used to identify individual Penrose tiles. When we have found a certain tile, we can use the position and rotation information to draw a correctly oriented Penrose tile on top of the Ammann bars. This step will allow us to display the forced Penrose tiles visually after we have computed an empire.

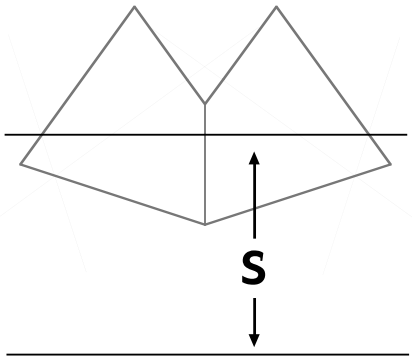


Figure 4.6: The "Double Kite" formation. Although the two Kites do not directly force any Ammann bars, they force an Ammann bar at a distance **S** from the Ammann bar that passes through both Kites.

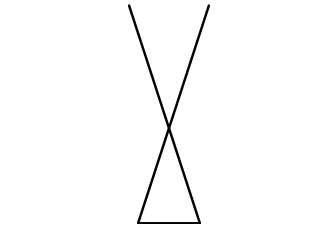


Figure 4.7: A Kite tile, with the "Kite Triangle" Ammann bars in bold.

4.4.1 Kites

The Kite tile contains three Ammann bars which form a small triangle in the center near the base of the Kite, as in Figure 4.4.1. Ammann bars that form a triangle with these angles and side lengths are not found in any other shape but the Kite. Thus, when we find this exact configuration of Ammann bars, we know that a Kite is present.

The Kite does not force any additional information by itself, so finding this pattern does not allow us to add any new information to the tiling. When we render the final empire as a set of Penrose tiles, however, we must look for this pattern so that we know where to draw Kites as part of the Penrose tiling representation.

4.4.2 Darts

The Dart configuration, mentioned previously and shown in Figure 4.3.1, is also matched for drawing purposes. When an empire is finally rendered as Penrose tiles, this pattern must again be matched so that the Dart tile can be drawn in the tiling.

4.5 Review

Patterns of Ammann bars allow us to find regions in the plane that interest us. These areas of interest may be areas where tiles can be drawn or new Ammann bars can be forced. We create patterns so that they are unique, which prevents us from drawing or forcing incorrectly. When we match a pattern, we fill in any missing information (thereby forcing new Ammann bars), apply any forcing rules (such as the Double Kite forcing rule), and draw any tiles. Doing so allows us to find as much forced information in the tiling as possible.

Chapter 5

Implementation

We have now discussed all of the s necessary to find partial[1] empires of Penrose tiles. We have outlined the relationship between Penrose tiles and Ammann bars, which will allow us to convert from one representation to the other. We have explained algorithms that will allow us to deal with arbitrary forcings in sequences of Ammann bars, which will allow us to get long-range forcing information in a Penrose tiling. We have explained how we plan to find patterns that interest us in the sequences of Ammann bars, and how we exploit these patterns to obtain information about the Penrose tiling, as well as where we should render Penrose tiles.

5.1 Overview

Now we must actually implement these algorithms. This chapter will discuss the methods used to find the partial empires of Penrose tiles. We have chosen to leave some details out, as they would only cloud the understanding of the software. Such details will be noted in the text, but they will not be addressed in the algorithms presented here.

Our software assumes that we are beginning with five sequences of Ammann bars, arranged in a plane so that they define a Penrose tiling. We could have started with tiles, and then had the software perform the additional step of translating the initial tiles to Ammann bars. Giving exact coordinates of Penrose tiles is no easier than simply defining the initial Ammann bars, however, so our software simply begins with the Ammann bars.

[1] We emphasize here that the empires we find are as complete as possible, but are not necessarily the *entire* empires of any set of Penrose tiles.

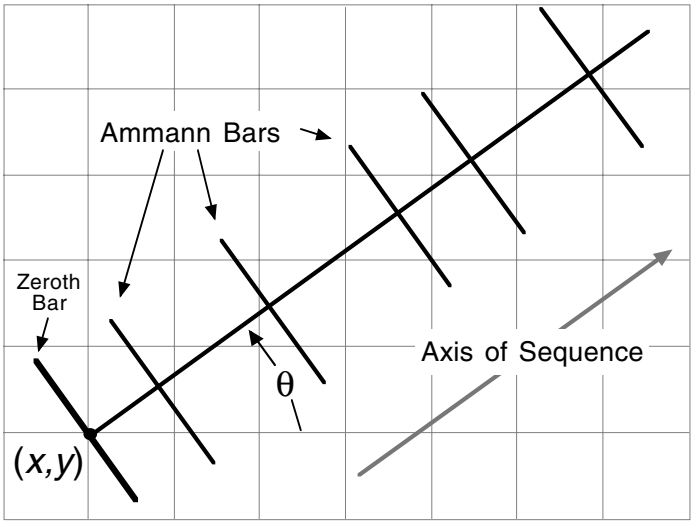


Figure 5.1: A sample Ammann bar sequence, with center (x, y) and rotation θ.

5.2 Musical Sequences and Ammann Bars

As described in Chapter 3, we have implemented Minnick's geometric algorithm for musical sequences. We have augmented the Ammann bar representation so that a sequence of Ammann bars can be placed precisely in a two-dimensional space. Our Ammann bars have a "center," which is the *zeroth bar* of the sequence. All other bars are numbered relative to this bar.

Our Ammann bars also have a *rotation*, defining their orientation in the plane. Because five sequences of Ammann bars interact to form a Penrose tiling, we must be able to represent sequences of Ammann bars that have been rotated through multiples of 72°.

Finally, our Ammann bars have a *center point*. This is a point that the zeroth bar passes through. This allows us to precisely align the Ammann bars in the plane so that they interact properly. There is no point in a Penrose tiling where all five sequences intersect, so we cannot simply have all the sequences of Ammann bars centered at (0,0) in the plane.

Our final representation looks like Figure 5.2. The axis of the sequence passes through the *center point* (x, y), and has an orientation θ, measured in degrees from the x-axis. The Ammann bars themselves are perpendicular to the axis of the sequence.

As shown in Chapter 3, we have routines to force arbitrary bars at a certain distance from the origin. We also have routines that will return whether a certain bar is forced or not.

While they will not be used in our conceptual explanations, our software also contains routines to convert bar numbers to two-dimensional coordinates

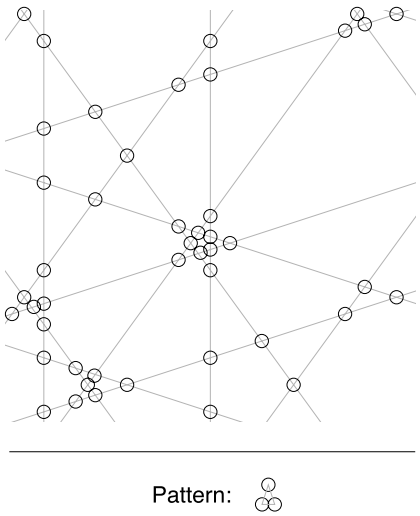


Figure 5.2: A sample field of intersection points. The Ammann bars are shown in light grey. A sample pattern like one we might search for is shown at bottom.

that lie along the axis of a sequence, as well as convert these coordinates back to bar numbers. This allows for easier comparisons of points in the plane.

5.3 Intersections

Given five sequences of Ammann bars, our first task is to find all of the intersections of these Ammann bars. Intersections provide us with the easiest way to find relatively positioned Ammann bars, by outlining the shapes that they make. We only test the *forced* Ammann bars when we compute our intersections, because only the forced bars correspond to forced tiles.

We do not simply store two-dimensional coordinates. Rather, we store the intersection of two bars from two sequences. For example, if the fifth bar from the sequence with rotation 72° intersects the second bar from the sequence with rotation 216°, then the intersection is said to occur at $((72°, 5), (216°, 2))$. By storing the rotation and bar number, we make our calculations exact by avoiding round-off errors associated with real coordinates. Additionally, they make our pattern matching easier later on by allowing us to analyze the relative positions of the Ammann bars.

Figure 5.3 shows the intersection points for a small section of the plane. A pattern like one that we might search for is shown below.

To find these intersection points, we must test each sequence of Ammann bars for intersection with the others. Because the Ammann bars are arbitrarily long, and because no sequence of Ammann bars is parallel with any other sequence, every Ammann bar from every sequence intersects every Ammann bar

Start of Algorithm **FindIntersections**

Input:

Ammann bars, an array of five sequences of Ammann bars

Ranges, Bars in each musical sequence that should be checked for intersection

Output:

Returns a **set** of bar intersections, in the form of $((A_{orientation}, A_{barnumber}), (B_{orientation}, B_{barnumber}))$

Begin:

Loop for every sequence, **SequenceA**, in the array of Ammann bars
Loop for every *remaining* sequence, **SequenceB**, in the array of Ammann bars
For Each bar in **SequenceA** and each bar in **SequenceB**:
(Perform Range-checking on the bar number)
If the bars intersect **Then Store** the intersection value in the set to be returned
Return the set of Intersections

End of **FindIntersections**

Figure 5.3: A pseudo-code implementation of the Intersection-finding algorithm

in every other sequence. Therefore, it is not unreasonable to test every bar for intersection. In a real system we would limit the intersection test to only return intersections in some finite area of the plane, so we must add in a small check to confirm that the intersections are computed only for bars that fall within a certain range. This step is not fleshed out in the pseudo-code for the intersection routines is shown in Figure 5.3, but is easy to implement.

5.4 Pattern Matching

After the intersection routines have been run, we have a set of intersection points that we can begin searching through for patterns. Because searching for certain configurations of points is similar to searching for constellations in the night sky, we say that we search for a *constellation* in a *sky* of points.

A *constellation* is the template pattern that we are searching for. It is defined in its own plane in a way that is easy for the researcher to describe. A sample constellation is shown in Figure 5.4. It is a small triangle, made from three Ammann bars intersecting like the ones inside a Kite.

Because our constellations may appear at different positions and rotations throughout our sky, we must have a mechanism that can find the constellation despite these disturbances. When we have found a constellation in the sky, we have created a *mapping*. A mapping details the rotation and translation necessary to map the prototypical constellation to an actual pattern match in the sky. The mapping is needed when we take action on the pattern, as we need to know how to find it and its surroundings to perform actions such as forcing new bars and rendering tiles.

The simplest approach to finding constellations in the sky is to simply it-

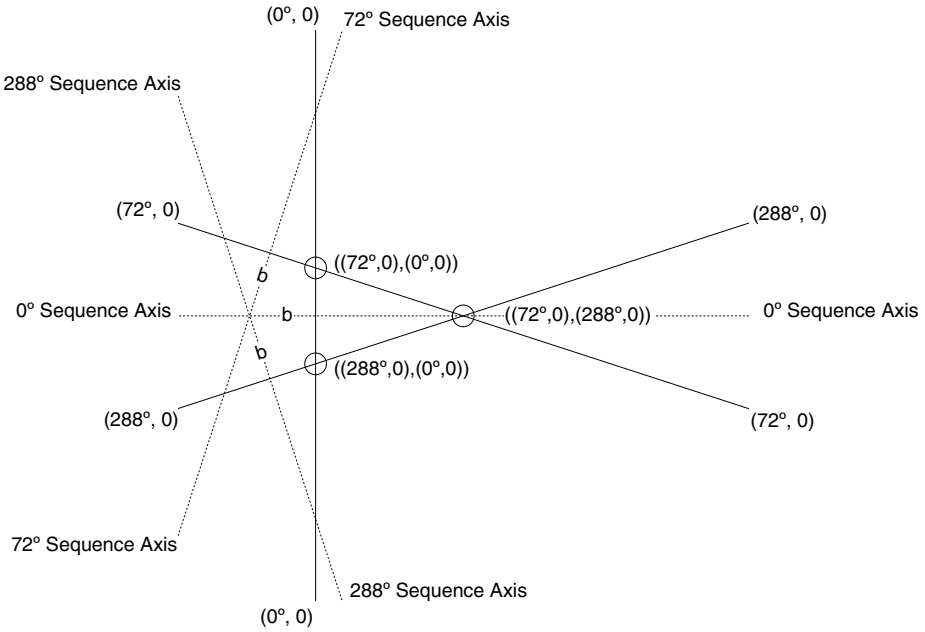


Figure 5.4: A sample constellation. Note that the constellation is defined at an angle and position that is easy to compute.

