The Eternity Puzzle: A Linear Algebraic Approach

John Burkardt, Marcus Garvie

July 11, 2021

Abstract

The Eternity puzzle is an unusually complex geometric tiling task. Several solutions have been discovered, which carry out a brute force search on a reduced puzzle after some clever preprocessing. However, like many tiling problems, the Eternity puzzle lends itself to a linear algebraic analysis, in which the determination of solutions can be carried out by an automatic analysis of an underdetermined linear system $Ax = b$, in which the solution x is only allowed binary values. Powerful linear programming packages are available to solve such systems exactly. In this report, we describe in detail the methods for constructing the linear system. We demonstrate the feasibility and complexity of the solution procedure on a set of smaller problems. The approach can be extended to the Eternity puzzle, to produce all possible solutions, of which there are at least two. The details of actually setting up and solving that puzzle are deferred to a later report.

1 Introduction

In 1999, Christopher Monckton introduced "Eternity: the puzzle worth a million", for which a £1,000,000 reward was offered for the first correct solution. A set of 209 polygonal tiles were provided, which needed to be configured to exactly cover an irregular dodecagonal region. Assuming a brute force search is applied, the cost of discovering a solution justifies the puzzle's "Eternity" name. But to the surprise of the proposer, two distinct solutions were reported in 2000, the first by Alex Selby and Oliver Riordan, the second by Guenter Stertenbrink, neither of which seems to correspond to the Monckton's undisclosed solution, for which a few tile placements were provided as initial hints. The approach of Selby and Riordan began with an insight that the puzzle probably had many distinct solutions. This led them to preprocess the puzzle to a point where a brute force search could be effective.

The Eternity puzzle is a member of a large class of exact cover problems, consisting of a fixed *grid*, composed of a number of many adjacent polygonal elements, and a number of *tiles*, each composed of a smaller number of elements. A solution of the puzzle is rule for laying the tiles down onto the grid in such a way that each grid element is covered exactly

once by a tile element, each tile is used exactly once and each tile element is matched to exactly one grid element.

Selby and Riordan used a two-phase algorithm, which first set up a "favorable state" of a certain size in which the most awkward pieces are positioned and the remaining uncovered region has a favorable shape, followed by an exhaustive search of all possible placements of the remaining "easy-to-place" tiles. By greatly reducing the number of tiles to be handled in the brute force portion of the procedure, they brought the cost of the search down to a feasible range.

It is our purpose to consider puzzles of the Eternity type, and reformulate them as an underdetermined system of linear algebraic equations, for which there are efficient, reliable methods that, in theory, can produce all solutions automatically.

Each equation of this linear system specifies that a particular element of the grid is covered, or that each tile is used exactly once. Each variable of the linear system corresponds to a distinct configuration of a tile, that is, a combination of rotation, reflection, and translation that aligns the tile with the grid and does not extend outside of it. As a side condition, we must also require that the solution vector is binary, where a value of 1 or 0 indicates that a particular configuration is used or not used in the tiling.

2 The Geometry of the Eternity Grid

The grid for the Eternity puzzle appears as Figure 1. On a first glance, a regular pattern of equilateral triangles appears. A closer look reveals an underlying finer grid of 30-60-90 triangles, which make geometric calculations more complicated.

The Eternity puzzle has several notable characteristics:

- The grid uses a *fine grid* of 2,508 elements, which are 30-60-90 triangles; these elements are sometimes called *drafters*, since they share the shape of the drafter's triangle;
- The grid includes a *medium grid* of equilateral triangles each formed from 6 of the elements; however, some triangles are only partially formed because they intersect the boundary of the grid;
- The grid includes a coarse *hex grid* of hexagons, each formed from 12 of the elements. If we wish to locate any specific element in an Eternity-type grid, we can do so by specifying a particular hexagon, and then the rotation degrees, or clock-position, of the element. This idea offers us a clue into how to construct a coordinate system.
- The grid reveals small rectangular structures, which we will call *cells*, involving 12 elements. These cells occur in pairs, with one the upside-down version of the other. When analyzing tiling problems, it will be to our advantage to imagine a rectangular background grid, starting with a single cell, extended horizontally and vertically by

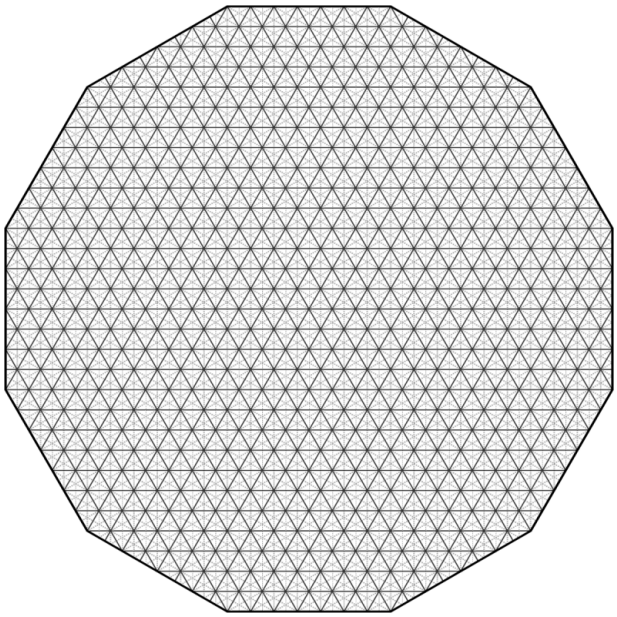


Figure 1: The Eternity Grid: The fine grid of 30-60-90 triangles is outlined in light gray. The medium grid of equilateral triangles is in darker gray. Many hexagons are easily visible, but they overlap. The coarse grid is constructed by selecting staggered rows of them. It is clear that some triangles and hexagons along the border are only partially formed.

alternating the cell and its upside down version. Conway [1] terms this the *kisrhombille tiling*. Having such a rectangular grid allows us to devise a fairly regular numbering scheme for its nodes and elements. Labels, adjacency data, and other information from this regular structure can then be transferred to a general, less regular, puzzle grid.

- The Eternity grid is in the shape of a slightly irregular dodecagon, a 12-sided polygonal figure. Assuming that the equilateral triangles in the medium grid have a side length of 1 unit, the irregularity of the Eternity grid arises because its 12 boundary sides alternate in length between 7 units and $8\frac{\sqrt{3}}{2} \approx 6.9292$ units,

- There are 209 tiles; each tile is formed from 36 elements; each tile is unique; no tile has any symmetry; a tile that is flipped over has a distinct shape. Because they are formed from drafters, the tiles are sometimes called *polydrafters*.

From the assumption that the equilateral triangles in the grid have a side length of 1 unit, we can deduce measurements for other features of the grid:

- Each 30-60-90 element has sides of length $\frac{\sqrt{3}}{6}$, $\frac{1}{2}$, and $\frac{\sqrt{3}}{3}$, opposite the angles of 30°, 60°, and 90° respectively

- Each equilateral triangle of 6 elements has sides of length 1, and vertical height $\frac{\sqrt{3}}{2}$.

- Each rectangular cell of 12 elements has horizontal width 1 (one equilateral triangle width) and vertical height of $\frac{\sqrt{3}}{2}$ (one equilateral triangle height).

