UNIVERSITÉ DU LUXEMBOURG

Faculty of Science, Technology and Medicine

Bachelor in Mathematics

Academic Year 2021 - 2022

Random Tilings & the Arctic Circle Theorem

Experimental Mathematics

Authors:
Sebastien Plaasch
Julien Pezzi

Supervisor:
Prof. Pierre Perruchaud

Contents

Introduction

The arctic circle theorem of Jockusch, Propp, and Shor states that uniformly random domino tilings of an Aztec diamond of high order are frozen with asymptotically high probability outside the “arctic circle” inscribed within the aztec diamond.
Let us define the different mathematical objects that are involved but also the inutuiton behind the theorem.

Definition 0.1. An *Aztec Diamond* A_n of order $n \in \mathbb{N}^*$ is defined as the union of all the lattice squares $[a, a+1] \times [b, b+1] \subset \mathbb{R}^2$ $(a, b \in \mathbb{Z})$ that lie completely inside the tilted square:

$$|x| + |y| \leq n + 1 \tag{1}$$

Definition 0.2. A domino is a closed 2×1 rectangle in $\mathbb{R}^2$ with corners in $\mathbb{Z}^2$, oriented either horizontally or vertically.

Definition 0.3. Suppose we have an aztec diamond of order n that we can cover with dominoes without overlapping. There will be several possible tilings. A randomly chosen tiling among the set of all the possible tilings, is defined as a *random tiling*.

Before considering random tilings on aztec diamonds, let us have a look at a random tiling on a tilted square:

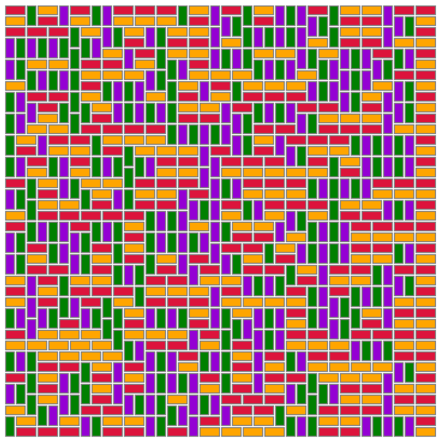


Figure 1: The Random Tiling of the Square of Side 40

By looking at the random tiling of the square of side 40, we observe that there are not any kind of organization inside of it. In contrast, let us now consider an aztec diamond of order 20 and let us uniformly choose 2 random tilings. Assume that we get the two following tilings.

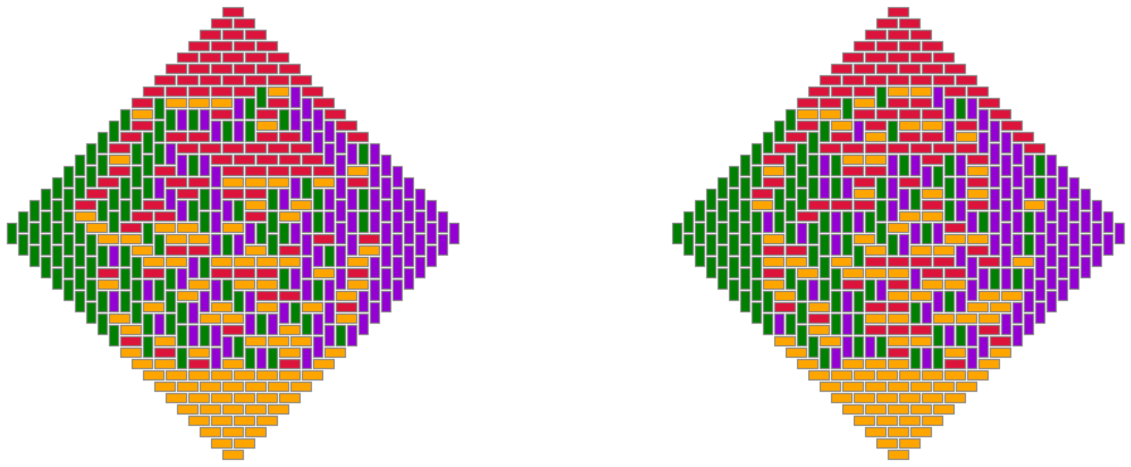


Figure 2: Two Random Tilings of a 20-Order Aztec Diamond

One can notice that, for both of the tilings, the four regions adjacent to the corners of the Aztec diamond, also known as the *polar regions*, are organized. By organized, we mean that each corner of the random tiling have the same orientation. Indeed, the upper and the lower polar regions of the aztec diamond are both horizontally tiled whereas the left and right polar regions are vertically tiled. The circle delimited by the polar region is called the *arctic circle*.
We could keep choosing random tilings of that 20−order aztec diamond, and we will almost certainly end up with similar organized tilings. This is the intuition behind the Arctic Circle theorem.
Now that we have an idea on what the theorem state, let us explain what will be done throughout in that report.
On the one hand, we will study theoretical properties of the random tilings by defining them as Markov chains. We will also show that there are $2^{n(n+1)/2}$ possible tilings for an Aztec diamond of order $n \in \mathbb{N}^*$.
On the other hand, we will explain the technichal implementation of the random tilings code. Then, we will run several simulations to point out some phenomena. We will especially consider the mixing time of the organized tiling (a tiling in which all the dominoes are laid horizontally). We will give an approximation of the needed number of rotations in order to get a tiling whose polar regions are well-tiled.