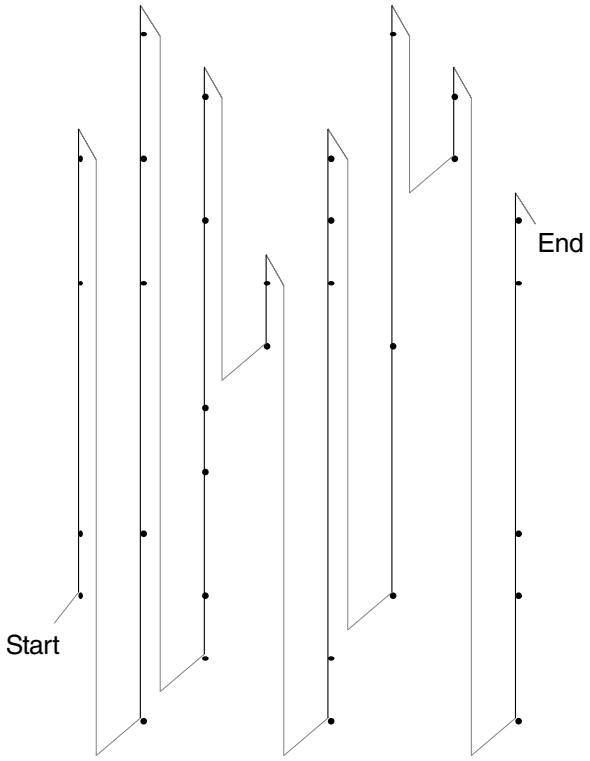


Figure 5.5: A sample ordering of points in a plane, from left to right, top to bottom. Iterating over the points in order results in the path shown.

erate over all of the points in the sky, and check to see if it is part of a valid constellation. Obviously, this method would be too time-consuming in practice. Therefore, we must adopt a slightly more intelligent algorithm.

Our algorithm begins by sorting all of the points in the sky into a dictionary ordering. We chose to sort the points leftmost to rightmost, and for coordinates with the same x value, bottommost to topmost. This allows us to iterate over the points in the order shown in Figure 5.4.

We now consult our constellation. We use a pair of points that are a known distance δ apart from each other. In practice, it is best to choose a pair of points with a distance between them that does not appear *except in this constellation.* This way, less time will be spent later on when we verify matches. Additionally, we should choose δ to be as small as possible, so that our scan line optimizations (which will be discussed later) are more effective. It should be noted, however, that the algorithm will still function correctly even if a poor choice of δ is made. The constraints on δ are made only so that the algorithm will run faster. Figure 5.4 shows our sample constellation with this pair of points marked.

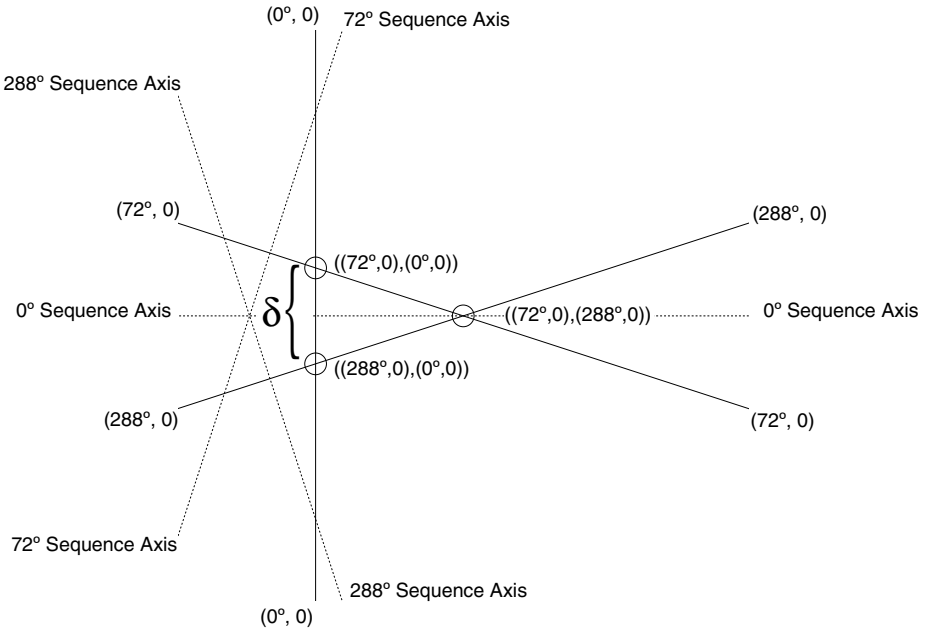


Figure 5.6: A constellation with the unique point-pair marked. The points are separated by a distance δ.

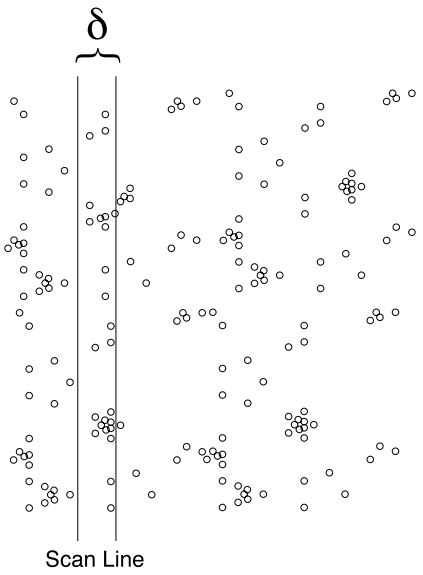


Figure 5.7: An example of the scan line used in the pattern-finding algorithm.

To reduce the practical number of points that must be compared in this algorithm, we use a *scan line* to define the set of points that are currently being considered as part of a match. The scan line has a width of δ, and only the points which fall within this scan line are considered eligible for matching. Because the size of δ determines how many points are under consideration, smaller values of δ are desirable because they reduce this number of points.

The actual pattern-searching algorithm works as follows: After ordering the points, we iterate over them one at a time. The point currently under consideration is compared against all the points in our scan line. If any of the points are exactly distance δ away, the two points are stored as a pair for further consideration. Once all the points in the scan line have been checked, the scan line advances to include the current point. Any points that were farther than δ away on the x-axis have now left the scan line, and are no longer under consideration. However, since we are only looking for points that are exactly δ away, we know that those *farther* than δ away need not be considered. This is the purpose of the scan line: to limit the points under consideration to those that could possibly be close enough.

Once we have iterated over all the points in the sky, we are left with a set of point-pairs which potentially belong to the pattern we are searching for, because the point-pairs correctly match part of the constellation. We now consider each of these point-pairs in turn, and determine if they actually form the pattern we are looking for. If they do, then a mapping has been constructed, and we store this for later use. If they do not, then the point-pair is discarded as incorrect.

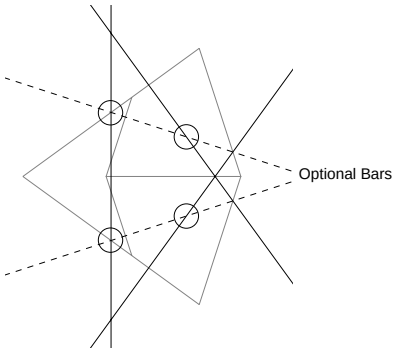


Figure 5.8: An example of how forcing information contained in the constellation is applied to force new information in a tiling.

We repeat this process for all the different kinds of constellations that we want to search for. We associate the mappings with the constellations they belong to, resulting in a *pattern match*. These pattern matches will be used to force new bars and render tiles.

5.5 Feature Forcing

Given a collection of *pattern matches*, we must force any new Ammann bars that we know must be forced by the presence of other bars. A constellation can store both the bars that are *required* to be matched, as well as those that can *optionally* be matched (the Dart is an example of this). If this information is included, then forcing new information is relatively straightforward. to force new information, we first map our constellation to the pattern in the sky. The mapping we use was found earlier, when we searched for patterns. To force new information, we consult the constellation, and see where forcings should occur. We map those forcings out to a real location in the plane, and then apply them.

As an example, consider Figure 5.5. The constellation shows us where the intersection of two specific Ammann bars should occur. Naturally, we cannot simply force bars using the addresses provided in the constellation; they describe a different location and rotation than the one we wish to force. Therefore, we apply the mapping, which changes the addresses of the intersections involved. With these new addresses, we can apply the forcings correctly.

A more general algorithm to use when forcing new information is given in Figure 5.9. The algorithm assumes that we have some constellation, with the required intersections labeled separately from the optional intersections. Since we have already found the required intersection pattern, we may use those points as a base reference for the other intersections. By knowing the placement of one Ammann bar in our constellation, we can find all other bars relative to it, and force them if necessary.

In more detail, here is how the algorithm works: We must know the position of at least one intersection of Ammann bars, otherwise we could not have made a match. Knowing the position of this intersection, we can construct a relationship between the Ammann bars of the constellation and the Ammann bars of the actual pattern in the plane. For example, let's say that we have some known point **A**, which has the coordinates $((72°, 1), (216°, 1))$. However, in our real pattern, this intersection occurs at $((144°, 12), (288°, 14))$. The bar numbers are not as important as the fact that the *sequences* have changed. Our whole template has been rotated 72°, and so all the other intersection addresses must be altered to compensate for this.

To force an optional intersection, we first determine what sequence it belongs to by adding in the rotation found in the s just described. We now know which sequences the intersection lies on in our pattern. Now, we map the constellation point out into the plane, and then determine the Ammann bars that it must lie on. Since we know the sequences, and we know the point that the forced Ammann bars must intersect at, this becomes a simple matter of converting the coordinates to distances along the musical sequences, and then converting the distances to bar numbers. Having done this, we can force the bar numbers, and the operation is complete. Repeating this procedure for all the optional bars will ensure that any information that should be forced is forced.

5.6 Feature Drawing

After we have forced all of the new information in a tiling, the only step that remains is to convert the Ammann bars back into Penrose tiles for final rendering. This is relatively simple, and works in a similar fashion to the forcing algorithms.

To draw a tile in the correct area of the plane, we need only to have a drawing that matches our constellation. Grünbaum and Shephard [GS87] explain the relationship between the Ammann markings and the Penrose tiles, and by applying this information we can construct a drawing that overlays our constellation perfectly.

Once we have this information, drawing the tile is elementary. Any pattern match in the plane has a mapping which describes how to translate and rotate the constellation to match the pattern in the plane. Because our drawing is simply an overlay on this constellation, drawing the tile simply requires translating and rotating it with the same mapping. Once we have done so, the drawing is in the correct place.

5.7 Pattern Matching Algorithm In Detail

We have now described the basic algorithm for finding forced Penrose tiles in the plane. We will now return to the portion of the algorithm which deals with the actual matching of patterns in the plane and treat it in more detail.

Start of Algorithm **ForceNew**

Input:

Ammann bars, an array of five sequences of Ammann bars

Constellation, the set of intersection points that define the pattern we are looking for. **Constellation** should contain *required* intersections of Ammann bars, which must have already been matched, and *optional* intersections, which are those that will be forced by this algorithm

Mapping, the transformation that maps the template constellation to a real position in the plane

Output:

Alters the Ammann bars so that any bars that are forced by the presence of the constellation are now forced

Begin:

Map a known intersection point from the constellation to a real point in the plane and **Store** it in **BaseIntersection**

Compute the difference between the known point's sequence rotation and the sequence rotation of **BaseIntersection**, and **Store** this value in **BaseRotation**

For Each intersection point that is *optional*:

Calculate the sequences that the point must lie on, using **BaseRotation** as a compensation value

Map the point to a real position in the plane

For Each sequence that this intersection point involves

Calculate the distance along the sequence where the point lies
Calculate the bar number that this distance corresponds to
Force the bar number for this sequence

End of **ForceNew**

Figure 5.9: A pseudo-code implementation of a function to force new Ammann bars in a Penrose tile.

5.7.1 Scan Line Algorithm

The scan line is used to limit the points that are under consideration during our initial search for patterns. Earlier, we only mentioned that the scan line "moved" across the sky of points so that we were only considering points that were close to the current point that we were looking at. We will now return to this concept and further develop our algorithm.

When we begin our search through the sky of points, we must already have a distance δ, derived from some pair of points in the constellation. This distance should be one that does not occur frequently outside of the constellation. The scan line is as wide as this value δ to prevent consideration of points that are obviously farther than δ away. If a point has an x value that differs from the current point by more than δ, then it is guaranteed to be too far away. Because of this, δ should also be as small as possible so that we can keep the number of points under consideration as small as possible. Note that both of these constraints on δ are not mandatory; they are used to make the algorithm run faster, but are not required for it to function correctly.

To implement the scan line, we use a queue. As points are added to the scan line, they are pushed onto the tail of our queue. As points leave the scan line they are popped off of the head of the queue. If our scan line moves from left to right, the leftmost points (those that have been in the scan line longest) are those that are closer to the head of the queue.

The queue algorithm is shown in Figure 5.10, and works as follows: the points at the head of the queue are compared against the current point. If the point at the head of the queue has an x value that differs by more than δ from the current point's x value, it is popped off the queue. This is repeated until the point at the head of the queue has an x value which is within δ to the current point. This is how we "move" the scan line to the right. The points that are popped off the queue correspond to those that leave the scan line as it moves to the right.

Our queue now only contains the points that *might* be the correct distance from our current point. We iterate over these points, calculating the distance between them and the current point. If we compute a distance of δ, then the pair of points (that is, the current point and the one in the queue that is δ away), is saved for further consideration.

When all of the comparisons have been made, the current point is added to the tail of the queue, the next point in the sky is made to be the current point, and the algorithm begins again. When we have iterated over all the points in our sky, we are left with pair of points that are the correct distance δ apart.