- Each hexagon of 12 elements has horizontal width 1 and vertical height $\frac{2\sqrt{3}}{3}$.

3 Characteristics of the 30-60-90 Triangles

Many geometric tiling puzzles rely on an underlying grid of identical elements, which are typically squares, triangles, or hexagons, as in puzzles involving polyominoes or polyhexes. In the Eternity puzzle, the fundamental element is, unusually, a 30-60-90 triangle. Geometric questions about the Eternity puzzle must take account of the properties of these elements.

- Each element has a *position*. Once we have chosen a coordinate system for the grid, we can specify the position of any element by, for example, giving the coordinates of its vertices. We will always order the vertices to appear in the same 30-60-90 ordering as their corresponding angles; hence an element's position with report the coordinates of the vertices in this order.

- Each element has a *rotational angle*. Consider in Figure 2 the 12 elements in the center that form a hexagon. The element bounaries lie along rays at $0°, 30°, ..., 330°$. As a shorthand for the rotational angle, we assign a *type* to each element. The element between the 3 o'clock and 2 o'clock positions has a type of 1, and by applying counterclockwise multiples of a 60° rotation to this element, we can generate 5 more elements, which we assign to types "2" through "6". Similarly, if we label "-1" the element between the 3 o'clock and 4 o'clock positions, we can use 60° rotation in a clockwise sense to generate the remaining elements, which we will label "-2" through "-6". The type of an element will become important when we seek to carry out the tiling process.

- Each element has a *parity*; the parity is essentially the sign of the label. For any element with a positive label, traversing the nodes in the order $30°, 60°, 90°$, results in a counterclockwise motion, corresponding to a positive sense. Correspondingly, elements with a negative label are traversed in a clockwise motion, and have a negative sense. During a physical tiling process, we may need to turn a tile over in order to get a match. This is equivalent to reversing the parity of the tile, and corresponds to a reflection operation.

It may help, at this point, to imagine the physical process of tiling a grid such as the Eternity puzzle. We do so by selecting the next tile to be placed. Holding the tile in hand, we see that we may need to turn the tile some multiple of 60°. Then we may need to flip the tile over. Finally, we need to move the tile to the appropriate location on the grid. Correspondingly then, a computational solution to the tiling problem requires specifying the rotation, reflection, and translation applied to each tile from some starting default position.

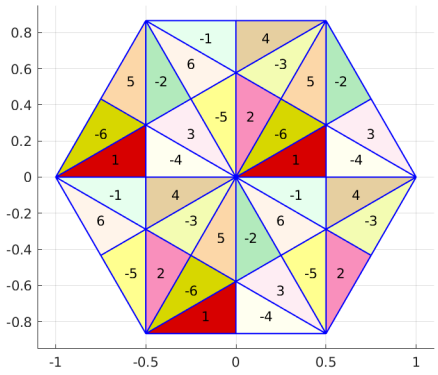


Figure 2: This grid contains a central hexagon of 12 elements, which includes all 12 element types, distinguished by label and color, which are also displayed on similar elements in the grid.

In other words, a tiling solution can be thought of as an ordered list of the transformations applied to each tile.

4 Trinity: a Smaller Version of the Eternity Puzzle

To discuss a solution technique for the Eternity puzzle, it is helpful to delay the consideration of the sheer size of the Eternity grid and the number of tiles employed. We can start with a much smaller analog of the puzzle, in order to focus on algorithmic details. Alex Selby posted online the notes from one of his presentations about his solution techniques [8], in which he displayed a simple Eternity-type puzzle involving just four tiles. He did not name this puzzle, and so we have taken the liberty of referring to it as the "Trinity puzzle", since the name vaguely rhymes with "eternity", and because Alex Selby comes from Cambridge University, which has a Trinity College, and because we might see the shape of a mythical Trinity cathedral suggested by the grid, as in Figure 3.

The Trinity grid is to be covered using the four tiles shown in Figure 4. Each tile must be appropriately rotated, reflected, and translated in order to specify its place in the tiling of this grid. The tiles exhibit some features worth mentioning, because they will also be characteristic of the Eternity tiles:

- Each tile is formed from exactly 36 of the 30-60-90 elements;
- Each tile can be formed by 12 halves of equilateral triangles, joined at edges of the same length; This fact somewhat limits the possibility of any tile having a bizarre shape.
- Notice, moreover, that any shape formed by combining three contiguous elements within an equilateral triangle is also a 30-60-90 triangle. This fact means that the

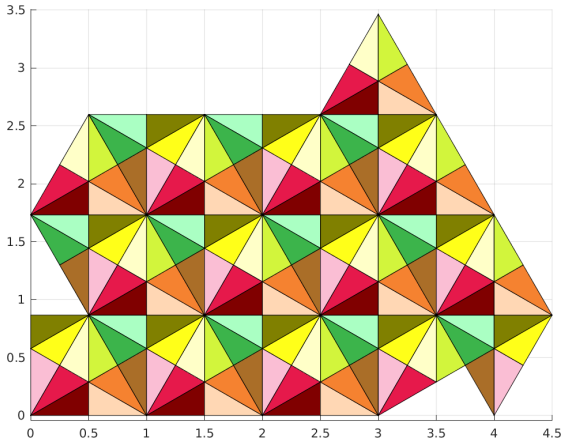
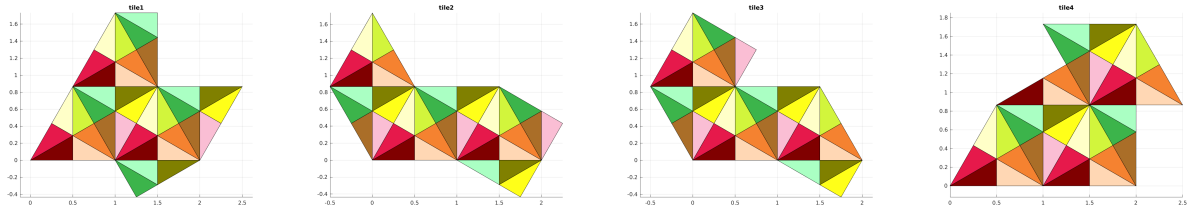


Figure 3: The Trinity grid comprises 144 copies of the basic 30-60-90 elements Colors distinguish the 12 possible orientations of the elements.



(a) (b) (c) (d)

Figure 4: The Trinity Tiles. There are 4 tiles, each involving 36 of the elemental 30-60-90 triangle. Element colors are for illustration, and are not part of the tiling process.

simple terms *"30-60-90 triangle"*, and even *"polydrafter"* can be ambiguous. Thus we prefer to say *element* when referring to the smallest triangles in the grid, and *half-triangles* to the shapes formed by three contiguous elements of an equilateral triangle.

- Each tile can be assigned a binary parity. If we use a red/black alternating color for the triangular elements of a tile, then the parity P is simply the difference between the number of red and black elements, $P = R - B$. An Eternity tile is always formed by combinations of halves and wholes of equilateral triangles. Since whole triangles have a parity of zero, the parity of the tile is the sum of the parities of any half triangles.