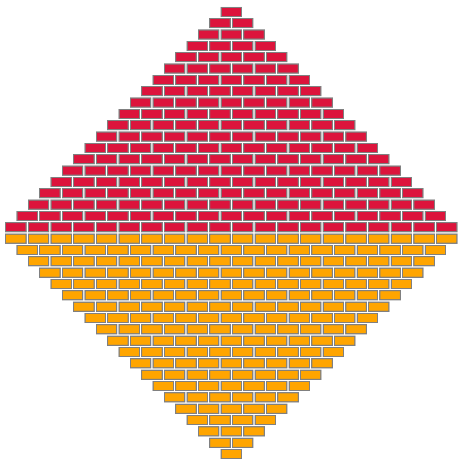


Figure 3: The Organized Tiling of A_{20}

1 Theory on Random Tilings

When running some simulations for several aztec diamonds A_n and interpret the results, we will need to know the number of possible domino tilings. This leads us to the aztec diamond theorem stated as follows.

Theorem 1.1. *Let T_n denote the number of the possible domino tilings of the aztec diamond A_n. Then:*

$$\mathrm{T}_n = 2^{n(n+1)/2} \tag{2}$$

Remark 1.2. One can notice that the number of possible configurations can rapidly become really large. Hence, when considering aztec diamonds of high order, it is computationally impossible to determine all possible configurations and uniformly choose one at random.

In order to prove that theorem, let us introduce some intermediate results.

1.1 Proof of the Aztec Diamond Theorem

Definition 1.3. Let $\mathrm{A}_n \subset \mathrm{A}_{n+1}$ be two Aztec diamonds. We define a node of A_n as a point $(i, j) \in \mathrm{A}_{n+1}$ satisfying $i + j \equiv 0 \mod (2)$. We call:

- interior nodes, the nodes contained in A_n
- closure nodes, the extreme points of A_{n+1}

Definition 1.4. We define a *lattice square* of A_n as a 1×1 square whose corners are lattice points in A_{n+1}. We call *boundary square*, a square contained in $\mathrm{A}_n + 1$ but not in A_n.

Let us give a visual example of what nodes and lattice squares are.

Example 1.5. In the figure below, the red dots represent the closure nodes and the black ones the interior nodes of the aztec diamond A_2.
In order to point out that the closure nodes are the extreme points of A_3, we used dashed line to draw A_3.
The lattice squares of A_2 solely are the 1×1 squares in A_3 defined by the dotted lines of the grid.
We colored in green all the boundary squares.

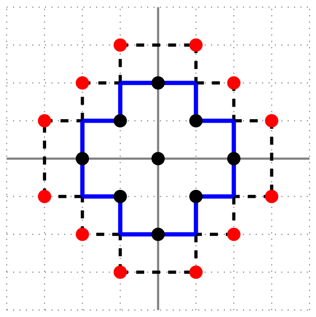


Figure 4: A_2 and its Nodes

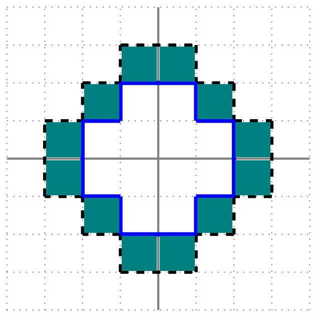


Figure 5: A_2 and its Boundary Squares

Now that we know what a node and a lattice square are, let us define the fields of arrows.

Definition 1.6. We call *field of arows*, the collection of arrows pointing from one node's corner to another satisfying the arrow field condition.
The arrow field condition states that each interior node N is either:

- *attracting* all adjacent arrows point towards N

- *repelling* all adjacent arrows are point away from N
- *transient* any two collinear arrows adjacent to N point in the same direction

Definition 1.7. Any domino of a tiling can be defined with nodes and arrows. It satisfies the domino condition stated as follows:

- two corners of the domino are nodes
- the two arrows inside the domino point towards these corner nodes

Proposition 1.8. *Following the arrow field condition and the domino condition, we can associate to each tiling of* A_n *a field of arrows by building the arrows domino by domino with* $n \in \mathbb{N}^*$*.*

Example 1.9. Let us represent with fields of arrows each of the eight possible tilings of the aztec diamond of order 2 A_2.