5.7.2 Mapping Verification

Once the scan line algorithm has been run, we have pairs of points that are some distance δ apart. These point pairs represent *potential* matches, and must be checked further. The reason they are potential matches is that they are the correct distance apart to be one pair in our constellation. The distance δ was

Start of Algorithm **ScanLine**

Input:

Sky, a set of Ammann bar intersection points

δ**,** a known distance in the constellation. We are searching for points that are exactly this distance apart

ScanQueue, an initially empty queue that will hold points. Points are added to the tail and removed from the head

Output:

Returns a **set** of pairs of points. These pairs are points that are exactly δ apart from each other

Begin:

For Each intersection point **J** in the sky:

While the x value of the intersection point at the head of the **ScanQueue** differs from the x value of **J** by more than δ:

Remove the point at the head of the **ScanQueue**

For Each intersection point **K** in the **ScanQueue**:

If the distance from **J** to **K** is δ
Then Store the point pair **{J,K}** to be returned
Else *(nothing)*

Add J to the tail of **ScanQueue**

End of **ScanLine**

Figure 5.10: A pseudo-code implementation of the scan line queue from the pattern matching algorithm.

chosen from the distances in our constellation, and so these point pairs represent points that would fit in our constellation.

We must now verify that the other points in the constellation also exist in the sky, near the points that the scan line algorithm found. To do so, let us assume that we have a point pair, found by the scan line algorithm, consisting of points **{J,K}**. We shall also assume that these points are the same distance δ apart as the points **{M,N}** from our constellation.

The first thing to do is to construct a *mapping*. A mapping is a formula that will translate and rotate a constellation point to some place in the plane where the constellation is supposed to occur. Mappings are used by the later algorithms, as we saw in the forcing and drawing algorithms discussed earlier. We can build a mapping from only two points in the following manner.

Determine a translation that will move point **M** to point **J**. This is the translation part of the mapping, expressed as $(\Delta x, \Delta y)$. Computing these values is as simple as

$$\Delta x = J_x - M_x \qquad \Delta y = J_y - M_y$$

We have now mapped the first points from the pairs. Using the second points, we can get the rotation required for the mapping. Conceptually, the way this is done is to first translate the points so that the first two (**M** and **J**) coincide, and then simply determine the angle θ which we must rotate **N** through (with respect to **M**) to coincide with **K**.

Mathematically, we can represent this using the arctan function. If we translate the points by Δx and Δy, then they both have the same center of rotation at $(0, 0)$. If we compute the rotations with respect to the origin, as follows:

$$K_{rot} = \arctan \frac{K_x}{K_y} \qquad N_{rot} = \arctan \frac{N_x}{N_y}$$

If we subtract K_{rot} from N_{rot}, we are left with the angular difference of the two. This is the rotational component θ of the mapping. Note that this angular computation does not confirm that the points **K** and **N** actually coincide; it merely finds the angle between them. A final check should be made to ensure that the points actually line up.

We now have a complete mapping, with translation components $(\Delta x, \Delta y)$ and rotational component θ. With it, we can map any point in our constellation to a real point in the sky. The two points in our constellation that were separated by a distance δ trivially map to the correct points in the sky. But the rest of the constellation needs to be verified. to do so, we simply map every point in the constellation with the mapping we just constructed. The resulting point is a point that *must* exist for the constellation to be found at that point in the sky. We test for that point in our set of sky points. If it exists, then we continue checking the constellation. If it doesn't exist, then we can stop. When we have finished, if all the mapped constellation points exist in the sky then the mapping is valid and the pattern has been matched.

Start of Algorithm **PatternVerify**

Input:

Sky, a set of Ammann bar intersection points

Constellation, the constellation that defines the pattern we're looking for

PointPairs, a set of pairs of points that correspond to a certain pair of points in our constellation, **{M,N}**. **PointPairs** should be generated by the **ScanLine** algorithm

Output:

Returns a **set** of **Mappings**, which map the **Constellation** to all the locations in the plane where it occurs.

Begin:

For Each pair of points **{J,K}** in the set of **PointPairs**:

Compute a **Mapping** from points **{J,K}** to **{M,N}**

For Each point **P** in the **Constellation** (excepting **M** and **N**):

Map P using the **Mapping** to point **P'**
Search for point **P'** in the **Sky**
If the point is found
Then Continue
Else Exit For Loop

If all the points in the **Constellation** mapped successfully

Then add the mapping to the **set** to be returned

Compute a reversed **Mapping** from points **{J,K}** to **{N,M}** and **repeat** the **Constellation** point verification For Loop above using this **Mapping**.

End of **PatternVerify**

Figure 5.11: A pseudo-code implementation of the pattern verification algorithm.

This is how we verify the point pairs that we obtain from our scan line algorithm. The algorithm for this pattern verification is shown in Figure 5.11. One final note: as shown above, the mapping is constructed by arbitrarily corresponding point **J** with **M** and point **K** with **N**. Obviously, there is no reason why the points couldn't be associated the other way (**J** with **N** and **K** with **M**), because the pattern may be flipped over. In order to deal with this, we should always test *two* mappings per point pair, to ensure that we've checked both possibilities.

5.8 Complexity

As noted in Chapter 3, we would like the algorithms that we use to be as efficient as possible. The algorithms above should allow us to find the empires of arbitrary patches of tiles. We are interested in knowing how long the algorithms will take, so that we can predict the performance and search for improvements. We will consider the various pieces of our empire-finding algorithms below in an attempt to determine the overall complexity.

5.8.1 Ammann Bars

In order to get the intersection points that we will search through for patterns, we must first find all the intersections. To do this, every Ammann bar must be compared against every Ammann bar from the other sequences exactly once. If there are m Ammann bars per sequence, then we have a factorial number of comparisons. The first sequence must have its m bars checked against the other 4 sequences' m bars. The second must be checked against 3 other sequences, and so on. With 5 sequences total, the number of comparisons can be represented by

$$4m^2 + 3m^2 + 2m^2 + m^2$$

Dropping the coefficient, we have a complexity of $O(m^2)$, dependent on the number of Ammann bars per sequence, m. The result of these comparisons, however, is some number of points n. Because each intersection produces one point, and because *every* bar interacts with every other, each test results in one point. Thus, the overall complexity of the algorithm, in terms of the number of points in the sky, is $O(n^2)$.

For the actual scanning of points, several factors work against us. Every point in the sky is potentially part of a pattern. We must therefore inspect every point at least once, which requires n inspections.

When we inspect a point to see if it is part of a pattern, we must see if there is some other point in the sky that is a distance δ away. If we did not use the scan line algorithm, every other point in the sky would have to be checked to see if it was the correct distance away. This would also require n comparisons, and so the total complexity of just finding candidate pairs alone would be $O(n^2)$.

After we have found the candidate pairs, they must be further tested to see if they form a true pattern or not. As we saw above, doing so requires constructing a mapping, and then mapping all the points in a constellation and verifying that they appear in the sky. Constructing a mapping is a constant-time algorithm (a fixed number of simple operations), and so it does not affect our run time. Also, if we assume that the number of points in the constellation is relatively small compared to n, then this too has no impact on our order of magnitude. However, each constellation point that is mapped must be searched for in the sky of points. Assuming we store the points in a structure that is efficient to search, such as a binary search tree, our search time is $O(n \log n)$. This will dominate the verification algorithm's run time, and so the verification step alone is $O(n \log n)$.

This verification step must be performed for all pairs of points that are the correct distance δ away. In the worst case, all n points would match this distance δ, and so the verification step would be performed n times. This is why choosing a "rare" value for δ is extremely important; it k the number of verifications to a minimum if we choose a value for δ that does not occur frequently. Assuming that the number of candidate pairs is much smaller than n, the verification step does not adversely affect our run time.

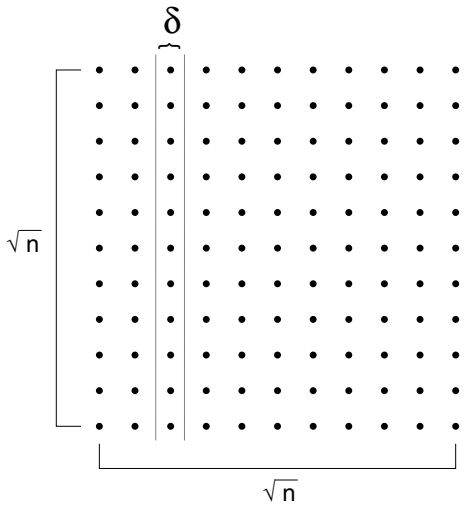


Figure 5.12: A graphical demonstration of how the scan line reduces the complexity of the pattern matching algorithm. The total number of points is n, but only $\sqrt{n}$ points are under consideration at any time.

Adding up the s of the Intersection algorithm, the pair-finding algorithm, and the verification algorithm, we would find that the run time of the whole pattern matching algorithm to be

$$O(n) + O(n^2) + O(n \log n) \approx O(n^2)$$

Clearly, the $O(n^2)$ term will dominate the complexity. What this means is that we should try and make the pair finding algorithm as efficient as possible. We do this through the use of the scan line described above. The scan line heavily limits the number of point comparisons that we perform.

For computational purposes, assume that our n points are evenly distributed throughout the sky. This means that we can draw a grid such that one and only one point falls within each square of the grid. A simple calculation will show that the dimensions of our sky can be expressed as $\sqrt{n}$ units wide and $\sqrt{n}$ units tall, because multiplying the two together yields n, with one point per square.

Assume that we make δ small enough so that it is on the order of $\sqrt{n}$. This is not unreasonable; δ is roughly the size of a single Penrose tile, so any empire involving a large number of tiles will quickly grow so that δ is relatively small. If δ is the same size as $\sqrt{n}$, and if our scan line is as wide as δ, then we are considering $\sqrt{n}$ points at any given time, because that is how many points will be found inside the scan line. Thus, rather than considering all n points in the sky to find a pair, we consider only $\sqrt{n}$. This concept is demonstrated in Figure 5.8.1.

The complexity of the algorithm is reduced from $O(n^2)$ to $O(n\sqrt{n})$. While this is not as good as $O(n \log n)$, it does represent a tangible difference in the running time of the algorithm.

5.8.2 Further Optimizations

The scan line algorithm allows us to reduce the run time of our pattern matching algorithm. In addition to this, we have implemented a partitioning scheme that improves the run time of our pattern matching algorithm even more.

Conceptually, the partitioning scheme works as follows: given a square section of the plane to search for intersections, the partitioning scheme divides it into four smaller squares. The partitioning scheme is recursively called on each of these smaller squares, which will continue to be divided up until the squares have reached some predetermined size that is considered small enough to work on.

In practice, the algorithm is very similar. The first consideration is the loss of patterns on the borders of the squares. If a pattern was split by the boundary between two squares, then it would fail to be recognized by either of the two independent pattern matches in both of the squares. to prevent this from happening, we make sure that the squares overlap slightly, such that the overlap is the same size as the largest pattern we will ever look for. This way, any pattern is guaranteed to fall completely inside one of the two squares (or both).

The second consideration is determining when we have divided the squares up "enough." Dividing up squares introduces some memory and computational overhead, and we don't want to make our problems worse. Therefore, we should stop partitioning before it becomes more inefficient to partition than to just compute the intersections for the whole square.

Our software takes both of these factors into account. It does not partition unless the area to be considered is sufficiently large, and when it partitions it makes sure that all the squares overlap by at least as much as the largest pattern that is being searched for. In doing so, the run time of the software noticeably improved. Even more importantly, memory consumption dramatically decreased, because the software did not have to store large numbers of intersections while it searched for patterns. By converting the partitioning scheme to an iterated system, we virtually eliminated the problem of running out of memory, because we only needed enough memory to consider the smallest partition size plus the space to store the other partition information.

With these optimizations, our software is capable of rendering large portions of Penrose Empires in a reasonable amount of time. With this new tool, the search for more forcing rules in Penrose tilings can begin.

Chapter 6

Results

Having implemented the algorithms from the previous chapter in software, we can now produce empires of any arbitrary set of initial tiles. Note that the algorithms we use are not necessarily complete; there may still exist forcing rules unknown to us. We therefore stress that the figures shown in this chapter are *partial* empires of tiles. There are more tiles shown here than in other previous works, but we do not claim to have complete empires shown here.

6.1 Verified Empires

The configurations of tiles that have been studied the most are those of the *seven vertex configurations*. These seven arrangements are the only ways that Penrose tiles can legally arrange themselves around a single point. For this reason it is natural to begin our exploration of empires here.

Each configuration must be defined in terms of the Ammann bars that force the tile configuration. This step is not trivial, and so we have included diagrams of the initial bar forcings for each vertex configuration. The notation used for distances comes from Minnick, who cataloged the distances between the vertex points of Penrose tiles and the Ammann bars in her research. Her table of these distances is included for reference in Figure 6.1.7. Additionally, Figure 6.1.7 and Figure 6.1.7 show the Dart and Kite tiles with these distances marked.

In these initial Ammann bar configurations, we show only the initial forced tiles and the Ammann bars that cause them to be forced. There are often locally forced tiles as well (caused by the presence of a Dart). In the interests of clear diagrams, however, these local tiles have been omitted, as have the Ammann bars that force them.