- It is possible to consider a higher order parity of the tiles. There are 12 possible orientations of an element, and these can be grouped way into six pairs of positive and negative orientations. This allows a six-fold parity vector to be associated with any specified orientation of a tile.

A solution of the Trinity puzzle is displayed in Figure 5. Computationally, to get from the specification of the grid and the tiles to a solution requires determining algorithmically the

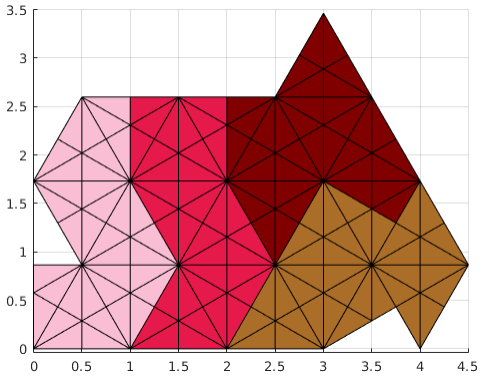


Figure 5: How 4 tiles cover the Trinity region.

same rotations, reflections, and translations that a puzzle solver would discover "by hand". Our approach will use linear algebra to represent such a puzzle, to determine whether the tiling problem has no, one, or many solutions, and to exhibit solutions that are discovered.

5 The Boundary Word of an Eternity Object

In order to describe the shapes of the puzzle grid and tiles, we need an efficient, flexible computational representation, from which we can determine structure, and orientation, and which allows us easily to apply the transformations corresponding to tile manipulation.

The grids and tiles can be referred to in general as *Eternity object*. Such an object is a shape formed by connecting a finite number of uniform 30-60-90 triangles edge-to-edge. A *grid* is an Eternity object that is to be tiled, and a *tile* is an Eternity object that is part of a set used to cover a grid. Although it is perfectly possible to create Eternity grids that have internal holes, we will assume for our discussion that all grids and tiles are simply connected, that is, that they do not involve internal holes.

Under this "no-holes" assumption, it turns out that any Eternity object can be represented by its *boundary word* (also called a Freeman chain code) [2], [3], a standard technique from combinatorial geometry that describes a polygonal shape by specifying a path around its boundary. To construct this representation, we need to know the possible path directions and step lengths for our Eternity objects.

The bounding edges of an Eternity object will always be edges of a basic triangular element. These edges can have only 3 possible lengths, which we designate as **A**, **B** and **C**, corresponding to the sides which are opposite the element angles of $\alpha = 30°$, $\beta = 60°$, and $\gamma = 90°$, respectively.

To build the boundary word, we need to agree on a coordinate system, within which we fix an Eternity object in some standard reference position. This guarantees that the object conforms in size and orientation with the underlying existing grid. Assuming that the object

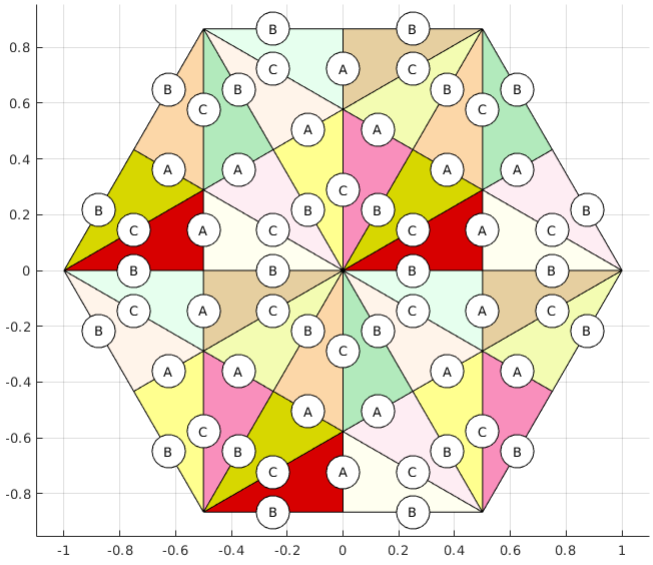


Figure 6: Edges of length B occur at angles of $s * 60°$, while A and C lengths are at angles of $30 + s * 60°$.

has been aligned with the Eternity grid, there are 12 directions in which an edge can be drawn. In the six directions which are a multiple of 60°, the edge will always be of length **B**. For the six intermediate directions, edges can either be of length **A** or **C**. Figure 6 suggests the correspondence between edge direction and edge lengths.

Now suppose that we want to specify the position and shape of an Eternity object. We designate some boundary vertex as our starting point, recording its coordinates as **P**. We then trace, in a counterclockwise direction, the boundary of the object until we return to **P**. We record our travel by listing the directions and stepsizes taken as we follow the element edges that constitute the boundary of the object, If we use a set of symbols to represent the particular directions and stepsizes, then this string of symbols constitutes the boundary word of the object, which we might refer to as **W**. Together, the values of **P** and **W** provide a compact representation of the position and shape of our objects.

Table 1 lists the symbols used for Eternity object boundary words. Each alphabetic character specifies a direction. In cases where two stepsizes are possible, a lower-case version of the letter is used to indicate that a half-step is taken. A sort of boundary word "compass" is displayed in Figure 7, which graphically suggests the direction and length of the steps corresponding to each symbol.

As an example, let us consider the Trinity grid, whose boundary word is **AAAAAAB-bKCCEEEEEEIIGGGGIIKKGjJ**. Figure 8 shows how each segment of the boundary

Letter	Angle	Stepsize	Step	Half step
A	0°	1/2	$(1/2, 0)$	
B; b	30°	$\sqrt{3}/3$	$(1/2, \sqrt{3}/6)$	$(1/4, \sqrt{3}/12)$
C	60°	1/2	$(1/4, \sqrt{3}/4)$	
D; d	90°	$\sqrt{3}/3$	$(0, \sqrt{3}/3)$	$(0, \sqrt{3}/6)$
E	120°	1/2	$(-1/4, \sqrt{3}/4)$	
F; f	150°	$\sqrt{3}/3$	$(-1/2, \sqrt{3}/6)$	$(-1/4, \sqrt{3}/12)$
G	180°	1/2	$(-1/2, 0)$	
H; h	210°	$\sqrt{3}/3$	$(-1/2, -\sqrt{3}/6)$	$(-1/4, -\sqrt{3}/12)$
I	240°	1/2	$(-1/4, -\sqrt{3}/4)$	
J; j	270°	$\sqrt{3}/3$	$(0, -\sqrt{3}/3)$	$(0, -\sqrt{3}/6)$
K	300°	1/2	$(1/4, -\sqrt{3}/4)$	
L; l	330°	$\sqrt{3}/3$	$(1/2, -\sqrt{3}/6)$	$(1/4, -\sqrt{3}/12)$

Table 1: Boundary word table for the (x,y) coordinate system.

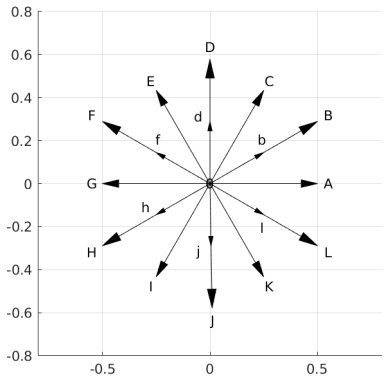


Figure 7: The Boundary Word "Compass", (x,y) coordinates. Each symbol indicates a direction and stepsize. Lower case letters indicate a step half the length of their uppercase version.