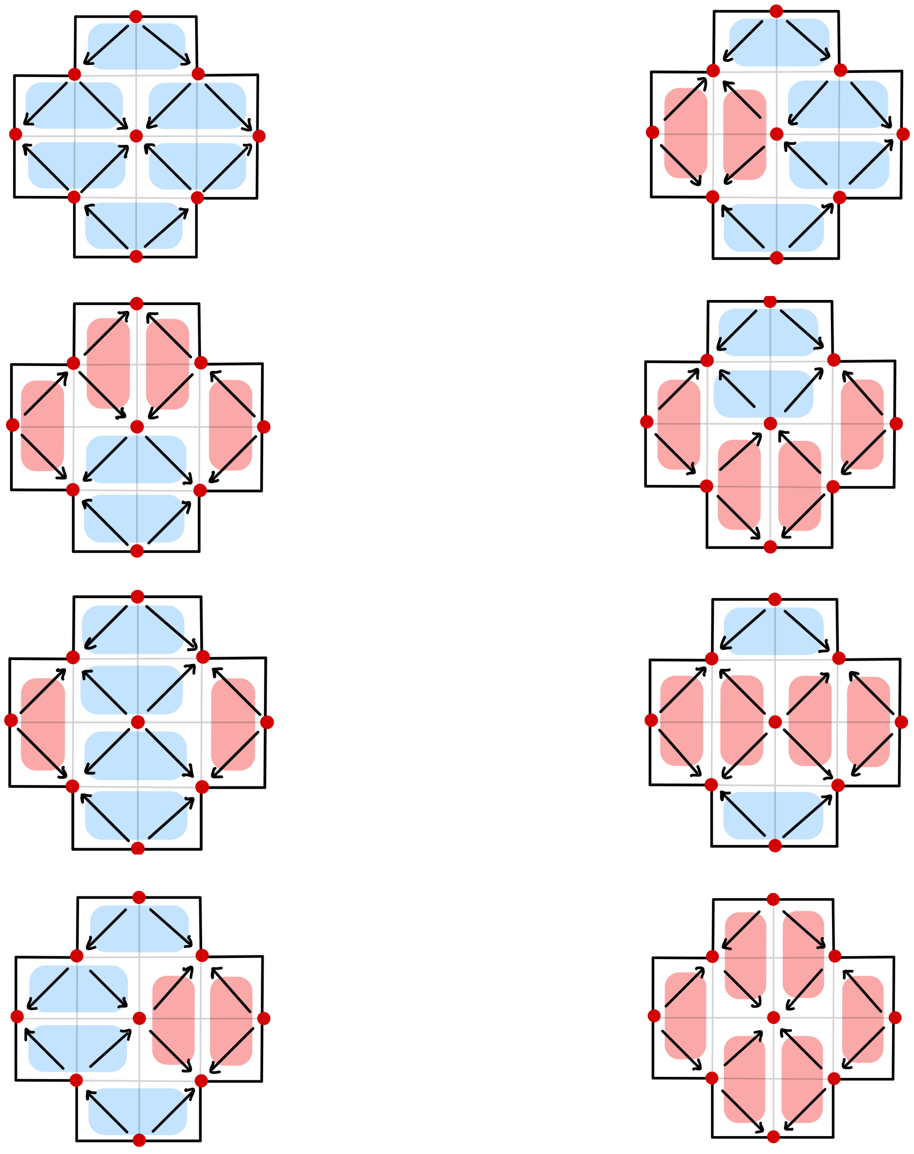


Figure 6: Random Tilings of A_2 and their Respective Representation with Field of Arrows

One can remark that a field of arrows can represent 2 different random tilings. This raises the following question about fields of arrows. What is the number of tilings for a fixed field of arrows?
In order to show that, let us introduce a few more definitions.

Definition 1.10. We qualify of outward pointing (respectively inward pointing) a field of arrows whose all arrows in boundary squares point outward (respectively inward).

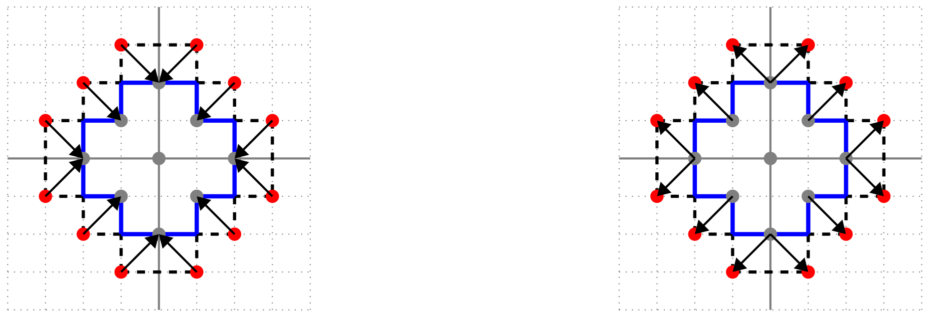


Figure 7: An Inward Pointing Field of Arrows and an Outward Pointing Field of Arrows of A_2