In the center of each diagram is a point where all the axes (shown using dashed lines) of the musical sequences intersect. This corresponds to the point $(0, 0)$ in the plane. Each musical sequence has an angle that is an integer multiple of $72°$, and axes are labeled in terms of their angles of rotation.

The Ammann bars are defined as being forced a certain distance away from

the center point $(0, 0)$. The tables below the figures show the distance that the zeroth and first (if necessary) bars are from the center point. The distances used are those defined in Figure 6.1.7. If a distance is negative, it simply means that the distance should be applied at 180° to the axis of the sequence. In other words, if we want to travel a distance $-x$ along a sequence with angle 72°, we simply travel a distance x at an angle of 252°.

For each vertex configuration we have shown a diagram of the Ammann bars that force the tiles, as well as a short table of initial bar distances from the zero point. If two bars are forced in a single sequence, the interval between those bars is also listed.

6.1.1 The Ace

The Ammann bars required to force the Ace are shown Figures 6.4 and 6.5. The final empire of the Ace is shown in Figure 6.1.7. The Ace configuration alone does not force any other Penrose tiles. Looking at the Ammann bars, the reason why becomes obvious. Only one Ammann bar is forced for each musical sequence. Because one bar does not force any others, no Ammann bars besides the initial ones are forced, which in turn causes no tiles to be forced. This figure is included to demonstrate that the software correctly finds the empire of the Ace. The scale is large to show that no extra tiles have been introduced by the software.

6.1.2 The Deuce

The Ammann bars required to force the Deuce are shown Figures 6.7 and 6.8. The final empire of the Deuce is shown in Figure 6.1.7. Like the Ace configuration, the Deuce only forces one bar per musical sequence. Unlike the Ace, however, the Deuce forces other non-local tiles. This happens because our bar forcing rules allow us to force new bars from our initial set.

A Dart has optional bars that do not need to be matched. The Deuce configuration lacks one optional bar from both of its Darts; we can fill in these bars because we know they must exist. After we have done so, there are now two bars in two of the musical sequences. This causes more Ammann bars to be forced, and thus there are non-local tiles forced as well. The Deuce therefore serves as an excellent example of how optional bar forcing can help us find more forced information.

Previous research by Grünbaum and Shephard [GS87] has shown the empire of the Deuce to contain a set of Kites stretching perpendicular to the axis that separates the two Darts (in Figure 6.7, this is the 144° axis). Taking into account the complete forcing rules, however, yields a much larger empire, which can be seen by the numerous diagonal alignments of Kites, rather than just a single diagonal passing through the center of the figure. These "new" tiles were first found by Minnick [Min98], and Figure 6.1.7 shows a larger set of forced tiles found using Minnick's method.

6.1.3 The Sun

The Ammann bars required to force the Sun are shown Figures 6.10 and 6.11. The final empire of the Sun is shown in Figure 6.1.7. Just like the Ace, the Sun does not force more than one bar per musical sequence. Therefore, the Sun does not force any other tiles. Like the Ace, we include a figure of the Sun to demonstrate the software's correctness.

6.1.4 The Star

The Ammann bars required to force the Star are shown Figures 6.13 and 6.14. The final empire of the Star is shown in Figure 6.1.7. The Star forces numerous non-local tiles, because it forces two Ammann bars per musical sequence. The Star also demonstrates the five-fold symmetry present in Penrose tilings. Patterns and self-similarities begin to emerge at the scale shown in Figure 6.1.7.

6.1.5 The Jack

The Ammann bars required to force the Jack are shown Figures 6.16 and 6.17. The final empire of the Jack is shown in Figure 6.1.7. Similar to the Deuce, the Jack has Darts which force more than the initial set of Ammann bars. These extra bars are responsible for the forcing of all the non-local tiles.

This rendering of the Jack's empire shows tiles that have never been published before. The single Kites found throughout the picture are the result of forced Ammann bars interacting at long distances from the initial patch of tiles.

6.1.6 The Queen

The Ammann bars required to force the Queen are shown Figures 6.19 and 6.20. The final empire of the Queen is shown in Figure 6.1.7. The number of tiles shown in this figure are larger than those presented by Minnick, though the extra tiles are found using the same rules.

6.1.7 The King

The Ammann bars required to force the King are shown Figures 6.22 and 6.23. The final empire of the King is shown in —figfig:king-empire.

This rendering of the King's empire shows tiles that have never been published before. In addition to being much larger than the figures published by Minnick, Figure 6.1.7 shows a few small patches of Kites that Minnick did not include in her figures.

a	$sin(54°)(\frac{2\tau+1}{2\tau})$	1.059017	v	$\frac{1}{4}(3-\frac{tan(36°)}{tan(18°)})$	0.190983
b	.25	.25	x	$\frac{2\tau+1}{2\tau}$	1.3090169
c	$cos(36°)+.75$	1.559017	y	$\frac{1}{2\tau}$	0.309016 9
d	$a-1$	0.059017	z	.25	.25
e	$\tau-b$	1.368034	w	.75	.75

Figure 6.1: Minnick's vector distances relating Penrose tile vertices and Ammann bars. The distances correspond to those marked in Figure 6.1.7 and Figure 6.1.7.

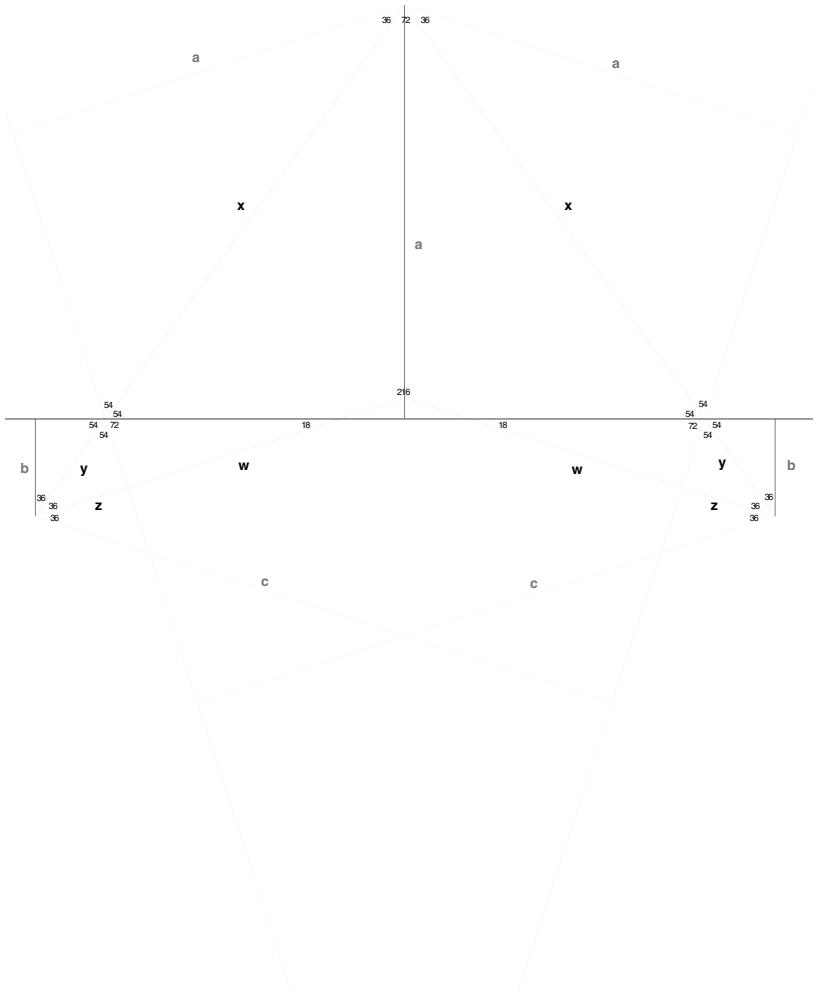


Figure 6.2: Dart tile with Minnick's distances from Figure 6.1.7 marked.

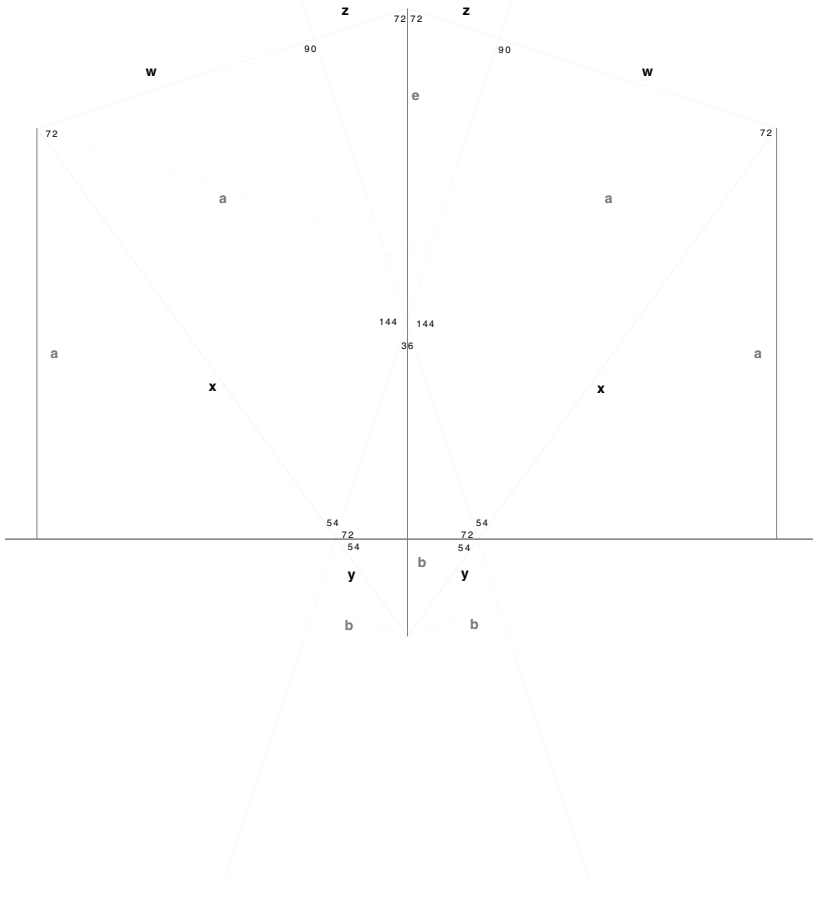


Figure 6.3: Kite tile with Minnick's distances from Figure 6.1.7 marked.

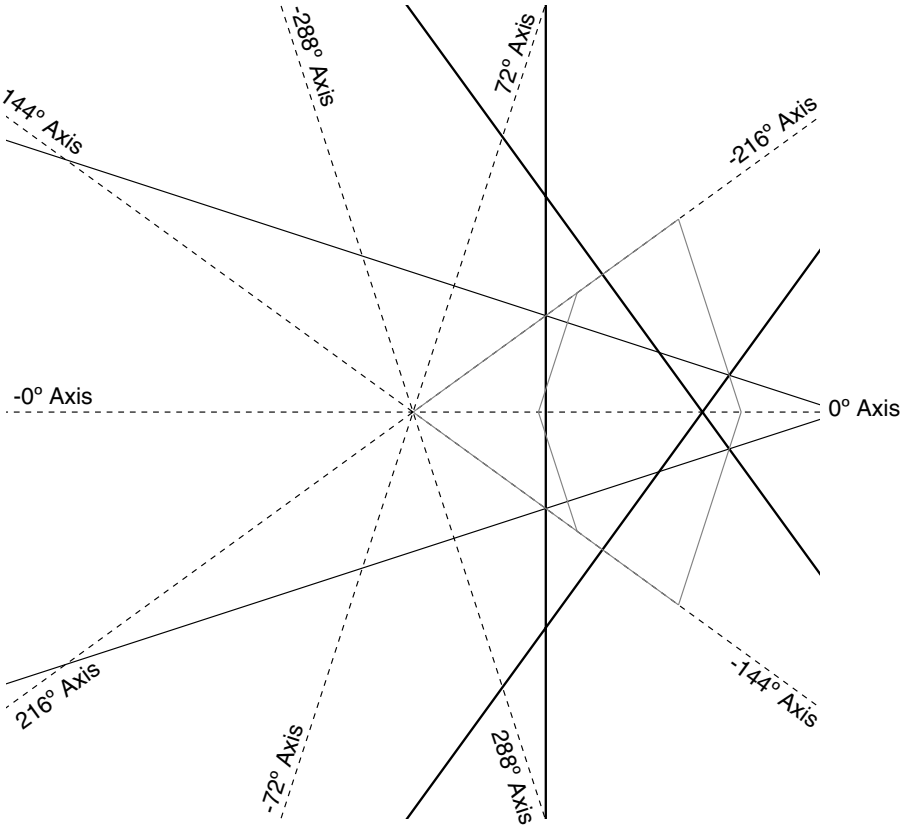


Figure 6.4: The initial constellation of the Ace configuration.

θ of Sequence:	0°	72°	144°	216°	288°
Zeroth Bar:	a	a	-(x+y+z)	-(x+y+z)	a

Figure 6.5: Ammann bar forcing distances for the Ace configuration.