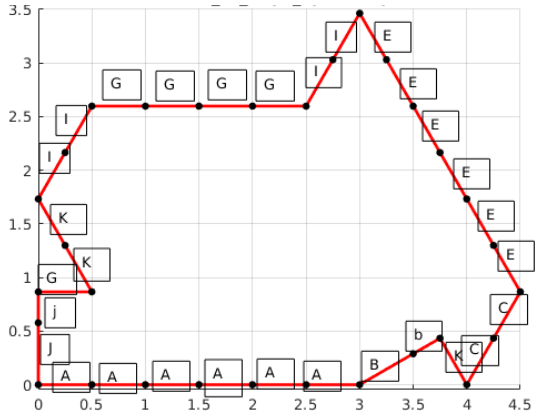


Figure 8: The Trinity object constructed from its boundary word, (x,y) coordinates, starting at (0,0).

of the grid corresponds to a successive letter in the boundary word.

The data **P** and **W** only record the position and shape of the boundary of the polygonal Eternity object. However, since we understand the structure of the underlying Eternity grid, we can use the boundary word as a skeleton, and then use the Eternity grid to fill it in with the internal elements and their connectivities.

6 Building Arbitrary Rectangular Grids

We have mentioned that the grid of elementary triangles used by the Eternity puzzle can be generalized to a rectangular grid of variable dimensions. This fact relies on the fundamental rectangular cell of 12 of the 30-60-90 elements. This means that any Eternity object can be thought of as a subset of such a grid. Hence, we can use our understanding of the regular grid to transfer information, such as coordinates or labels, to the Eternity object. Moreover, starting with the boundary word of the object, which only specifies the perimeter, we can use the underlying grid to identify and index the nodes and elements contained in the interior of the object.

In analogy to a checkerboard, we can think of the rectangular cell as defining a "red" square, and its upside down version, which has opposite parity, as a "black" square. If we tile a rectangular region by alternating this pair of cells, we can create grids of Eternity type that are as large as we wish. To suggest how the red and black cells fit together, consider Figure 9, in which the two cells are shown in a vertical stack. We will describe this as having height $h = 2$ and width $w = 1$, where these measurements refer to the number of cells involved. In general, an $h \times w$ rectangular cell grid will be h cells high and w cells wide. To settle a parity choice, we will always set up cell grids so that the cell in the lower left corner is a "red" cell, identical with the bottom cell in the reference double rectangular cell.

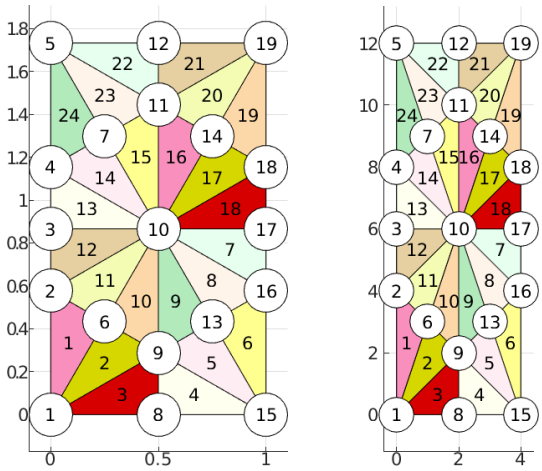


Figure 9: The $h = 2 \times w = 1$ double cell, showing (x,y) and (i,j) coordinates.

The numbering of the 24 elements of the double cell can be assigned somewhat arbitrarily. The 19 nodes, however, will be more carefully numbered, by starting at the lower left corner, proceeding upwards as far as we can, and then shifting to the right, one column at a time. This labeling process gives us a sort of map of the double cell, which we will use as a template when we create larger cell grids.

Suppose that we want to extend the grid by stacking more cells vertically. We will always ensure that the stacking is done in such a way that the cells alternate in parity, forming a sequence of cells, with possibly one final unpaired cell. Given this scheme, can we number the nodes and elements?

We can begin with the element numbering. If our stack involves just one or two cells, then we can simply copy the element numbers from our reference map. If there are more cells, then we process them in pairs, adding 24 to the element labels in the previous pair, until all elements are numbered. If we have h cells in the vertical stack, we will have labeled all $12\,h$ elements.

Now suppose that we have an $h \times w$ grid of cells. After we have numbered the elements in the first column of cells, we move to the neighboring column on the right. We repeat the numbering process, starting from the base value of $12\,h$. Continuing in this way, we will finally process the cells in column w, uniquely numbering all $12\,h\,w$ elements.

A similar process can be applied to the nodes. However, here we want to consecutively number all the nodes sharing the same x coordinate, before moving to the next column of nodes. Even viewing the double rectangular cell, it is clear that the columns of nodes vary significantly in number and position as we move from left to right. Moreover, when counting nodes, there is a sort of edge effect that has to be handled. For an $h \times w$ array of cells, we need to start with an initial "investment" of $2h + 2w + 1$ nodes along the extreme left and bottom boundaries. Thereafter, each additional cell adds 6 nodes. Hence an $h \times w$

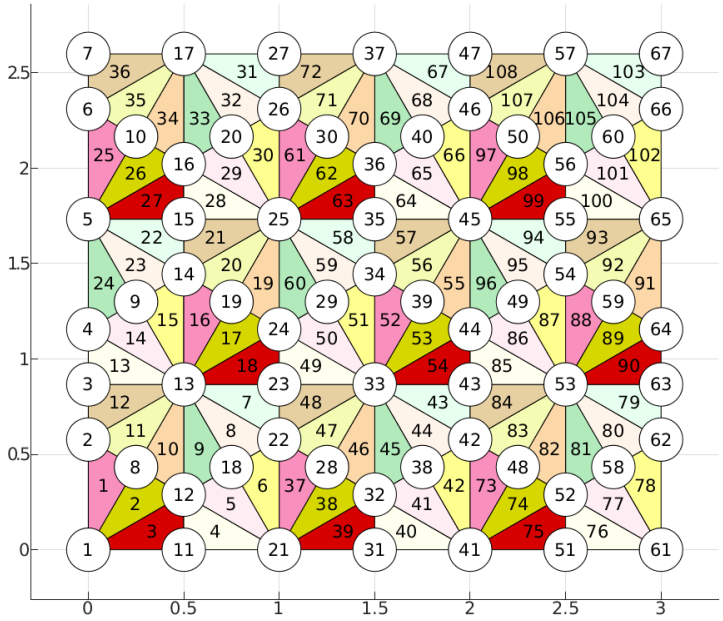


Figure 10: A 3x3 rectangle grid, (x,y) coordinates.

rectangular grid will contain a total of $6\,hw + 2h + 2w + 1$ nodes. As an example, Figure 10 displays a 3×3 cell grid in which nodes and elements have been numbered according to this procedure.

7 Filling in an Eternity Object from its Boundary Word

We plan to rely on the boundary word as our description of the Eternity objects we deal with. Even when we rotate, reflect, or translate them, we will want to do so by applying the appropriate operations to the boundary word. But the boundary word only gives us information to trace out the boundary of the region, and we will at times need to know the internal structure of the item as well, that is, the number, location, and adjacency of the nodes and elements. We will do this by creating a background cell grid, as described above, that is large enough that the Eternity object can be regarded as a subset.