Notation 1.11. We will denote by:

- $\mathcal{T}_n$ a tiling of an aztec diamond of order n
- $F_n(\mathcal{T}_n)$ a pointing field of arrows on A_n for each tiling $\mathcal{T}_n$ of A_n
- $r(\mathcal{F}_n)$, the number of repelling nodes for a field of arrows $\mathcal{F}_n$
- $a(\mathcal{F}_n)$, the number of attracting nodes for a field of arrows $\mathcal{F}_n$

Also, from now on we will use the following color code:

- blue for repelling nodes
- grey for transient nodes
- green for attracting nodes
- red for closure nodes
- black for an interior node that can be transient, repelling or attracting

We are now ready to show the following lemma, which gives us the number of tilings for a fixed field of arrows.

Lemma 1.1. *Given an inward pointing field of arrows $\mathcal{F}_n$, there are precisely $2^{r(\mathcal{F}_n)}$ domino tilings $\mathcal{T}_n$ of A_n satisfying $F_n(\mathcal{T}_n) = \mathcal{F}_n$.*
Respectively, given an outward pointing field of arrows $\mathcal{F}_{n+1}$, there are precisely $2^{r(\mathcal{F}_{n+1})}$ domino tilings $\mathcal{T}_{n+1}$ of A_{n+1} satisfying $F(\mathcal{T}_{n+1}) = \mathcal{F}_{n+1}$.

Proof. Let us prove the second statement of the lemmma.
First of all, let us consider an outward pointing field of arrows $\mathcal{F}_{n+1}$. For each lattice square, let us draw the corners of the square that lie in direction of its arrow. If the field of arrows is associated to a tiling, note that the corners we just drew are nothing else but the corners of the dominoes. The two images below describe a field of arrows on which we drew the corners of the lattice squares.

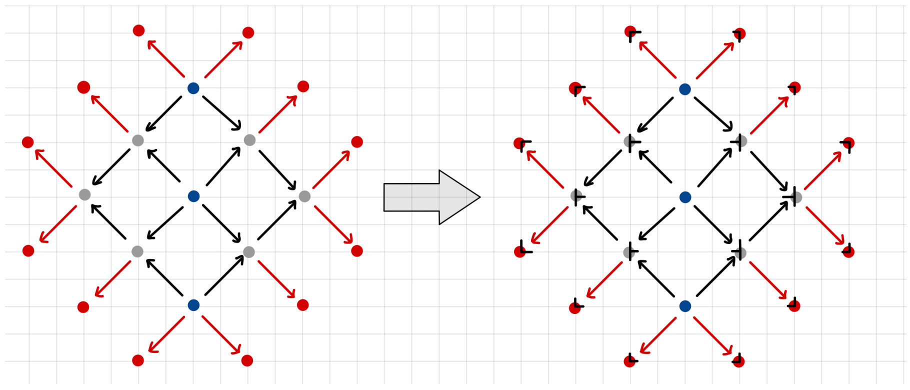
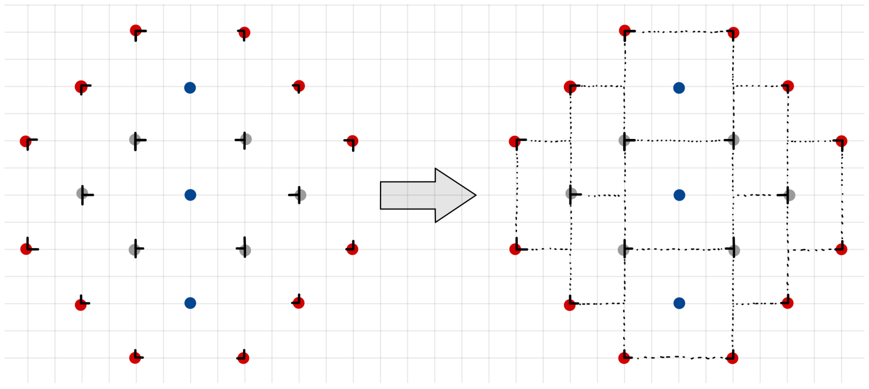


Figure 8: A field of arrow with its corners drawn

We can now draw all the lines (dotted lines on the image below) that lie in between the corners by extending the corners we just drew as follows:



We now claim that each component of A_{n+1} can be:

- a 2×1 rectangle
- a 1×2 rectangle
- a 2×2 square where each square have a repelling node at its center

Let us show that claim.
Let us consider a lattice square S. Any lattice square either have an arrow that exactly is a diagonal whose endpoints are nodes. Then, without loss of generality, let us assume that:

- the lower left corner L and the upper right corner R of S are nodes
- the arrow of the square S goes from R to P.