Figure 6.6: The empire of the Ace configuration. No non-local tiles are forced by this configuration.

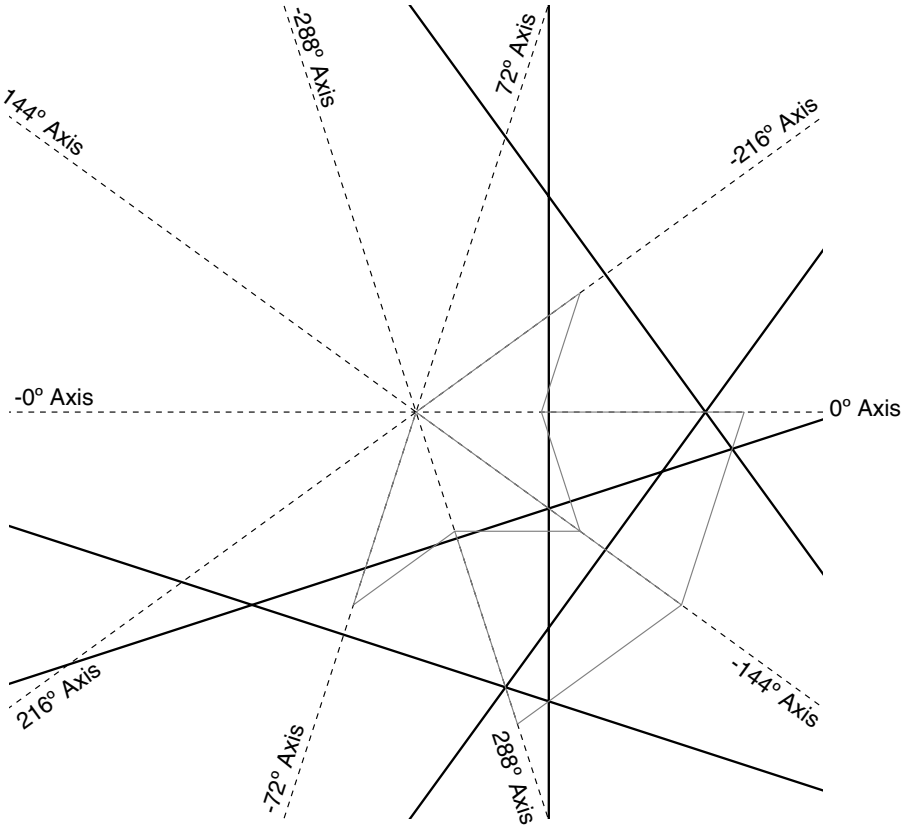


Figure 6.7: The initial constellation of the Deuce configuration.

θ of Sequence:	0°	72°	144°	216°	288°
Zeroth Bar:	a	-(x+y+z)	-(x+y+z)	-(x+y+z)	a

Figure 6.8: Ammann bar forcing distances for the Deuce configuration.

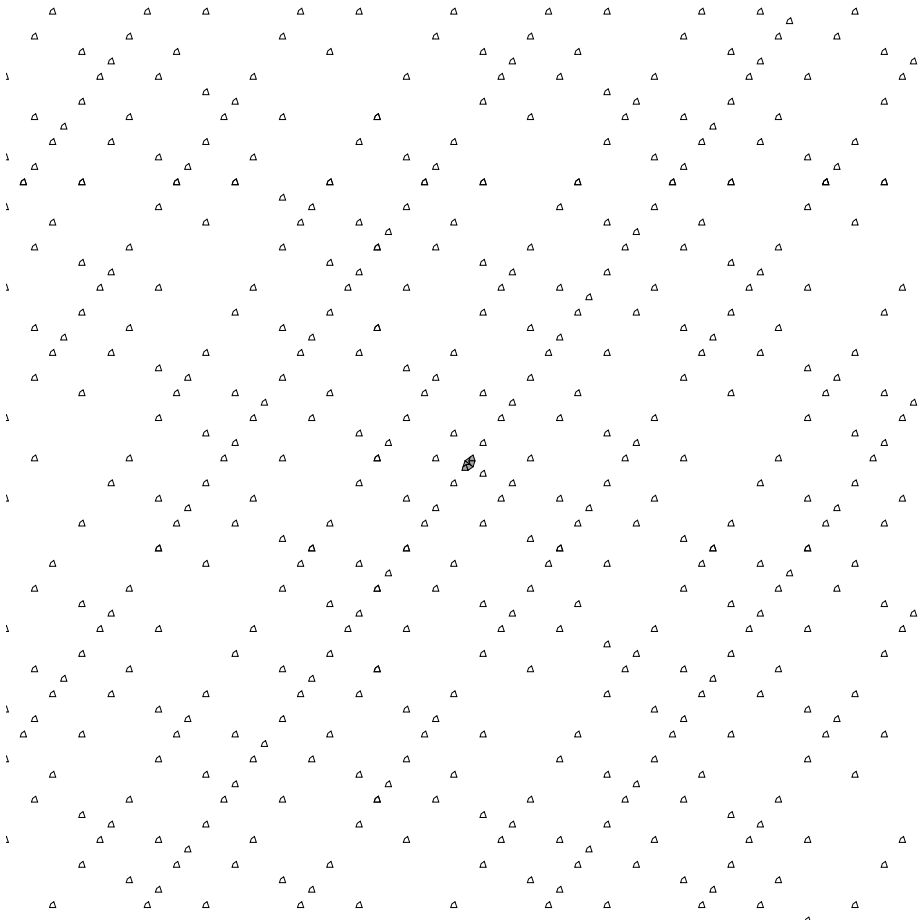


Figure 6.9: Part of the empire of the Deuce configuration.

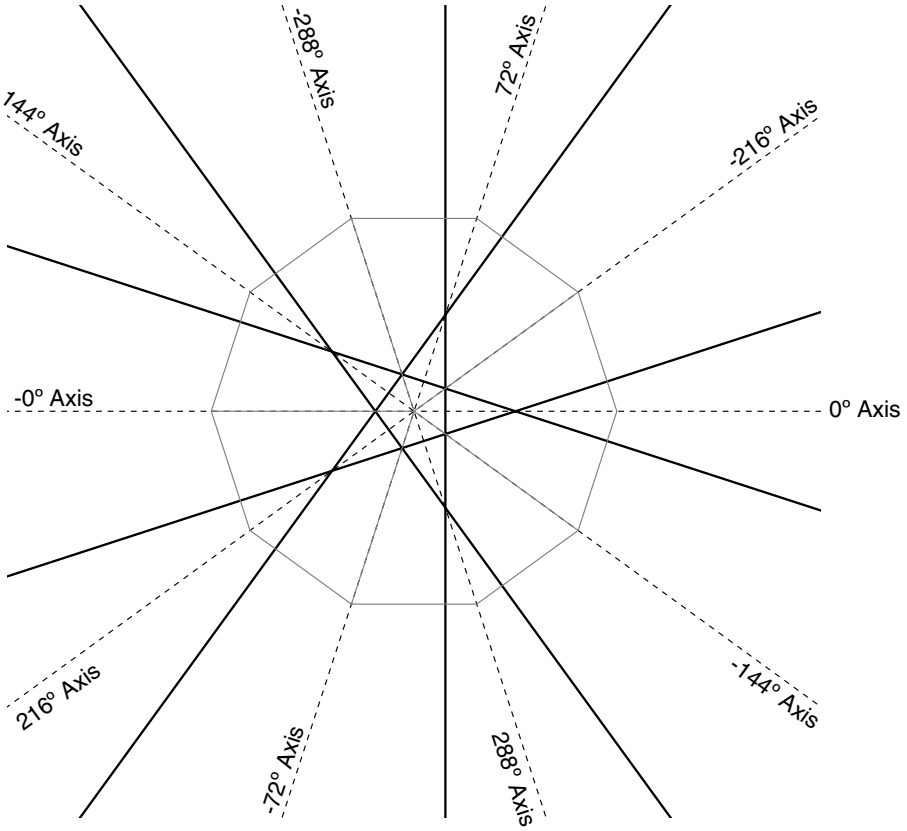


Figure 6.10: The initial constellation of the Sun configuration.

θ of Sequence:	0°	72°	144°	216°	288°
Zeroth Bar:	b	b	b	b	b

Figure 6.11: Ammann bar forcing distances for the Sun configuration.

Figure 6.12: The empire of the Sun configuration. No non-local tiles are forced by this configurations.

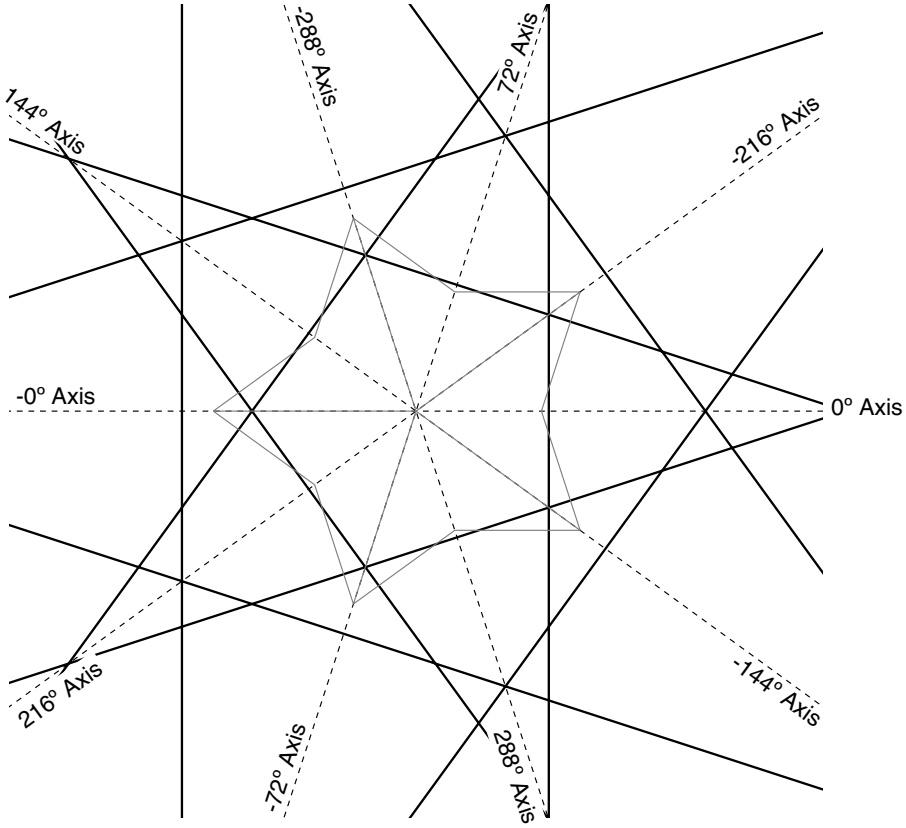


Figure 6.13: The initial constellation of the Star configuration.

θ of Sequence:	0°	72°	144°	216°	288°
Zeroth Bar:	-(x+y+z)	-(x+y+z)	-(x+y+z)	-(x+y+z)	-(x+y+z)
First Bar:	a	a	a	a	a
Interval:	**L**	**L**	**L**	**L**	**L**

Figure 6.14: Ammann bar forcing distances for the Star configuration.

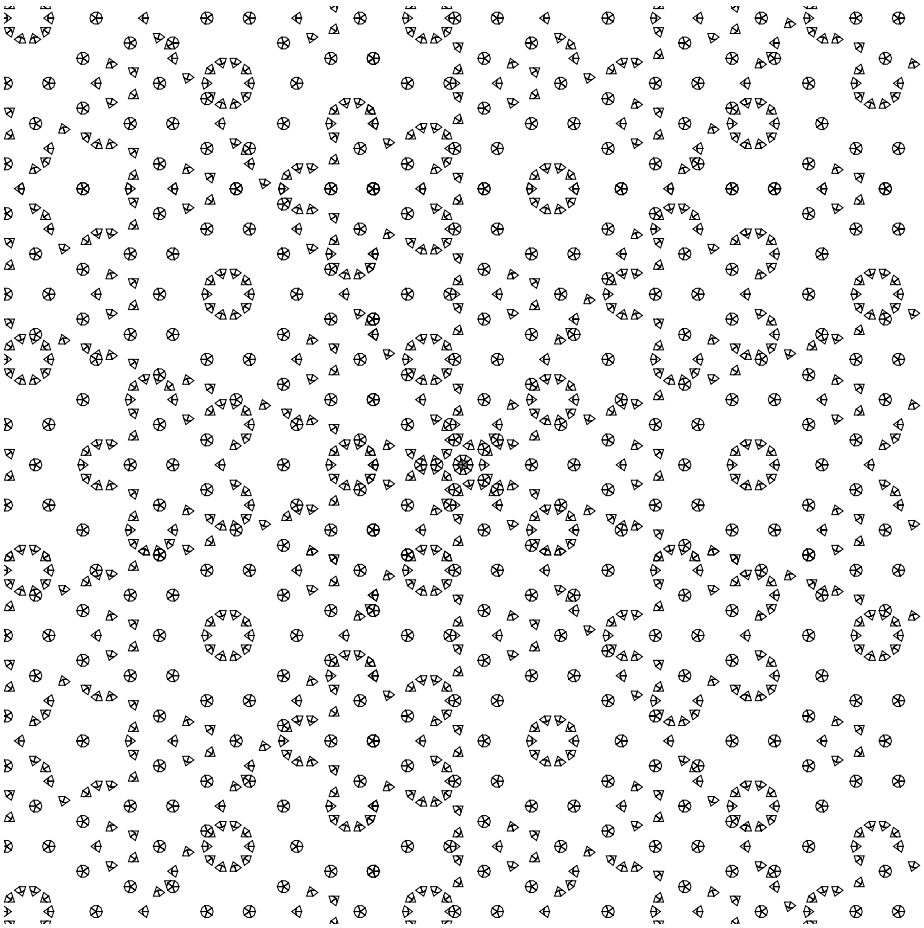


Figure 6.15: Part of the empire of the Star configuration.