Just from the boundary word information, we can determine the height and width of the Eternity object, which allows us to generate a background cell grid large enough to accommodate it. Some small horizontal and vertical adjustments may be necessary when matching the object to the grid, but this can be done automatically from the available data, by a small translation of the object.

Now the nodes and lines of the Eternity object are part of the cell grid. If we could visualize the cell grid and the object, we could simply read off the desired information, such as the coordinates of the object nodes. Instead, we must devise an automatic that can

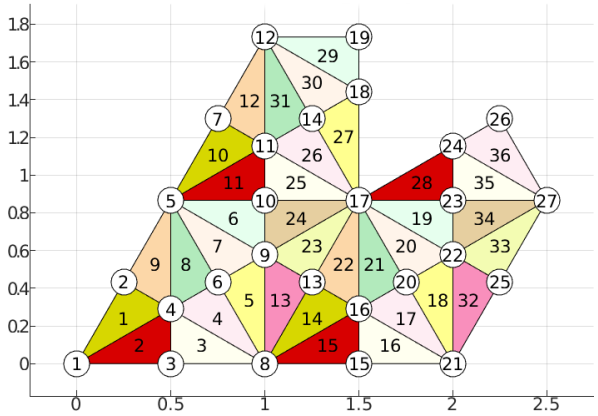


Figure 11: Eternity tile #11, with nodes and elements numbered.

retrieve this data.

Because the cell grid covers the object and properly aligns with it, every node in the object is also a node in the cell grid. We know how to sequentially index and locate every node of the cell grid. Therefore, for each cell grid node, we can pose the question of whether it is contained in the Eternity object as well. This question can be answered using a well known algorithm known as *point-in-polygon* [9]. Because some nodes will be on the exact boundary of the object, we can avoid roundoff issues by using an integer (i, j) coordinate system involving a simple linear transformation of our (x, y) system. This process produces an indexed list of the coordinates of all the Eternity object nodes, both on the boundary and in the interior.

We now perform a similar analysis to determine which elements (30-60-90 triangles) of the cell grid are included in the object. We do this by computing the centroid of each cell grid element and calling the point-in-polygon algorithm. If we determine that the element also belongs to the Eternity object, then we translate the representation of the element in terms of the Eternity object node numbers, and add it to the element list. Thus, the Eternity object elements are numbered in order of "discovery".

Thus, by using the rectangular cell grid as a template, we are able to reliably determine the number and arrangement of the nodes and elements in any Eternity object. As an example of this process, Figure 11 displays the numbering of nodes and elements for tile #11 of the Eternity tile set.

8 The (x, y) and (i, j) Coordinate Systems

We are working with 30-60-90 triangles, whose angles and opposing sides are correspondingly labeled (α, β, γ) and (A, B, C). Therefore, as physical objects, we know that, given the length

of side C as $||C||$, we have

$$||A|| = \frac{1}{2}\,||C||$$
$$||B|| = \frac{\sqrt{3}}{2}\,||C||$$

If we insist that the larger equilateral triangles associated with the problem are to have sides of length 1, then we must take $||B|| = \frac{1}{2}$ and so we have

$$||A|| = \frac{\sqrt{3}}{6}$$
$$||B|| = \frac{1}{2}$$
$$||C|| = \frac{\sqrt{3}}{3}$$

Once we have chosen our measurement scale, we can assign physical (x, y) coordinates to the nodes in the grid, to the vertices of a tile, to the endpoints of an edge. We can construct proper plots of the grid or any tile to scale. When we use this measurement scheme, we say we are working with the (x, y) coordinate system.

The (x, y) coordinate system is physically meaningful, but can be computationally awkward. For instance, given the (x, y) coordinates of a point, we might like to determine if it corresponds to a node, or lies on the edge between two nodes, or lies within the interior of a particular tile. These determinations require us to divide the vertical coordinate by an irrational number, and hence can result in very small, but nonzero, variations from the exact numeric result. Unfortunately, a very small error is enough to produce the wrong answer to our questions.

Looking at the geometry of the basic rectangular cell, we can see that a natural horizontal scale might be in terms of $||B|| = \frac{1}{2}$, while a natural vertical measurement might be based on $||A|| = \frac{\sqrt{3}}{6} \approx 0.2887$. In order to capture all the detail of the grid, we will use horizontal and vertical scales that are half this size. Given the structure of the Eternity grid, or simply looking at the basic rectangular cell, we see that, starting from the origin, all nodes are spaced an integer multiple of $||B||/2$ horizontally, and $||A||/2$ vertically. The converse is not true, since the Eternity grid is not a product grid.

Thus, we can use the (x, y) system implicitly, but convert to a simpler system, based on integer values, which we term the (i, j) coordinate system. In this system, a coordinate value of (a, b) means to go a horizonal steps right, and b vertical steps upwards, using the (i, j) stepsizes. The simplicity of the (i, j) system is illustrated by Table 2, which lists the (i, j) coordinates of the nodes in the $h = 2 \times w = 1$ double rectangular cell.

Because we have rescaled our (x, y) coordinates using different factors, distances and angles are slightly skewed in the (i, j) coordinates, but the general shape and structure of

Node	(i,j) Coordinates
1	(0, 0)
2	(0, 4)
3	(0, 6)
4	(0, 8)
5	(0,12)
6	(1, 3)
7	(1, 9)
8	(2, 0)
9	(2, 2)
10	(2, 6)
11	(2,10)
12	(2,12)
13	(3, 3)
14	(3, 9)
15	(4, 0)
16	(4, 4)
17	(4, 6)
18	(4, 8)
19	(4,12)

Table 2: (i, j) coordinates of nodes in the double rectangular cell. Refer to Figure 9

Letter	Angle	Stepsize	Step	Half step
A	0°	2.00	(2, 0)	
B; b	30°	2.82	(2, 2)	(1,1)
C	60°	3.16	(1, 3)	
D; d	90°	4.00	(0, 4)	(0, 2)
E	120°	3.16	(-1, 3)	
F; f	150°	2.82	(-2, 2)	(-1, 1)
G	180°	2.00	(-2, 0)	
H; h	210°	2.82	(-2,-2)	(-1,-1)
I	240°	3.16	(-1,-3)	
J; j	270°	4.00	(0,-4)	(0,-2)
K	300°	3.16	(1,-3)	
L; l	330°	2.82	(2,-2)	(1,-1)

Table 3: Boundary word table for the (i,j) coordinate system.