Since we assumed $\mathcal{F}_{n+1}$ to be an outward pointing field, it follows that the node R is an interior node which is either transient or repelling. Hence, let us discern the two following cases:

1. **R is repelling**. It forces all the arrow to point away from R and hence S is contained in the 2×2 square centered at R. (see figure 9)

2. **R is transient**. There are two possibilities for the arrows described in the figure 10. In both cases, the square S is either contained in a 1×2 or a 2×1 rectangle.

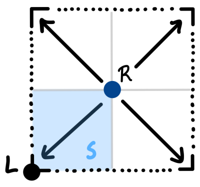


Figure 9: The single scenario in case R is repelling

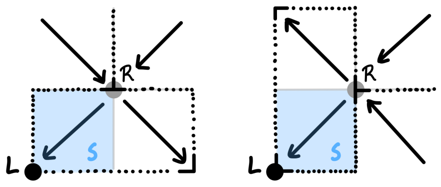


Figure 10: The two scenarios in case R is transient

The claim being proven, it implies that the outward pointing field of arrows $\mathcal{F}_{n+1}$ determines the tilings $\mathcal{T}_{n+1}$ except for the squares. Indeed, each square can either be two 1×2 dominoes or two 2×1 dominoes. Since each square can induce two possible tilings and recalling that each square's center is a repelling node, the number of tilings for a fixed field of arrows $\mathcal{F}_{n+1}$ is thus equal to $2^{r(\mathcal{F}_{n+1})}$. The lemma is proven in the case of an outward pointing field $\mathcal{F}$.
Following the same reasoning, one can prove the former case in which the field $\mathcal{F}_n$ is assumed to be inward pointing. $\square$

Example 1.12. Let us get back to the example 1.9.

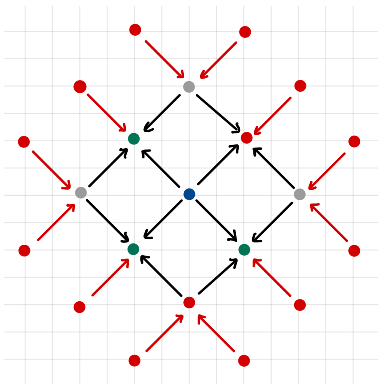


Figure 11: An Inward Pointing Field of Arrows of A_2

By the lemma (1.1), since there are only one repelling node, it follows that there are two tilings associated to that field. Indeed:

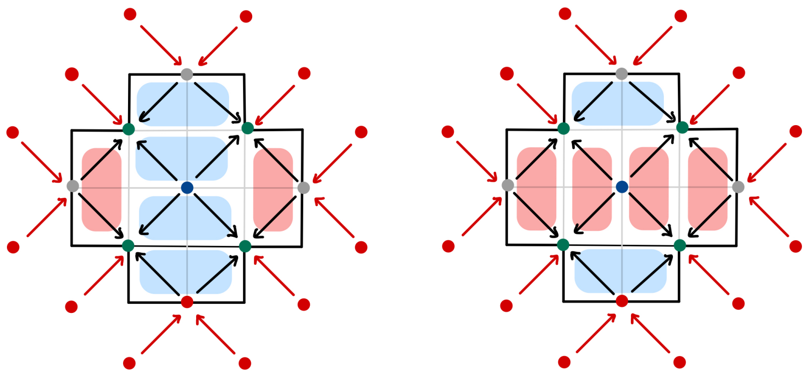


Figure 12: The Two Tilings of A_2 Associated to the Inward Pointing Field of Arrows above

The second result that we will need in order to prove the aztec diamond theorem is the following lemma.

Lemma 1.2. *For any outward pointing field of arrows $\mathcal{F}_{n+1}$ on the aztec diamond A_{n+1}, we have:*

$$r(\mathcal{F}_{n+1}) = a(\mathcal{F}_{n+1}) + (n+1)$$

Proof. First of all, let us point out the two following remarks:

- all the interior nodes of A_{n+1} lie on $n+1$ lines running from bottom left to top right
- we count $n+2$ arrows on each of these lines

In order to prove the lemma, since there are $n+1$ lines, we just need to show that for any line there always is one more repelling node than there are attracting ones.