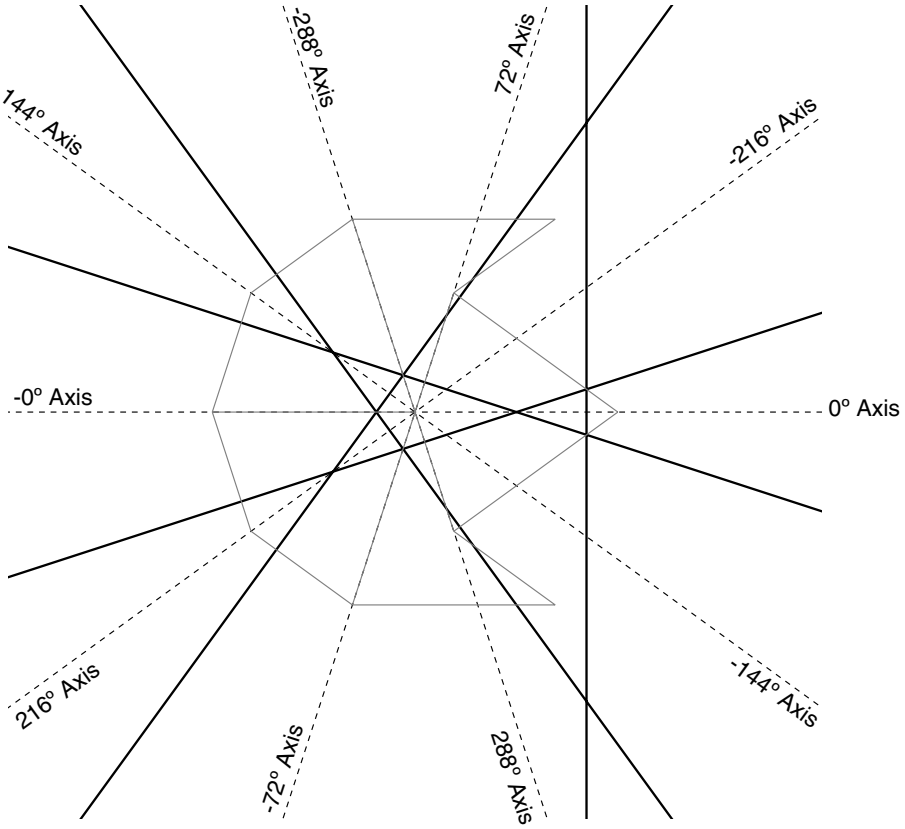


Figure 6.16: The initial constellation of the Jack configuration.

θ of Sequence:	0°	72°	144°	216°	288°
Zeroth Bar:	e	z	b	b	z

Figure 6.17: Ammann bar forcing distances for the Jack configuration.

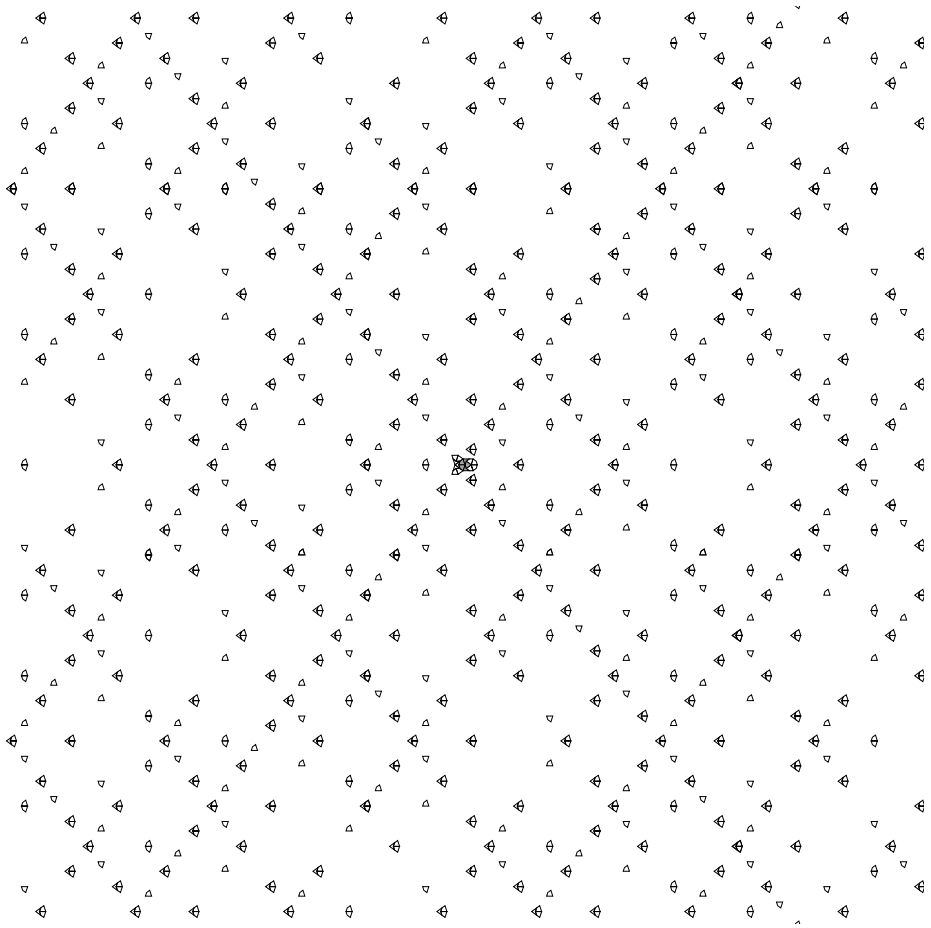


Figure 6.18: Part of the empire of the Jack configuration.

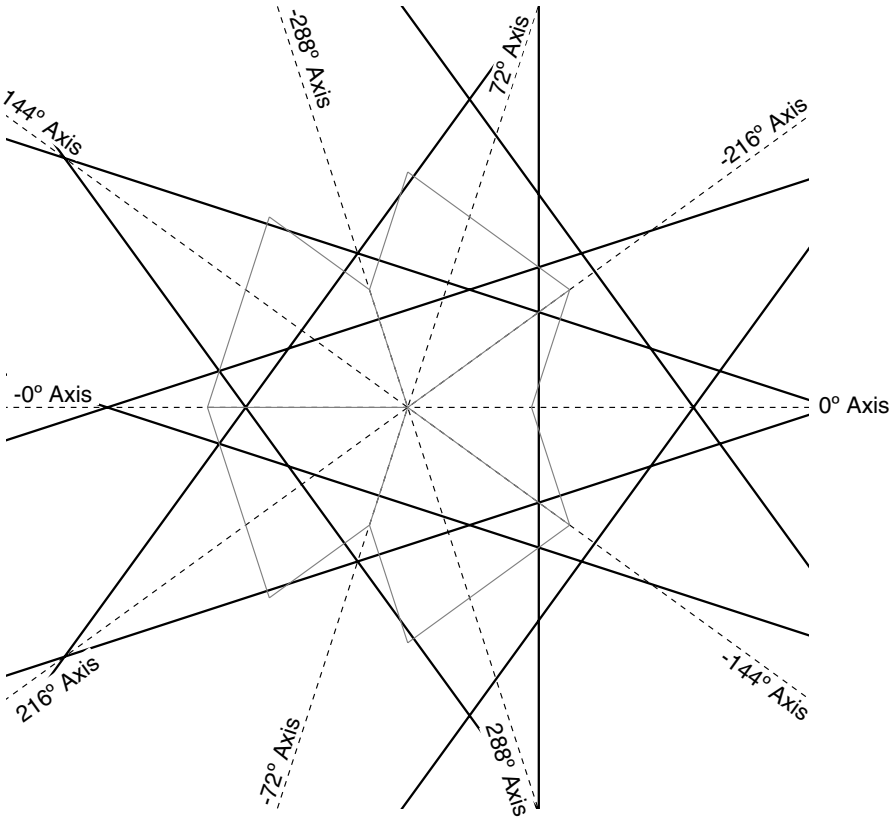


Figure 6.19: The initial constellation of the Queen configuration.

θ of Sequence:	0°	72°	144°	216°	288°
Zeroth Bar:	a	-w	-(x+y+z)	-(x+y+z)	-w
First Bar:		a	a	a	a
Interval:		**S**	**L**	**L**	**S**

Figure 6.20: Ammann bar forcing distances for the Queen configuration.

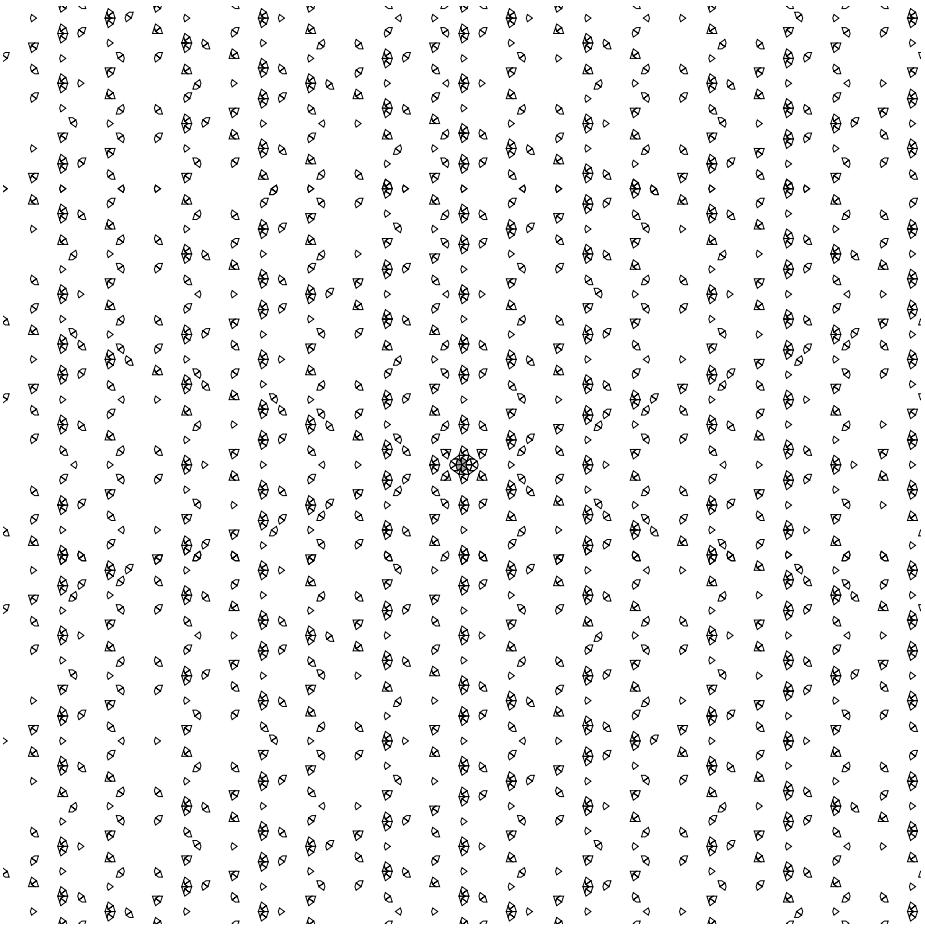


Figure 6.21: Part of the empire of the Queen configuration.