Eternity objects is reasonably preserved. For example, using the (i, j) coordinate system, it is much easier to sketch Eternity objects on a standard sheet of graph paper. Because node coordinates are always integers, it is easy to detect groups of nodes that lie on a straight line, to identify a node in a plot given its coordinates, or conversely to determine the exact coordinates of a node from its plot. At any time, we can convert an (i, j) coordinate to its corresponding (x, y) value using the formula:

$$x = \frac{1}{4}\, i$$
$$y = \frac{\sqrt{3}}{12}\, j$$

The boundary word **W** of an object can be used in the (i,j) coordinate system as well, as long as the coordinates of the starting point **P** are recomputed. The corresponding "compass" for interpreting the boundary word appears in Table 3, and is greatly simplified from the corresponding compass for the (x, y) coordinates, in Table 1

9 Representing the Tiling Process

We have the tools to represent the objects involved in an Eternity puzzle. Now we must represent a solution of the tiling puzzle by placing the tiles in their correct positions. To do so, we can imagine the physical process that would be involved. Assuming we know a solution, then we perform a sequence of moves. In each move, we select the next tile, initially stored in some "reference position", turn it through some angle, possibly flip it over, and then rigidly translate it to its correct configuration on the grid.

We will need a computational representation of each possible configuration, specifying a tile, a rotation, a reflection option, and a translation. For the Trinity puzzle, there are 142 configurations, whereas any tiling solution will comprise just 4 of them.

So far, we know how to describe a tile in a default position, using its boundary word **W** and base point **P**. It turns out that an given configuration corresponding to a given tile can be described by applying to this data a set of mathematical transformations corresponding to the rotation, reflection and translation steps.

The given set of tiles for the Eternity and Trinity puzzles have no rotational or reflective symmetry. This means that each tile always has 12 distinct orientations, and so the generation process does not need to worry that a previous orientation has been generated a second time. Here, the term "orientation" refers only to the rotations and reflections applied to a tile, which does not become a "configuration" until we also specify a translation.

We now consider the details of how these transformations are represented.

9.1 Rotations

Given a tile in some reference position, the first operation we want to perform is a rotation. Our options include rotations of $0°, 60°, 120°, 180°, 240°$ and $300°$. The rotation will be done about the point whose coordinates are given by **P**. Therefore, the base point **P** will not be changed by this transformation.

The boundary word **W** uses an alphabet that specifies direction. Thus, the letter A indicates movement to the east, along the $0°$ direction. Suppose we apply a $60°$ rotatation to a tile whose boundary word originally contained an A. The portion of the boundary corresponding to this A, which originally headed east, should now lie on a $60°$ angle. This corresponds to replacing the letter A by the letter C in the boundary word. In fact, all occurrences of A should be so replaced. Similary, the characters B and b should be replaced by $Dandd$, C by E, and so on, circularly shifting all boundary characters by two spots. If we applied a $120°$ rotation, the characters would shift by 4 positions, and so on.

Here are the boundary words for Eternity tile #1 after each possible rotation:

rotation	boundary word
0°	`AAAAAAEEFfIFfIII`
60°	`CCCCCCGGHhKHhKKK`
120°	`EEEEEEIIJjAJjAAA`
180°	`GGGGGGKKLlCLlCCC`
240°	`IIIIIIAABbEBbEEE`
300°	`KKKKKKCCDdGDdGGG`

9.2 Reflections

After rotation, a tile can be reflected, which is equivalent to turning it over. We choose to reflect about the horizontal axis. Recall that a tile is composed of elements, and that each element has a type, a signed integer with magnitude between 1 and 6. In a reflection, the type of each element in the tile reverses its sign.

Reflection across the horizonal axis can be applied to the boundary word by a simple alphabetical substitution, as occurred for rotations. The substitution table is as follows:

tile	boundary word alphabet
old	`ABbCDdEFfGHhIJjKLl`
new	`GfFEdDCbBAlLKjJIhH`

Thus, if Eternity tile #1 has already been rotated by 120°, and now a reflection is to be applied, we would compute the following sequence of boundary words:

rotation	boundary word
original	`AAAAAAEEFfIFfIII`
rotated	`EEEEEEIIJjAJjAAA`
rotated and reflected	`CCCCCCKKjJGjJGGG`

Our convention has been to assume that the boundary word corresponds to a counter-clockwise traversal of the boundary. When we apply this simple alphabetic substitution, however, the resulting boundary word describes a clockwise traversal. For our purposes, this is a harmless discrepancy, and so we tolerate it.

9.3 Translations

Once we have chosen a tile, a rotation and a reflection, it remains to choose a translation. We can think of the translation by focussing on what happens to the tile element labeled #1. Now that the tile has been oriented, element #1 has a particular type. When we translate the tile, this element must be superimposed on a grid element with this same type. So now that we have the tile orientation, we can determine the possible translations available to us simply by locating all grid elements of the same type as tile element #1.

The 30 degree node of tile element #1 is the base point identified as **P**. For any grid element of index #k, whose type matches that of tile element #1, we can imagine translating the tile so that tile element #1 is superimposed on grid element#k. Suppose the 30 degree node of the grid element has coordinates **Q**. Then the appropriate translation can be imposed simply by shifting the base point **P** to the base point **Q** using a translation of $Q - P$.

This guarantees that tile element #1 is properly superimposed on grid element \$$k$; however, the resulting configuration is only legal if *all* tile elements lie on some grid element. But in many cases, the translation might position some tile elements outside the grid. We

can confirm that each tile element is inside the grid by computing its centroid, and then doing a "point in polygon" query. If we get a positive answer for each of the 36 tile elements, then we can safely accept this choice of tile index, rotation, reflection and translation as a configuration that must be considered in the final tiling computation.

The configurations play the part of variables in our linear system. Now that we have determined how the variables are to be represented, we must construct the system of linear equations that represent the tiling conditions of our puzzle.

10 Constructing and Solving the Linear System

We have indexed the complete set of tile configurations that could be involved in a tiling of our puzzle grid. In terms of a linear system, we think of these configurations as our **variables**. Each configuration, by itself, represents a particular tile that could be placed on the grid in a particular way. To be considered a solution, however, we need to select a subset of all possible configurations that together exactly cover every grid element once. Selecting this subset is equivalent to assigning the values 1 or 0 to each configuration, indicating that is to be used, or not used, in a particular tiling of the grid.

As it happens, we can represent this task in terms of a linear system of equations, with a few additional constraints. Linear algebra provides a rich set of resources in theory and algorithms, which makes it possible to analyze our problem and automate its solution.

10.1 The Linear System for Tiling

To express the tiling problem as a linear system of equations, we need equations describing the interaction of the variables, the tile configurations are our variables. Each configuration can be represented by a list of the grid elements that it would cover. Correspondingly, each element can generate an equation expressing the condition that one, and only one, configuration must cover that element. Now we can start to see the structure of a matrix A, such that entry $A(i, j)$ is 1 if grid element i is covered by some element of tile configuration j. The right hand side vector b will have the value $b(i) = 1$, indicating that the covering must occur, and must occur just once.

Our linear system is not yet complete; we need to include the condition that each tile must be used exactly once. We know which tile was used to generate each configuration. Therefore, we must append an additional constraint equation corresponding to each tile. The constraint equation for the k-th tile must have a 1 in every column that corresponds to a configuration involving that tile, and a 1 on the right hand side, indicating that exactly one configuration of tile k was used in the solution.

We now have constructed a linear system of the form $A\,x = b$, that represents our tiling problem. Before we try to solve it, we must pose an additional condition: the only acceptable values in the solution vector x must be 0 and 1, with a 1 indicating that the corresponding

configuration was actually used in the solution. Unlike the tile counting constraints, this constraint cannot be expressed as another linear equation. It must be imposed in some external fashion by the equation solver.

