

Ben Sauerwine
Intro to Research II
Progress Report – January 15, 2006

Problem Description:

In creating custom alloys or ceramics, a natural question to ask is “what is the optimal proportion of two agents in order to maximize the frequency of a desirable structure”? In this research, a model for the defect-free combination of two agents is proposed and examined and a means of approximating and counting the number of lattices exhibiting a particular structure on a torus is developed.

Two-Dimensional Model:

Let me now examine some possible toroidal lattices and briefly discuss the features that a toroidal lattice must exhibit.

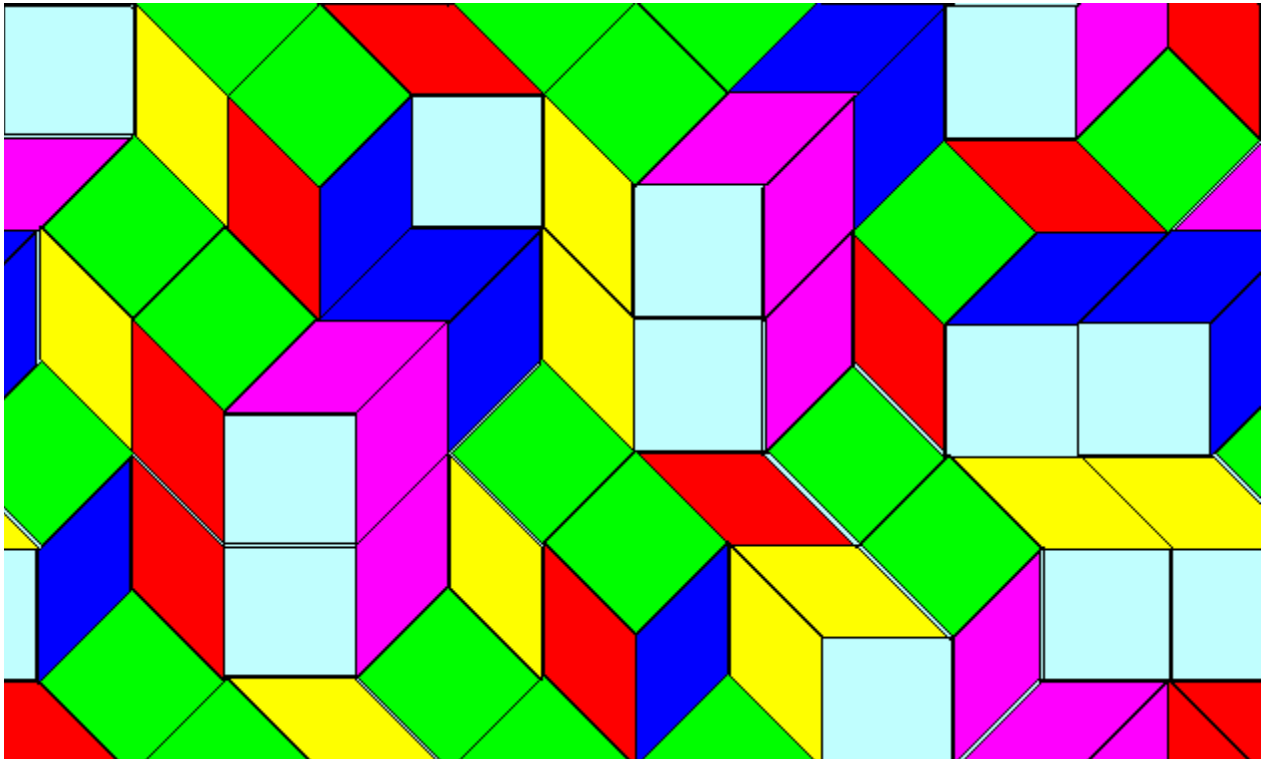


Figure 1: $4 + \frac{8}{\sqrt{2}} \times 3 + \frac{4}{\sqrt{2}}$

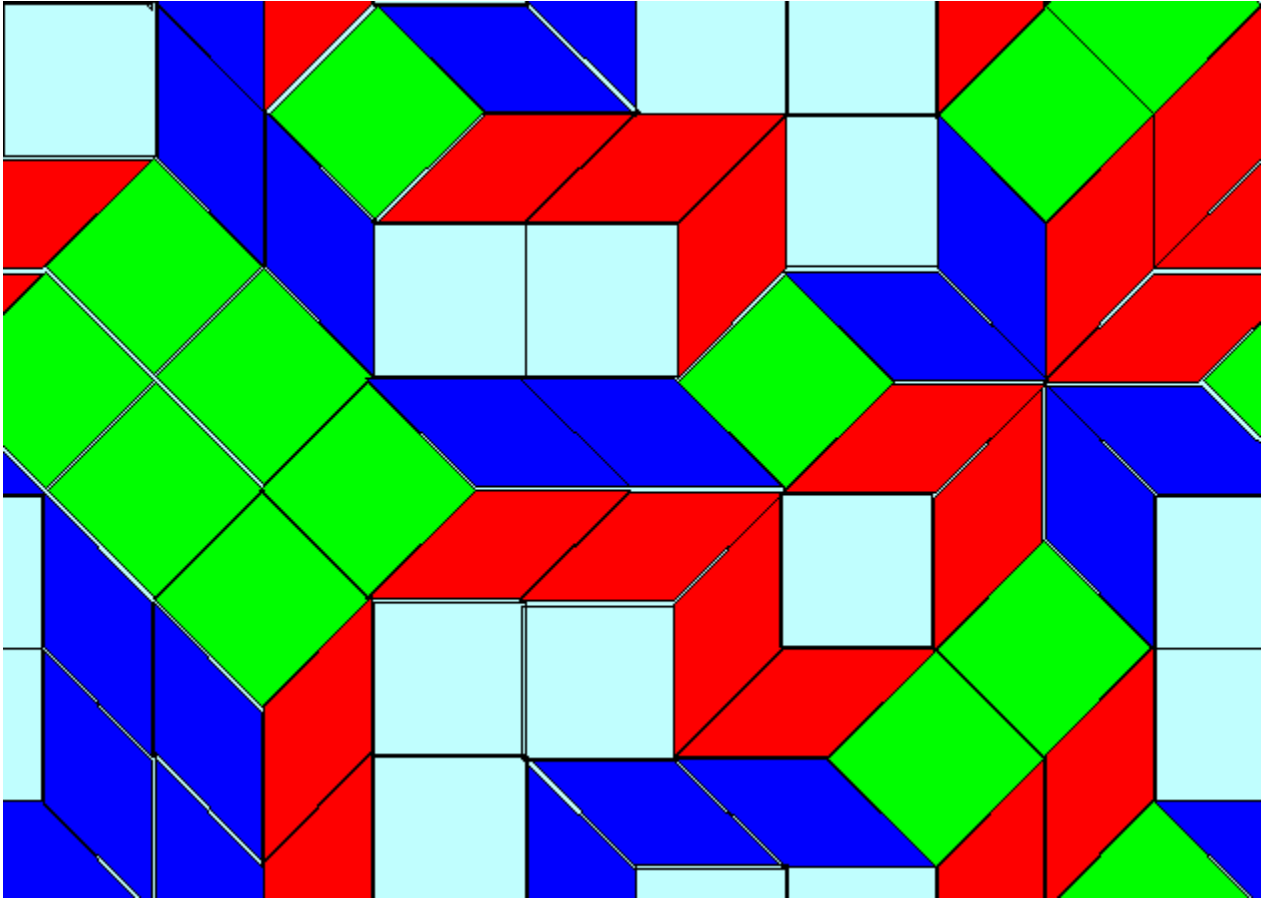


Figure 2: $4 + \frac{6}{\sqrt{2}} \times 3 + \frac{4}{\sqrt{2}}$

You may make tilings like this using the Tile Tool at:
<http://www.andrew.cmu.edu/user/bsauerwi/research/TileTool.exe>

Features of the toroidal lattices include:

‘Orthogonality’ of Dimensions: Within the torus, the only way to advance in a direction by a unit of $\frac{1}{\sqrt{2}}$ is with a rhombus, and the only way to advance in a direction by a unit of 1 is with a square. Because of this, the number of squares and rhombi and further the number of squares parallel to the axes of the torus versus the number of squares rotated is fixed by the dimensions of the torus. Namely, if the dimensions of a torus are given by $N_1 + \frac{M_1}{\sqrt{2}} \times N_2 + \frac{M_2}{\sqrt{2}}$

The number of squares parallel to the axes is always $N_1 N_2$.

In order for the torus to be periodic, both M_1 and M_2 must be even. The number of rhombi that must appear is $N_1M_2 + N_2M_1$, and the number of rotated squares must be $\frac{M_1M_2}{2}$.