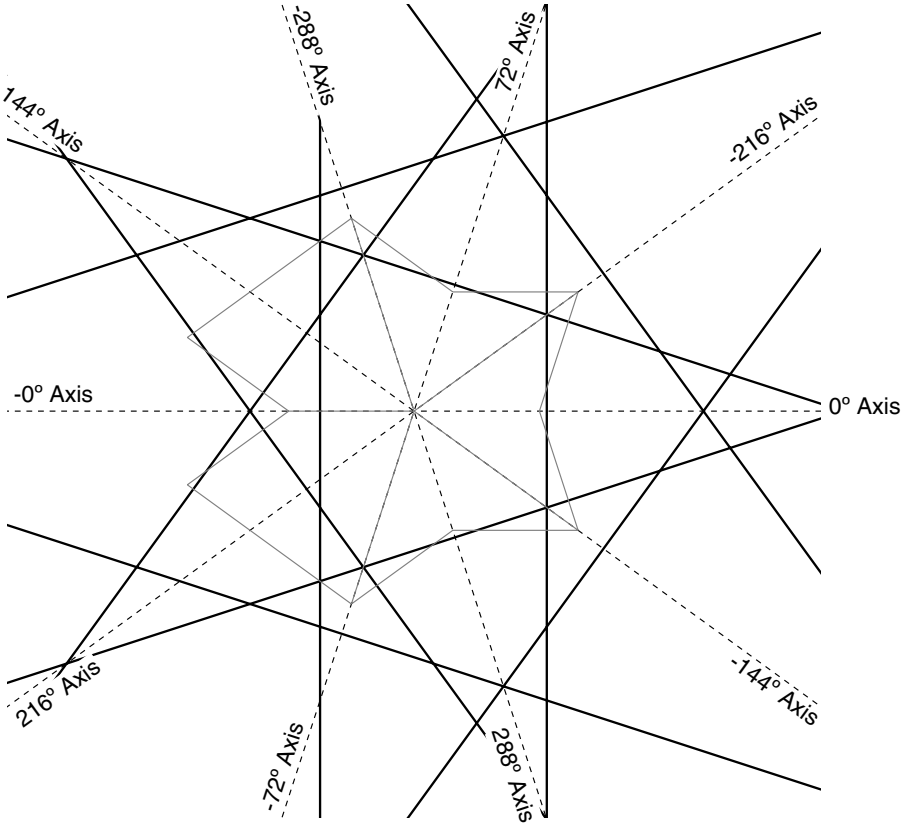


Figure 6.22: The initial constellation of the King configuration.

θ of Sequence:	0°	72°	144°	216°	288°
Zeroth Bar:	-w	-(x+y+z)	-(x+y+z)	-(x+y+z)	-(x+y+z)
First Bar:	a	a	a	a	a
Interval:	**S**	**L**	**L**	**L**	**L**

Figure 6.23: Ammann bar forcing distances for the King configuration.

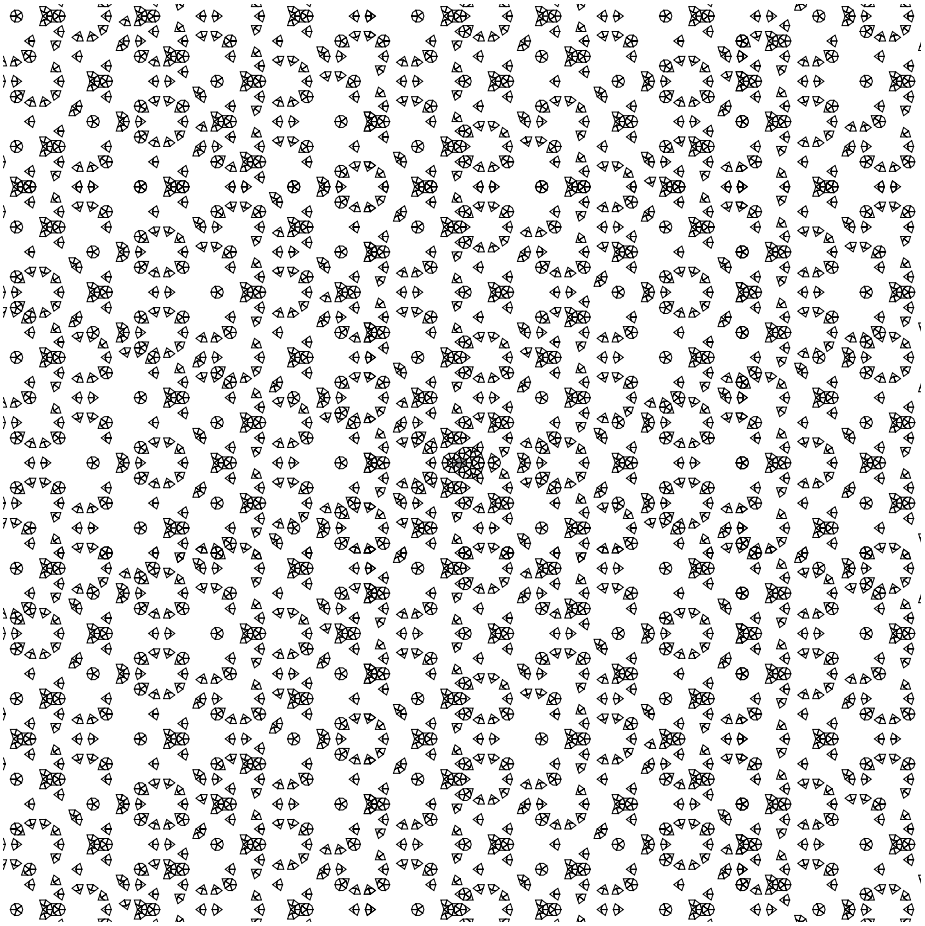


Figure 6.24: Part of the empire of the King configuration.

Chapter 7

Ongoing and Future work

We have now demonstrated the capability of the software that we have designed. Our hope is that it will allow us to find new forcing rules, similar to those outlined in Chapter 4. We have already begun some preliminary work on this, and those efforts are described below. We emphasize that what we have done so far is only the beginning; much analysis can still be done with respect to finding new rules.

Because our software makes it easy to generate empires of arbitrary patches of tiles, we can "set up" situations where we expect certain interactions to occur, and then observe the actual results in the Penrose tiling. Below are some of the ways that our software may help in the discovery of new forcing rules.

7.1 Interference

Minnick [Min98] noted that while the Ace does not force any non-local tiles, two Aces with different orientations can force non-local tiles. We decided to generate the empires of several configurations of Aces to see if there were any interesting phenomena occurring in the tilings.

We began by taking two Aces oriented at 144° from each other and placing them adjacent to each other, as shown in Figure 7.1. In this configuration, only one non-local tile is forced: a Kite that fills out the Sun formation created by the four Kites forced by the two Aces.

We then moved one of the Aces away from the other in a straight line from its initial position. We positioned this Ace at six other positions in the plane, increasing the distance of separation each time. The positions were determined by looking at the Ammann bars forced by the first Ace and calculating where another Ace could be placed so that it was not in violation of the forcing rules.

We expected to see the regions of forced tiles change as the initial tiles were moved away from each other. We dubbed the process "interference" because it was analogous to the interference patterns created by two sources of light; as the sources are moved away from each other, the pattern created by the interference

of their light changes.

The results from these interference patterns can be seen in Figures 7.1–7.1. Previous work by Minnick [Min98] had stated that only Kites would be forced by the presence of two Aces, but as some of these figures show, much larger structures are forced by just two Aces. Just as the interference pattern from light sources may grow stronger or weaker when the sources are moved, the number of tiles forced by the Aces can be seen to both increase and decrease as the initial Aces are moved. The configurations with many forced tiles are the result of fewer Ammann bars forced by the initial Aces overlapping. The more unique Ammann bars forced by the Aces, the more Ammann bars that will be forced in total, and so more tiles will be forced. Conversely, the more Ammann bars that the initial Aces have in common, the fewer Ammann bars and tiles that will be forced.

We have not been able to fully analyze these results. Our suspicion is that certain Ammann bar spacings force more information than others. In other words, forcing the first and fifth bar in a musical sequence may force more bars than forcing the first and tenth bar. This relationship is not obvious, however, and the fact that we must simultaneously consider five sets of Ammann bars only complicates matters.

Nevertheless, the results suggest that there are still many rules regarding forcings that have yet to be found. It is our hope that by relieving the researcher of the burden of computing empires we have made analysis of Penrose tilings easier and more productive.