There is no reason to think that our system is square, and in general it will not be, Moreover, for cases like the Eternity puzzle, the thousands of equations will be vastly outnumbered by the millions of variables. This means we should be prepared to solve an underdetermined linear system. We cannot determine in advance whether the system has zero, one, or several solution. However, since the linear system precisely represents the tiling problem, we can assert that there are exactly as many solutions of the linear system as there are distinct solutions to the tiling problem.

Because of its size and its restriction to binary solution values, this linear system is not suitable for treatment by a standard linear algebra package like `LAPACK`. It can, however, be regarded as a special kind of linear programming problem, for which there are many robust, flexible, and powerful packages available, including `CPLEX`, `GUROBI`, and `SCIP`. Thus, after forming the linear system, we proceed with our solution strategy by appealing to an external linear programming solver that can handle large systems, and that can honor our requirement for a binary solution.

10.2 Interacting with Linear Programming Software

Most linear programming software can process a problem if it is expressed using a common file structure known as the `LP` format. The `LP` format describes the linear system to be treated, an optional objective function, constraints on the data range, and other information such as a problem title.

Only the nonzero entries of the matrix are stored. In the Trinity matrix, here are 144+4 rows and 142 columns. However, each column contains only 36+1 nonzeros, out of the 148 entries. In the full Eternity matrix, again there will only be 37 nonzeros in each column, but now this represents a reduction from 2,508+209 entries in each column. The number of nonzeros in a row of the matrix is somewhat irregular, depending on the grid and the shape of the tiles. In some cases it could be as few as a single nonzero entry. For the Trinity puzzle, the first two equations look like this:

```
x76 = 1                          (Grid element 1 coverable by configuration 76)
x33 + x76 + x97 + x138 = 1       (Grid element 2 coverable by 33, 76, 97 or 138)
                                 (...and so on...)
```

After the equations describing the covering requirements, there follow equations specifying the constraint that each tile must be used exactly once. These constraint equations are very similar in form to the covering equations. For the Trinity puzzle, the first such equation looks like:

```
x1 + x2 + x3 + x4 + ... + x35 + x36 = 1  (Use exactly 1 of these configuration)
                                         (...and so on...)
```

because the first 36 configurations involve tile #1.

In the LP format, variables whose values are restricted to 0 and 1 values can be labeled as "binary". For our tiling problems, all variables will have this designation. For the Trinity puzzle, this section of the file looks like this:

```
Binary
  x1 x2 x3 x4 x5 x6 x7 x8 ... x141 x142
```

Once the LP file has been created, it is necessary to execute one of the linear programming applications. For instance, if the LP file is named trinity.lp, then the CPLEX program could be invoked with the following commands, to find all solutions and write them to the file trinity.sol:

```
set mip pool absgap 0.0
set mip pool intensity 4
set mip limits populate 10
set mip pool capacity 10
set output writelevel 1
read trinity.lp
populate
write trinity.sol all
quit
```

After the solution file has been created, a simple postprocessing procedure, written in a language such as MATLAB, can read the solution information as an array, which it can then analyze, plot, or print.

10.3 Recovering the Tiling Configurations from the LP Solution

A solution vector returned by the linear programming software is simply a sequence of 1's and 0's. The 1's identify the tile configurations used in the given solution. Given only that component k of the solution vector is a 1, how do we recover the corresponding configuration, that is, the tile that was used, and its rotation, reflection and translation values?

When we constructed the linear system $Ax = b$, we selected a tile, transformed it, and if it remained within the region, we assigned it the index value k, and then ... discarded the configuration information. This makes sense, because there is such a large number of configurations, only a few of which will actually end up being used in a solution. But once we know that configuration k is used, we want to recover that original information.

The way we have chosen to deal with this need is, once we know the configurations that were selected, to simply go through the configuration generation process again. Each

time that we generate a configuration, we momentarily have the associated tile index and transformations, which will be discarded when we move to the next configuration. But when we generate a configuration whose index k corresponds to a solution component, we can generate the transformed tile, including the coordinates of its component triangles, and write this to a file, or add an image of this selected tile configuration to a plot of the solution.

Given that, for tiling problems, the number of configurations can be very large, we are essentially trading time for space. We are willing to compute the set of configurations a second time, rather than setting aside a great deal of memory to allow a rapid lookup.

10.4 Solving smaller problems

From the beginning, it was decided to put off the issue of the size of the Eternity puzzle, and to focus on the development of a robust, general algorithm for solving tiling problems that used a similar grid structure and tile patterns. In order to facilitate development and experimentation, smaller versions of the Eternity puzzle were developed, in particular the Trinity puzzle. The linear algebraic solution approach has been successfully applied to a sequence of these smaller puzzles, running on desktop computers in a matter of seconds.

Thus, theoretically, we might claim to have solved the Eternity puzzle in principle. However, this is to neglect the fact that this class of tiling puzzles sustains an rapid growth in complexity once the grid size and number of tiles is increased past a certain threshhold. In such cases, a solution procedure that may apply theoretically might actually be unable to produce an answer over the lifetime of the universe. This means that our approach, while correct, may only be practical for problems limited to a certain size.

To give a hint of the magnitude of this issue, Selby [8] has suggested that, once an Eternity-type puzzle exceeds a certain bound on the number of tiles, which he estimated at between 70 and 80 tiles, the number of tiling solutions can grow explosively. A rough guess is that the Eternity puzzle actually may have as many as 10^{95} distinct tiling solutions. In fact, this observation was the key to Selby's success. There were so many distinct solutions that he could essentially stumble onto one, after partially filling in the grid in an intelligent way.

This might suggest that, as problem size increases, the linear algebraic approach can quickly reach a dead end as a stand-along solution procedure. However, it may still be the case that, in a hybrid approach like Selby's, the linear algebraic approach could be used in the second phase, when a much reduced grid is to be tiled, rather than using the search techniques that were employed at that stage.

11 Highlights of the Investigation

This study of the Eternity puzzle followed work on tiling problems for polyominoes. The two tiling problems are analogous, and so we were able to employ much of the modeling and

methods developed previously. However, as we worked on the Eternity puzzle, it became clear that its peculiar features required the discovery of new approaches and the construction of appropriate techniques to handle its complexity, including:

- The creation of smaller versions of the Eternity puzzle to facilitate testing and benchmarking;
- The analysis of the grid and tiles into the nonisotropic 30-60-90 triangular elements;
- The construction of an underlying Cartesian grid;
- The use of an (i, j) integer coordinate to simplify work and guarantee exact results;
- The identification of a basic cell of 11 nodes, that could be used, along with its reflection, to create rectangular grids of any size, to be used as a background for any Eternity object;
- The use of the basic cell to allow for the consistent numbering of nodes and elements in any Eternity object;
- The extension of the boundary word idea from 4 letters to 12, including a varying step size;
- The discovery of how to apply rotations, reflections and translations to a boundary word;
- The use of the basic cell to determine the internal structure of an Eternity object from its base point and boundar word;
- The use of the "point in polygon" test to check whether a transformed tile remains entirely inside the grid;
- The recovery of the details of a given configuration, given only its index from the `LP` solution to the tiling problem, by repeating the configuration construction process;

All of these concepts had to be thought through, then implemented in software, and tested. The goal of a tiling solution seemed to be just around the next corner, but making the turn, we found yet another corner. It was, therefore, an immensely satisfying moment to completely solve the Trinity problem,and several of its "cousins".