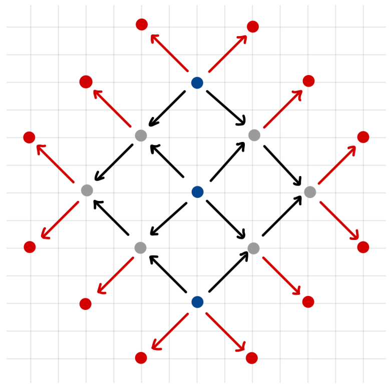


Figure 13: The Outward Pointing Field on A_{2+1}

To each line, we associate a binary sequence of length $n+2$ that describes the directions of the arrows where:

- 1 stands for forward
- 0 stands for backward

Since the field of arrows is supposed to be outward pointing, we know that for each line l, its sequence S_l is of the form: $S_l = \{\underbrace{0, \cdots, 1}_{n+2}\}$.

There are three possible combinations depending on the type of node:

- repelling node: $\{0,1\}$
- attracting node: $\{1,0\}$
- transient node: $\{0,0\}$ or $\{1,1\}$

Hence, since the first value of the sequence is 0 and the last value is 1, there must be one more combination of the form $\{0,1\}$ than there are combinations of the form $\{1,0\}$. Otherwise, it would not be possible to have 1 as the final value.
Therefore, we just showed that on each line there is always one more repelling node than there are attracting nodes. Since there are $n+1$ lines, it follows that $r(\mathcal{F}_{n+1}) = a(\mathcal{F}_{n+1}) + (n+1)$ which proves the lemma. $\square$

We are now ready to prove the aztec diamond theorem!

Proof. Let us prove the theorem 1.1 by induction on n.

- **Base Case:** There are two possible tilings of the aztec diamond of order 1 which are the following:

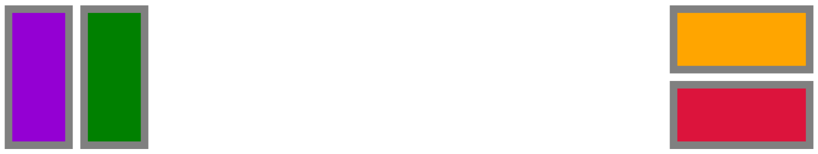


Figure 14: The Two Possible Tilings of A_1

It is straightforward to see that $2 = 2^{1(1+1)/2}$.

- **Induction:** Let $\mathcal{O}_{n+1}$ and $\mathcal{I}_{n+1}$ respectively denote the sets of outward and inward pointing field of arrows on A_{n+1}.

  On the one hand, reversing the direction of all the arrows of A_{n+1} still satisfies the arrow condition. On the other hand, reversing twice in a row the direction of all the arrows is equivalent to doing nothing. Hence, we can define the following bijection:

$$f : \mathcal{O}_{n+1} \longrightarrow \mathcal{I}_{n+1}$$

  For any outward pointing field $\mathcal{F}_{n+1} \in \mathcal{O}_{n+1}$ of A_{n+1}, the attracting nodes of $\mathcal{F}_{n+1}$ clearly are the repelling nodes of $f(\mathcal{F}_{n+1})$. Now, applying lemma 1.2, we have:

$$r(\mathcal{F}_{n+1}) = r\big(f(\mathcal{F}_{n+1})\big) + n + 1$$

  We can rewrite the latter equality as:

$$\begin{aligned} 2^{r(\mathcal{F}_{n+1})} &= 2^{r\left(f(\mathcal{F}_{n+1})\right)+(n+1)} \\ &= 2^{n+1} \cdot 2^{r\left(f(\mathcal{F}_{n+1})\right)} \\ &= 2^{n+1} \cdot 2^{r\left(\mathcal{F}_{n}\right)} \end{aligned}$$

  By the lemma 1.1, we now know that the number of tilings of A_{n+1} is equal to $2^{n+1} \cdot 2^{r(\mathcal{F}_n)}$.

Now, summing over all the inward pointing fields of arrows on A_n and using the induction hypothesis, we finally get:

$$\begin{aligned}
\mathrm{T}_{n+1} &= \sum_{\mathcal{F}\in\mathcal{I}_n} 2^{n+1}\cdot 2^{r(\mathcal{F}_n)} \\
&= 2^{n+1}\sum_{\mathcal{F}\in\mathcal{I}_n} 2^{r(\mathcal{F}_n)} \\
&= 2^{n+1}\cdot \mathrm{T}_n \\
&= 2^{n+1}\cdot 2^{n(n+1)/2} \\
&= 2^{(n+1)(n+2)/2}
\end{aligned}$$

This proves the Aztec diamond theorem. $\square$

We can now use the fact that there are $2^{(n+1)(n+2)/2}$ possible domino tilings of the Aztec diamond $\mathrm{A}_n \; \forall n \in \mathbb{N}^*$.

1.2 Random Tilings as Markov Chains

The idea we wish to implement in order to generate an aztec diamond A_m is to start with a tiling with a default configuration, and then perform a number of rotations to its dominoes. We want the end result of the process to be representative of the uniform distribution: any tiling must have about the same probability to be generated.
To do so, we will construct a Markov chain $((\mathrm{X}_{\mathrm{A}_m})_n)_{n\in\mathbb{N}}$ associated to the random tilings so that $(\mathrm{X}_{\mathrm{A}_m})_n$ converges towards the uniform distribution $\mathcal{U}(2^{m(m+1)/2})$.
Hence, let us define basic notions peculiar to Markov chains.