Figure 7.1: Two-Ace interference: Separation 1

Figure 7.2: Two-Ace interference: Separation 2

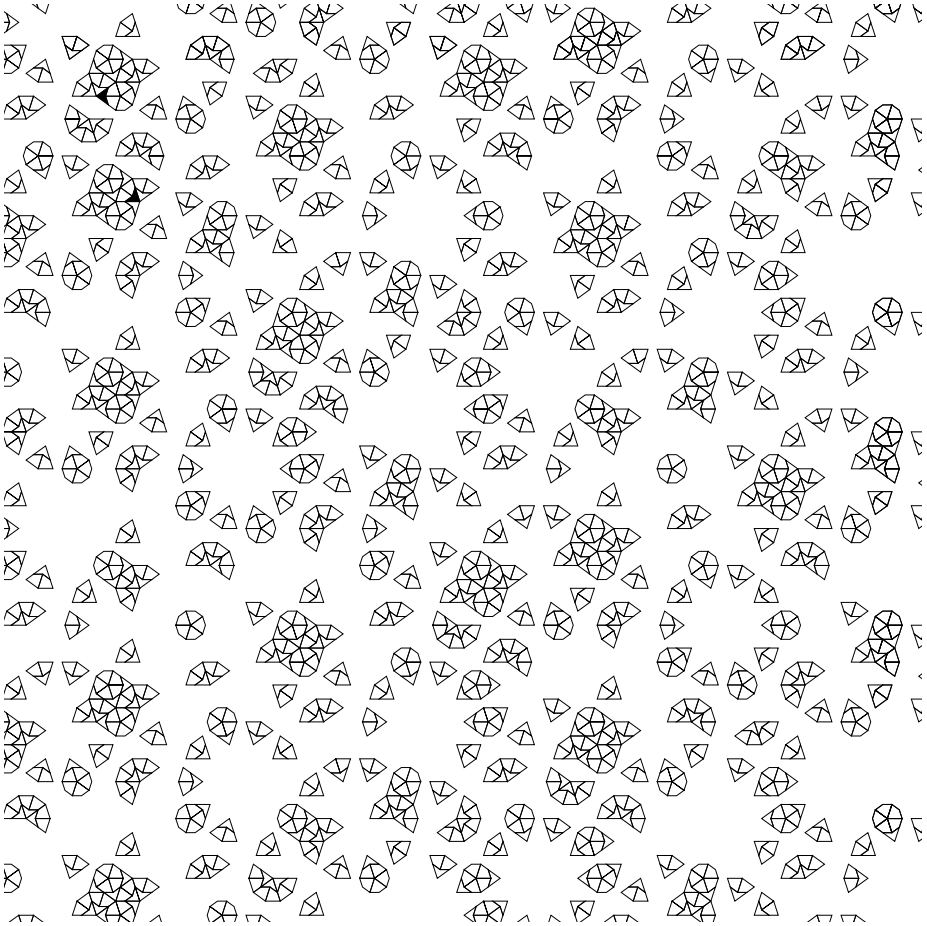


Figure 7.3: Two-Ace interference: Separation 3

Figure 7.4: Two-Ace interference: Separation 4

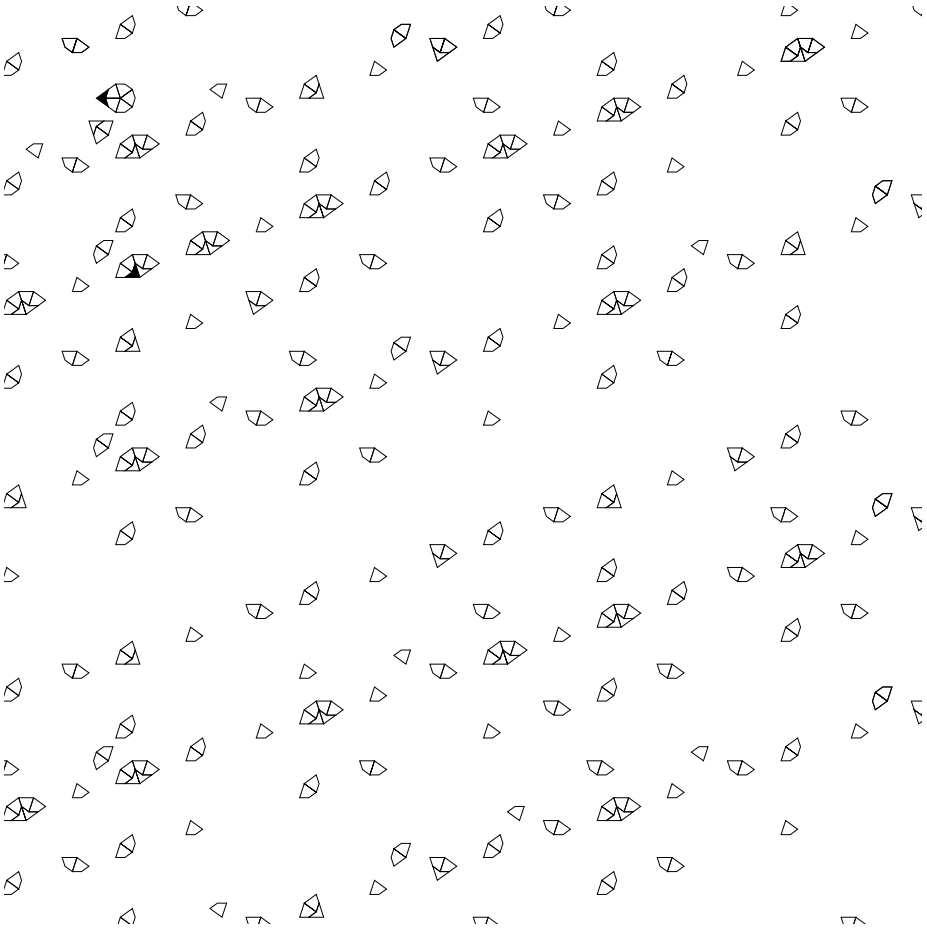


Figure 7.5: Two-Ace interference: Separation 5

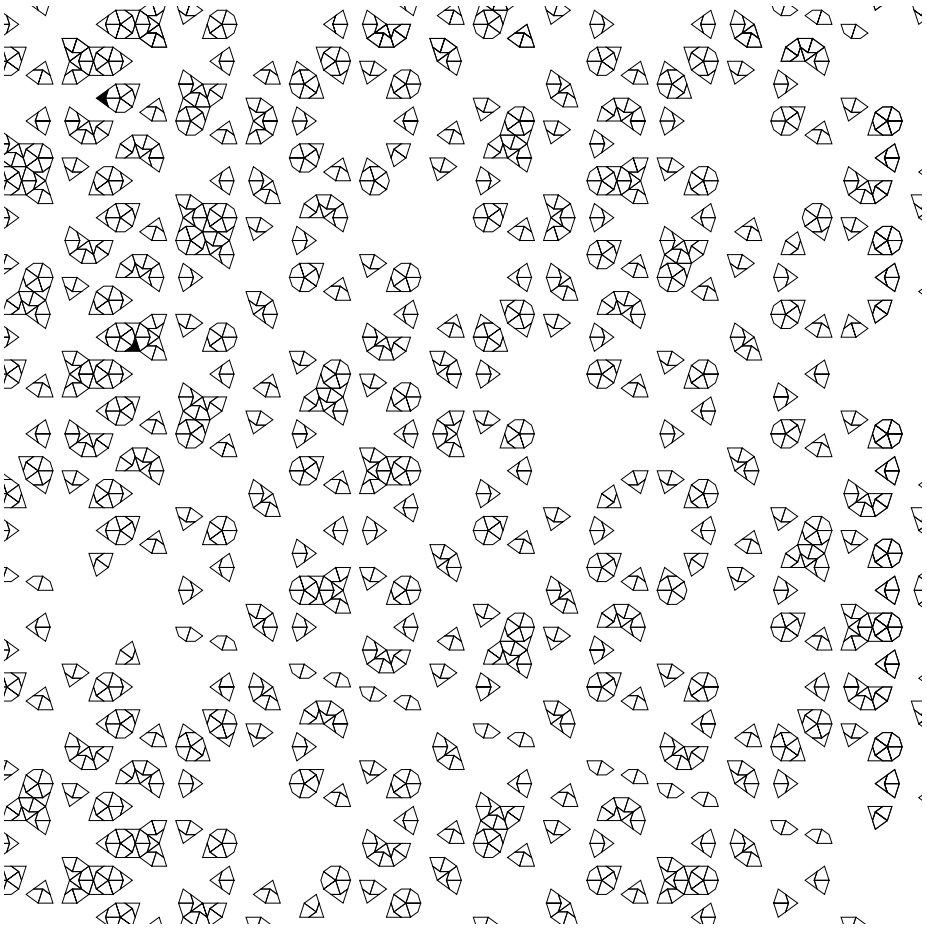


Figure 7.6: Two-Ace interference: Separation 6

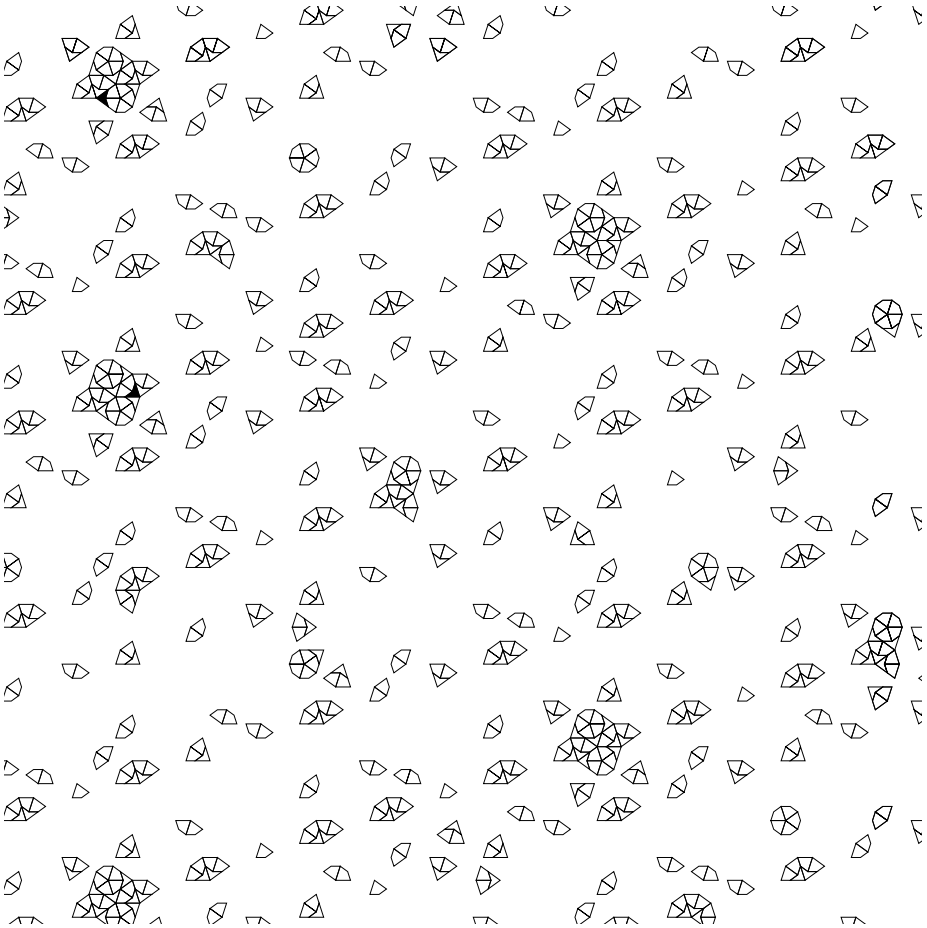


Figure 7.7: Two-Ace interference: Separation 7

7.2 Three-Dimensional work

Another possible area of research would be in the area of three-dimensional Penrose tilings (Danzer tilings). Penrose tilings in two dimensions are fairly easy to understand, which may help to explain why no one before us had gone to the trouble to write software to generate empires. The three-dimensional analogues to Penrose tilings, however, are extremely difficult to visualize and model. Because the software takes care of the part of the problem that computers are good at (calculations), but leaves the part that humans are better at (pattern recognition and rule creation), it could be extremely helpful in the three-dimensional case.

7.3 Iterative Verification

A final possibility for further work would be using the software in combination with iterative methods for generating Penrose tilings. Socolar [Soc91] developed algorithms to iteratively grow defect-free tilings. These algorithms exploit only local vertex-alignment rules, and so cannot take into account the long-range forcings that we have found. Used in conjunction with our software, however, they could help us find new forcing rules.

Socolar's method "grows" a tiling in the following way: An initial patch of tiles is placed in the plane. Any tiles that are locally forced by this patch are added until there are no more vertices where the tile choice is forced. At this point, an arbitrary tile is added to a certain part of the tiling (the exact placement is not important to this discussion), which creates a new area where forced tiles must be placed. In this way, the tiling grows in spurts as forced tiles are placed, a surface is reached where no forced tiles can be placed, and then one tile changes the surface such that more forcings can occur.

Our software attempts to find all forced tiles using non-local rules. Perhaps our methods could be combined with socolar's to find more forced tiles. Suppose we began with an initial set of tiles. Our software would find all the Ammann bars forced by this initial patch. Then, using Socolar's algorithm, we could begin to place tiles in the plane, but *only if the placed tiles were consistent with the Ammann bars.* socolar's method does not have any long-range structure, and so the arbitrary selection of tiles in his algorithms would cause problems with our empire finding. If we ensure that the tiles found through Socolar's method remain consistent with the Ammann bars that have already been found, then we can "grow" the empire of an initial set of tiles and know that it is consistent with our original patch. This prevents us from having to perform the time-consuming pattern matching to find forced tiles, and it may find tiles that are not found using our software alone.

Chapter 8

Summary and Conclusion

This work is useful because it sets the stage for future research in the area of Penrose tilings. We have implemented a constant-time algorithm that forces arbitrary bars in musical sequences. This required refining Minnick's algorithm so that it was suitable for use in a computer with finite precision; we had to keep the algorithm's arbitrarily precision, but still use only a finite number of resources.

We developed algorithms that would enable us to determine the interactions between several sets of Ammann bars in the plane. This required devising a coordinate system that would retain the precision of specific bar numbers. At the same time, we needed to allow for a conversion to two-dimensional coordinates so that Euclidean distances could be determined. Without these two-dimensional equivalents, pattern matching and tile drawing would not be possible.

Using the intersections gathered from the plane, we developed algorithms that would search through the intersection points, looking for patterns that interested us. While the algorithms are inherently of $O(n^2)$ complexity, we introduced a "scan line" algorithm which reduced the expected runtime to $O(n\sqrt{n})$. The algorithms are general, and can be easily modified to search for new patterns.

The patterns were not merely matched; they also provided a method to incorporate new information into the Penrose tiling. This feature makes it easy to integrate new rules for forcings in Penrose tilings. When a new rule is discovered, a pattern is created that matches the initial conditions for the rule, and then the postconditions for the rule integrate the new information into the tiling.

As part of the testing and validation of these algorithms, we had to generate the patterns for the basic Penrose tiles and rule forcings. Thus, we created patterns for the Kite (which forces no new information), the Dart (which forces two adjacent Kites), and the "Double Kite" (which indirectly forces an Ammann bar).

Additionally, to verify our work against previous work, we computed the initial Ammann bar sets which force the seven vertex configurations: Ace, Deuce,

Star, Sun, Jack, Queen, and King. The definition of these tiles in terms of pure Ammann bar addresses has never been done before. Because the Ammann bars are such a compact representation of the tilings, defining tiles in terms of the bars makes more sense than using the tiles themselves.

Beginning to look for new forcings, we computed the empires of two "interfering" Aces, spaced at different intervals from each other. In addition to these interference patterns, we suggested other possible avenues of further study. Our methods may be applicable to the three-dimensional version of Penrose tilings, advancing study in that area. Also, an iterative empire-generating method using a combination of our non-local methods and Socolar's local methods may yield rules that could not be found using either method alone.

It is our hope that the software and methods outlined in this work will assist future research by providing a tool that frees the researcher from the hard work of generating Penrose tilings. With such a tool, more can be learned about Penrose tilings.

Colophon

This document was written in the LaTeX document processing system, using GNU Emacs as the editor. The figures in the document were created with the software created for this research and ClarisWorks 4.0 (on the Macintosh). All figures were cropped and made ready for press in Adobe Illustrator 8.0 (again, on the Macintosh) before being exported to EPS format for inclusion in the LaTeX document.

The software itself ("the tiling package") was written in Java, version 1.1.8. It makes use of Duane Bailey's Java Structures packages for certain data structures (mainly the GUI and large structures such as binary trees and hash tables). All the other code is pure Java, consisting of over 6,000 lines of original code spread out over 18 classes.

To "date" this work: the figures produced by the software were rendered on a Sun Ultra workstation, and usually took between 10 minutes and 3 hours to complete, depending on the complexity of the picture. The average personal computer today has a 750MHz processor, 128MB of RAM and a 10GB disk drive for secondary storage. Such a device would cost $1,500. By comparison, a year of tuition at Williams College costs $30,000, and a Big Mac costs $2.79.

Bibliography

[CL90] J. H. Conway and J. C. Lagarias. Tiling with polyominoes and combinatorial group theory. *Journal of Combinatorial Theory, Series A*, 53:183–208, 1990. 4

[Dan89] L. Danzer. Three-dimensional analogs of the planar penrose tilings and quasicrystals. *Discrete Mathematics*, 76:1–7, 1989. 3

[Dis59] W. Disney. Donald in mathmagic land. Walt Disney Home Video Release, 1959. 9

[GS87] Branko Grünbaum and G. C. Shephard. *Tilings and Patterns*, chapter 10. W. H. Freeman and Company, New York, 1987. 5, 9, 13, 15, 47, 57

[Min98] Linden Minnick. Generalized forcing in aperiodic tilings. Technical report, Williams College Department of Computer Science, 1998. 8, 10, 21, 57, 76, 77

[Pen74] Roger Penrose. The rôle of aesthetics in pure and applied mathematical research. *Bulletin of the Institute of Mathematics and Its Applications*, 10:266–271, 1974. 3, 4

[Soc91] Joshua E.S. Socolar. Growth rules for quasicrystals. In D. P. DiVincenzo and P. J. Steinhardt, editors, *Quasicrystals: The State of the Art*, volume 11 of *Directions in Condensed Matter Physics*. World Scientific, Singapore, 1991. 5, 85