12 Conclusion

This investigation is inspired by the Eternity puzzle. However, it became clear that in order to test and debug the software being developed, a much smaller problem was necessary. This was the origin of the Trinity puzzle, which was an example used in a talk by Alex Selby. The grid involved 144 triangles, and the 4 tiles each had the usual 36 triangles. A solution to the puzzle was provided, but it was not stated whether there were more solutions.

The boundary words for the region and the four tiles were generated. The linear system was generated with 144+4 rows and 142 columns. `CPLEX` was used to analyze the system, and it returned a single solution vector x. This solution vector was then postprocessed to create a plot of the tiling, and a list of the transformations used on each of the four tiles.

The operations of generating the linear system in MATLAB, solving it with CPLEX, and postprocessing it back in MATLAB each took less than a minute.

Theoretically, the treatment of the Eternity puzzle is no different from that for Trinity. However, it should be clear that that problem is enormously more challenging. Generating the 209 boundary words for the tiles took a significant amount of patient sketching and programming. Just generating the matrix A is expected to be a substantial task. It will have 2,508+209 rows. The number of variables will be much larger, approximately equal to $e * t$, the product of the number of elements and the number of tiles. To see this, pick any tile, and identify one of its elements as "special". Then pick some element of the grid. There is exact one configuration of the tile that covers the grid element by the special tile element. Thus, there are exactly e distinct ways to place this tile on the grid. However, not all of these configurations are actually legal. We must reject any configurations in which part of the tile lies outside the grid. Thus, the number of distinct configurations of a given tile is no more than the number of elements in the grid. Hence the total number of configurations of all the tiles is no greater than $e * t$. This gives us an upper bound on the number of linear programming variables.

We cannot extrapolate a necessary processing time for the Eternity problem based on the size of Trinity, and some other intermediate test puzzles. If the current algorithms do not seem able to handle the problem, it may be necessary to search for other techniques to generate the configurations. Once we move to the LP solution stage, we anticipate running the problem through a version of CPLEX on a parallel computer system with extensive memory resources. If CPLEX cannot solve our problem, or cannot solve it in a reasonable time, then we may have to conclude that our approach, while mathematically correct, is currently computationally unfeasible as a stand-alone approach.

This discussion has laid out the models for viewing the Eternity puzzle, and the techniques designed to find solutions to the tiling problem. The example case Trinity has been successfully handled in this way. The results for the Eternity problem will be discussed in a later presentation.

A Complexity Study

The Eternity puzzle certainly looks "complex", but computationally, it's important to be able to estimate the amount of work involved in producing a solution. For our approach, once the linear system is formed, the solution process is passed off to a linear programming package. Only very limited information is available for how that process is carried out, and therefore, often the measure of complexity is simply reported as the time in seconds required for the computation.

A more accessible measure is the the number of rows m and columns n in the linear programming matrix, representing constraints and tile configurations, respectively. Often, knowing the size of the problem, we can predict the necessary computational work and

Name	Elements (e)	Tiles(t)	Equations (m)	Variables (n)	Variable prediction(e*t)
Trinity	144	4	148	142	576
Serenity	288	8	296	876	2,304
Whale	288	8	296	1,071	2,304
Boat	756	21	777	8,753	15,876
Pram	2304	64	2368	108,049	147,456
Eternity	7524	209	7733	?	1,572,516

Table 4: Complexity data for a sequence of tiling problems.

running time. For instance, using Gauss elimination to solve a linear system with a square matrix A, we have $m = n$, and both the number of arithmetic operations and the solution time will tend asymptotically to be proportional to $\frac{1}{3}n^3$. A linear programming problem is, in some ways, of similar structure to a simple linear system. Hence it is reasonable to expect that a doubling of the problem sizes m and n will result in a similar dramatic increase in work and running time, possible of quadratic, cubic, or even higher order.

In order to predict the required solution time for the linear algebraic approach to the Eternity puzzle, a sequence of smaller problems was devised and solved. Each problem in the sequence had between 2 and 3 times the number of elements e and tiles (t) as the previous one. For these smaller problems, we constructed the linear programming matrix and recorded the number of equations and variables. While the number of equations is always $e + t$, the number of variables depends in a complicated way on the geometry. However, we can overestimate the number of variables by $n \approx e * t$. Although the Eternity problem itself was not processed, we include the known information and the predicted number of variables.

Using this table, we can estimate the size of the Eternity linear programming matrix, and can make a guess as to the expected time required for a solution.

B Computer Software

During this initial work, a number of MATLAB files have been created. These files are publicaly available in a set of related web directories. Each web directory has an index page which briefly describes its contents. In general, the directories come in pairs, with a source code directory and a test directory.

The directories are located under `https://people.sc.fsu.edu/~jburkardt/m_src` and the directory page repeats the name of the directory with an `.html` extension. Thus, the directory page for eternity is:
`https://people.sc.fsu.edu/~jburkardt/m_src/eternity/eternity.html`

Source code	Tests	Comments
eternity	eternity_test	7,524 elements, 209 tiles
eternity_tiles	eternity_tiles_test	more data for Eternity tiles
trinity	trinity_test	144 elements, 4 tiles
serenity	serenity_test	288 elements, 8 tiles
whale	whale_test	288 elements, 8 tiles
boat	boat_test	756 elements, 21 tiles
pram	pram_test	2304 elements, 64 tiles

References

[1] John Conway, Heidi Burgiel, Chaim Goodman-Strass, "Chapter 21, Naming Archimedean and Catalan polyhedra and tilings", **The Symmetries of Things**, A K Peters/CRC Press, 2008.

[2] Herbert Freeman, "Computer processing of line-drawing images", *ACM Computing Surveys*, Volume 6, Number 1, March 1964, pages 57-97.

[3] Herbert Freeman, "On the encoding of arbitrary geometric configurations", *IRE Transactions on Electronic Computers*, Volume EC-10, Number 2, 1961, pages 260-268.

[4] Marcus Garvie, John Burkardt, "A new mathematical model for tiling finite regions of the plane with polyominoes", *Contributions to Discrete Mathematics*, Volume 15, Number 2, July 2020.

[5] Solomon Golomb, **Polyominoes: Puzzles, Patterns, Problems, and Packings**, Princeton University Press, 1996.

[6] "Who wants to ruin a millionaire?", The Guardian, 30 October 2000.

[7] Ed Pegg, "Polyform Patterns", in **Tribute to a Mathemagician**, Barry Cipra, Erik Demaine, Martin Demaine, editors, pages 119-125, A K Peters, 2005.

[8] Alex Selby, "Notes from talk", `https://www.archduke.org/eternity/talk/notes.html`

[9] Moshe Shimrat, "Algorithm 112: Position of Point Relative to Polygon", *Communications of the ACM*, Volume 5, Number 8, August 1962, page 434.

[10] Mark Wainwright, "Prize specimens", *Plus Magazine*, 01 January 2001.