‘Streaks’: I notice that the rhombi only appear in ‘streaks’ of two varieties. That is, each rhombus is attached to another rhombus oriented in one of only two ways. You’ll notice that in figure 2, the streaks colored for emphasis proceed only up and right or down and right. Where they cross, a green, rotated square appears. I notice that streaks traveling in the same direction may never cross. For illustration, here is a ‘streak diagram’ for figure 2:

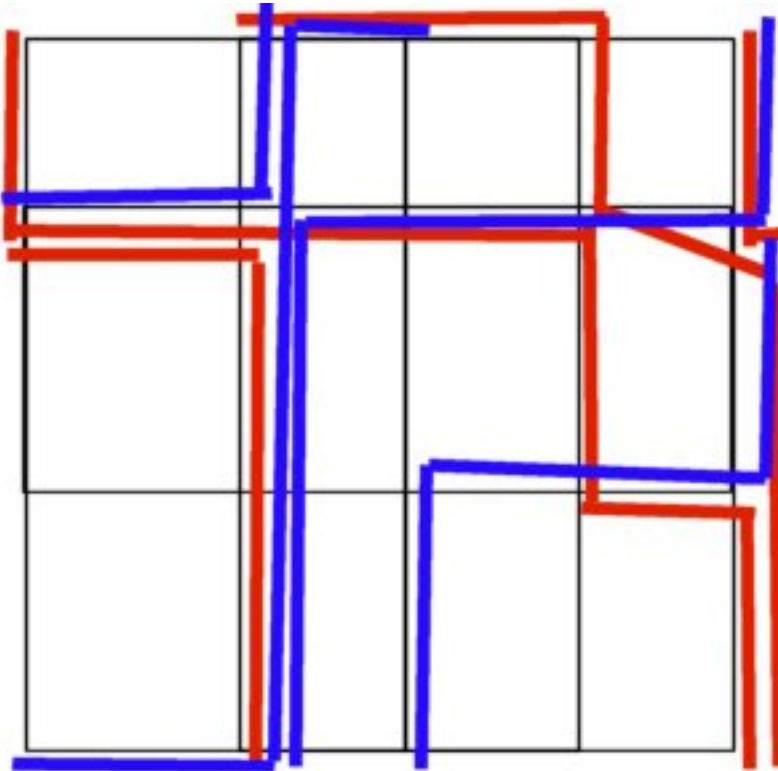


Figure 3: A Streak Diagram of Figure 2

So I see that I get the expected properties of these “streaks” of rhombi: Namely, the streaks going in the same direction do not cross. I also notice that the streaks going in opposite directions are almost independent of one another: except where they cross while going parallel to each other, a streak diagram uniquely identifies each possible situation. For instance:

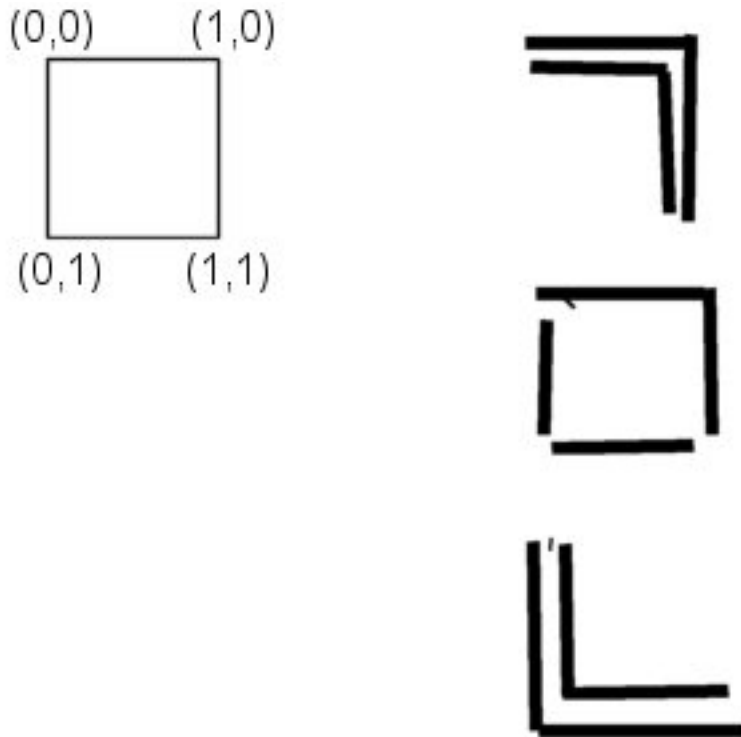


Figure 5: A simple example of the staircase algorithm.

Interestingly, the basic staircase algorithm is rather simple to implement using Dynamic programming. It works as follows:

Inputs: Number of staircases, endpoints for each staircase. This is an ordered list, corresponding to the “lowest” to the “highest” staircase’s endpoint.

Basis Case 0: If any staircase is to end at a point with a coordinate less than zero, return 0.

Basis Case 1: If all staircases are to end at $(0, 0)$, return 1.

Basis Case 2: If all staircases of the longest length are not properly ordered, e.g. their endpoints require a “lower” staircase to end somewhere above and to the right of a “higher” staircase, return 0. The image below illustrates proper ordering versus improper ordering.

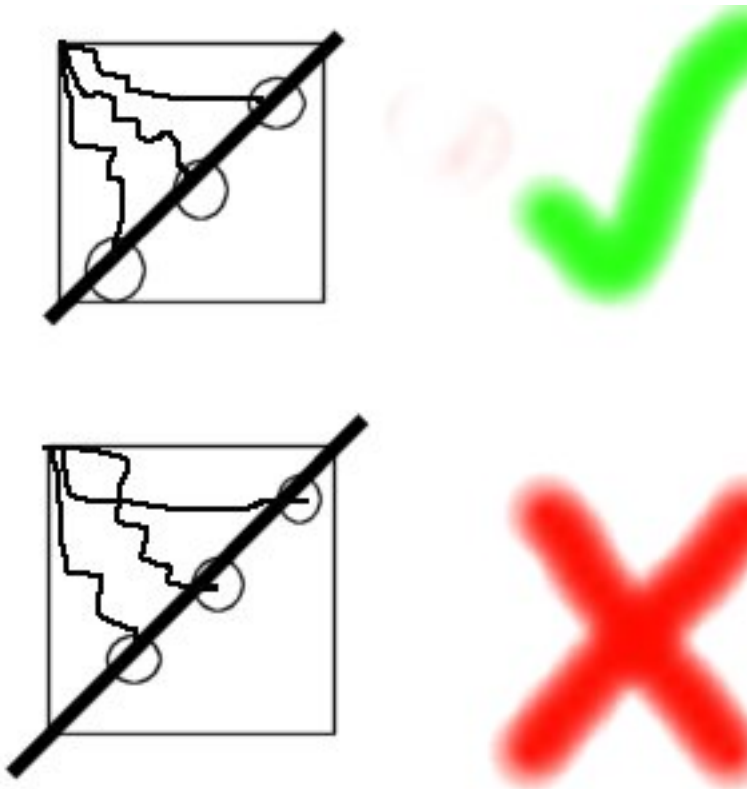


Figure 6: Proper ordering versus improper ordering

Note that by requiring the top staircase to end further down on the equal-distance line than the middle staircase, I have an impossible situation. A little thought will reveal that when this test is done for all of the longest lines at each step, it eventually must “cut out” impossibilities when the long lines are shortened.

Default Case: If neither of the basis cases succeeded, choose any of the longest staircases. Now return the sum of this function called with this staircase ending at the coordinate directly above this one and this function called with this staircase ending at the coordinate directly to the left of this one. In this way, I consider two cases that contribute to this one.

Advantages of this algorithm: This algorithm is suitable for grid computing, as the dynamic programming aspect makes it very easy to divide a problem into sub-problems to be shared among an array of computers.

Implementation and Code:

Please see the code and program available at:
<http://www.andrew.cmu.edu/user/bsauerwi/Tiling/Basic/>

Memory and Time Requirements:

Memory: $O((width \cdot height)^{pathcount})$

Basis Case 0: If any staircase is to end at a point with a coordinate less than zero, or if any staircase is to have a run less than zero, return 0.

Basis Case 1: If all staircases are to end at (0, 0), return 1.

Basis Case 2: If all staircases of the longest length are not properly ordered, e.g. their endpoints require a “lower” staircase to end somewhere above and to the right of a “higher” staircase, return 0. The image below illustrates proper ordering versus improper ordering.

Default Case: If neither of the basis cases succeeded, choose any of the longest staircases.

If this staircase is chosen to be limited in the right direction:

Return the sum of this function called with this staircase ending at the coordinate to the left with maximum run one smaller than the maximum run for this one plus the function called on the direction above with vertical run one smaller than the maximum.

If this staircase is chosen to be limited in the top direction:

Return the sum of this function called with this staircase ending at the coordinate to the left with maximum run one smaller than the maximum plus the function called on the direction above with vertical run one smaller the maximum run for this.

Some thought will make the following clear:

-The value for the maximum run size in either direction for a particular entry will be the same.

-This algorithm is an extension on the basic algorithm with the added constraint that “the left or top run for this particular cell for this particular staircase must be less than or equal to [x]”

Advantages of this algorithm: Again, this algorithm is suitable for grid computing, as the dynamic programming aspect makes it very easy to divide a problem into sub-problems to be shared among an array of computers.

Implementation and Code:

Please see the code and program available at:

<http://www.andrew.cmu.edu/user/bsauerwi/Tiling/Advanced/>

Memory and Time Requirements:

Memory: $O\left(\left(2 + \max left + \max top\right) \cdot width \cdot height\right)^{pathcount}$

Time: $O\left(pathcount \cdot \left(2 + \max left + \max top\right) \cdot width \cdot height\right)^{pathcount}$

Adapting the Staircase Algorithm to Toroidal Tilings: Issues and Solutions

Now there are two major issues to address in adapting this algorithm to finding Toroidal tilings. One is related to the nature of the streaks, and the other is related to connecting the streaks to one another at the borders.

First, I observe that these “streaks” may loop around multiple times, and that each streak may must loop around the same number of times. Notice how the red streak in figure 2 loops around 3 times, and the ones in figure 1 each loop only once. It is possible that the up streaks and the down streaks loop around a different number of times, and the divisibility of each dimension dictates exactly how many streaks may loop around how many times. To account for this, I may run the algorithm with a multiple of the height and width with fewer streaks and use the height and width as the maximum streak size. This will not affect the runtime of the algorithm.

The next problem comes from the property above also: I need a means of keeping the “streaks” within a certain range of each other, both vertically and horizontally. This prevents multiply wrapping streaks from overtaking one another, which is certainly not allowed. One method that is worth exploration might work as follows: for each step of the lowest streak, insert four additional streaks:

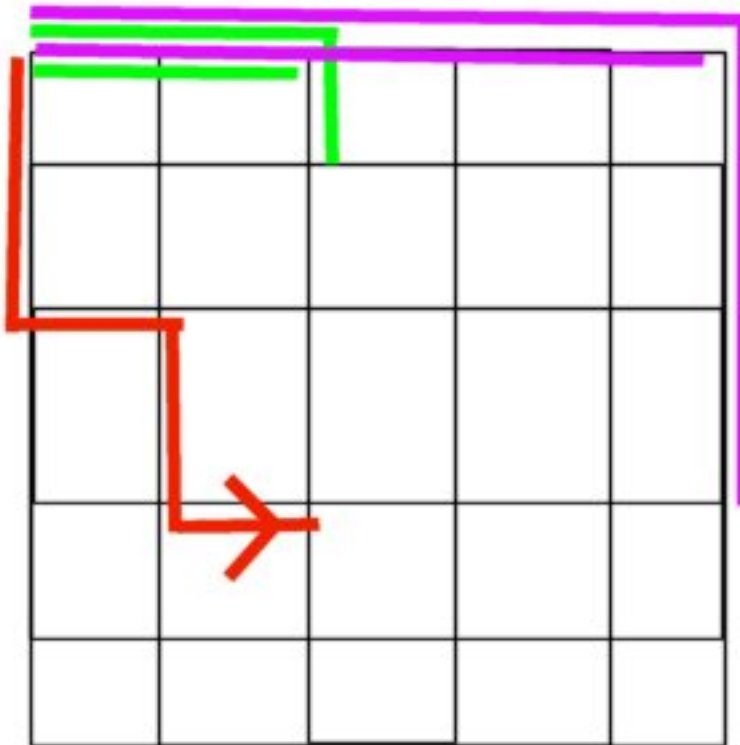


Figure 7: A Possible Solution to the “Keep Within” dilemma. If I want to keep within 2 units vertically and 3 units horizontally of the point ending at the red arrow, then I may add four additional streaks, ending where shown. There is only

one choice for each, so this will not affect my count but will cut off any other paths that may reach these points. This method will affect the runtime dramatically.

The final issue is actually connecting these paths at the end. By requiring the lowest path to travel the width of the torus right and the height down, one certainly constrains the first path to form a loop. The next step will be as follows, to force the remaining paths to connect:

For each path beyond the first:

Insert two new staircases to indicate a “starting point”, as with the green staircase in Figure 7. Do this for every possible choice of starting point for each path. This will result in an exponent increase of two for each path beyond the first in the dynamic programming algorithm.

Add the results for each path ending that many to the right of the leftmost path. The algorithm will then be able to reuse its array again for each case. This is a polynomial-time operation.

Counting Arrangements with a Desired Property

In order to count arrangements with a desired property, the first step is to design all substructures that could lead to the creation of the desired structure, and translate them into a partial staircase picture. This should not be particularly difficult for any structure. I will go through the steps with a sample structure.

Sample structure: two normal-orientation squares adjacent to one another but rhombi on three sides, two of which are long, on a torus with one up-staircase and one down-staircase with no loops.