Definition 1.13. An indexed sequence $\mathrm{X} = (\mathrm{X}_n)_{n\in\mathbb{N}}$ of random variables X_n with state space I is called a discrete-time stochastic process.

Definition 1.14. A discrete-time stochastic process X is a Markov chain if $\forall n \in \mathbb{N}$ and $i_0, \cdots, i_n, i_{n+1} \in I$:

$$\mathbb{P}\left(\mathrm{X}_{n+1} = i_{n+1} \mid \mathrm{X}_0 = i_0, \mathrm{X}_1 = i_1, \cdots, \mathrm{X}_{n-1} = i_{n-1}, \mathrm{X}_n = i_n\right) = \mathbb{P}\left(\mathrm{X}_{n+1} = i_{n+1} \mid \mathrm{X}_n = i_n\right)$$

In other words, it means that the stochastic process does not depend on the past.

When rotating any random tiling of order $m \in \mathbb{N}^*$, the outcome only depends on the last tiling we have. It follows that the discrete-time stochastic process $(\mathrm{X}_{\mathrm{A}_m})_n$ with state $\mathrm{I}_{\mathrm{A}_m} = \{\text{all possible tilings of order } m\}$ is a Markov chain.

Definition 1.15. A Markov chain $\mathrm{X} = (\mathrm{X}_n)_{n\in\mathbb{N}}$ is called time-homogeneous if the transition probabilities $p_{ij}(n) = \mathbb{P}\left(\mathrm{X}_{n+1} = j \mid \mathrm{X}_n = i\right)$ are independent of n. We call $P = (p_{ij})_{i,j\in\mathrm{I}}$ the transition matrix of $(\mathrm{X}_n)_{n\in\mathbb{N}}$.

Then, the Markov chain $(\mathrm{X}_{\mathrm{A}_m})_n$ is also time-homogeneous. Its transition matrix $P_{\mathrm{A}_m} = (p_{T_iT_j})_{T_i,T_j\in\mathrm{I}_{\mathcal{T}}}$ only depends on the number of 2×2 squares that can be rotated. In other words, for any two tilings $\mathcal{T}_i, \mathcal{T}_j$, we get that $p_{\mathcal{T}_i,\mathcal{T}_j} > 0$ if and only if all but (possibly) two dominoes of the tilings are laid the same way.

Example 1.16. Let us define the Markov chain associated to the aztec diamond of order 2 and its graph. For a matter of readability, we rather draw its graph than writing its transition matrix.
By the Aztec diamond theorem, we know that there are $8 = 2^{-2(2+1)/2}$ possible tilings. Hence, $\mathrm{I}_{\mathrm{A}_2} = \{\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_8\}$.
To compute the transition probabilities, we consider all the 2×2 squares inside the aztec diamond (for A_2 there are 5 of them) that we can pick with probability:

$$1/\#\{\text{Number of } 2\times 2 \text{ squares inside the aztec diamond}\}$$

(in our case with probability 1/5).
For instance, let us compute $p_{\mathcal{T}_1\mathcal{T}_j} \forall j \in \mathrm{I}_{\mathrm{A}_2}$. We have:

$$p_{\mathcal{T}_1\mathcal{T}_j} = \begin{cases} 3/5 \text{ if } j = 1 \\ 1/5 \text{ if } j \in \{2,3\} \\ 0 \text{ otherwise} \end{cases}$$

The graph of $\mathrm{X}_{\mathrm{A}_2}$ is then:

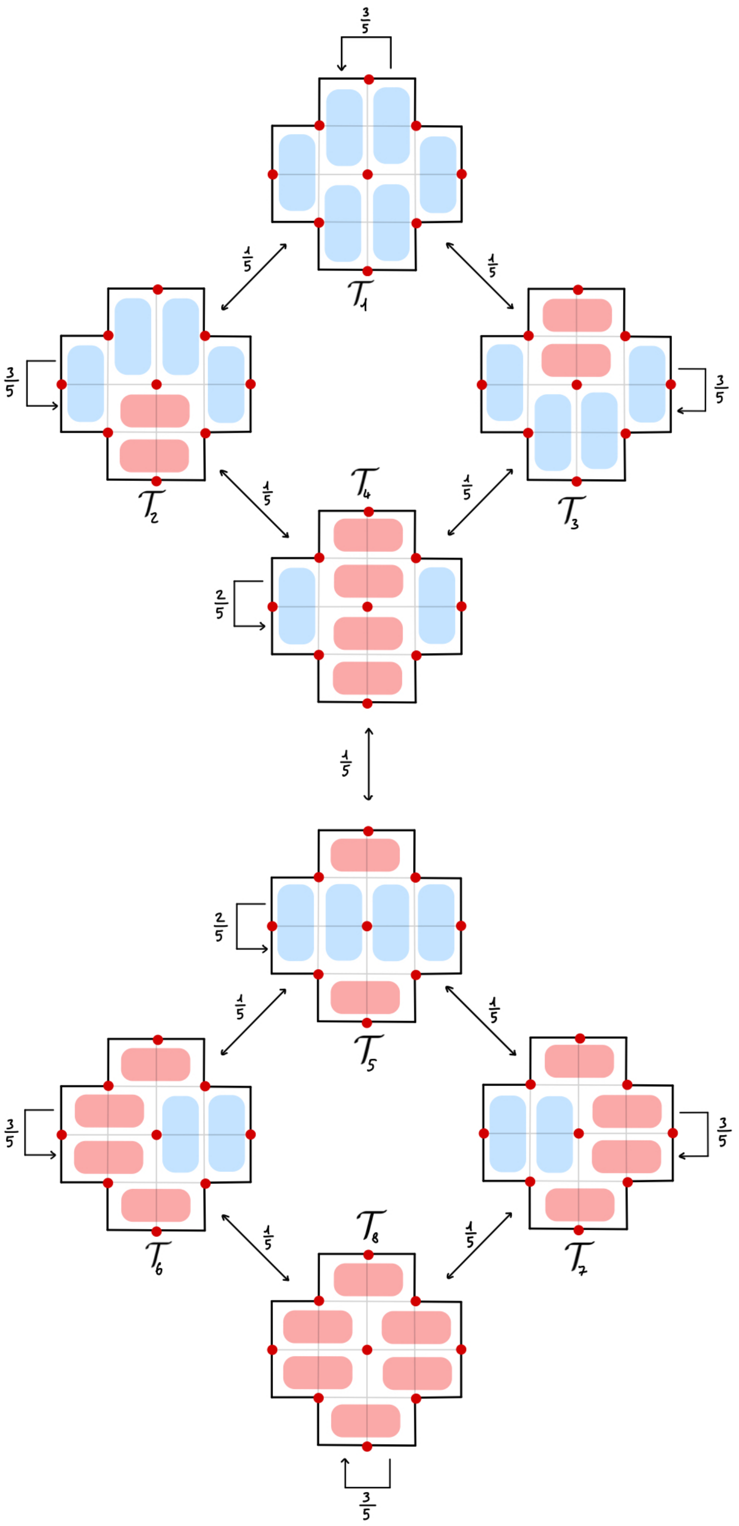


Figure 15: Graph of X_{A_2}

For any order of a aztec diamond of order n and any state (i.e. any possible tiling) $\mathcal{T}_i$, let us denote by S the number of 2×2 squares, and N_i the number of adjacent states, that is any tiling $\mathcal{T}_j \neq \mathcal{T}_i$ such that $\mathcal{T}_j$ can be obtained from $\mathcal{T}_i$ by rotating exactly 2 dominoes. Then we can generalize the transition matrix $P = (P_{\mathcal{T}_i \mathcal{T}_j})_{\mathcal{T}_i, \mathcal{T}_j \in I}$ as follows:

$$p_{\mathcal{T}_i \mathcal{T}_j} = \begin{cases} \frac{1}{S} & \text{if } \mathcal{T}_i \text{ and } \mathcal{T}_j \text{ are adjacent and } \mathcal{T}_i \neq \mathcal{T}_j \\ \frac{S-N_i}{S} & \text{if } \mathcal{T}_i = \mathcal{T}_j \\ 0 & \text{otherwise} \end{cases}$$

We can remark that $\sum_{\mathcal{T}_j \in I} p_{\mathcal{T}_i \mathcal{T}_j} = 1$, which is expected since the sum corresponds to the probability $\mathbb{P}(X_{n+1} \in I \mid X_n = \mathcal{T}_i)$. Additionally, rotations are reversible, so adjacency is a symmetric property and we have $p_{\mathcal{T}_i \mathcal{T}_j} = p_{\mathcal{T}_j \mathcal{T}_i}$. Thus, the number of states adjacent with a state $\mathcal{T}_i$ is also N_i, and so: $\sum_{\mathcal{T}_j \in I} p_{\mathcal{T}_j \mathcal{T}_i} = 1$. Now that we have defined the random tilings of an aztec diamond of order m as a Markov chain, let us define three properties we are interested in.

Definition 1.17. A Markov chain X is said to be:

- **irreducible** if it is possible to go from any state to any other
- **aperiodic** if it is possible to return from any state i to itself after an arbitrary number of steps m, if m is large enough. In particular, there exists a positive integer M_i such that, for all $m \geq M_i$, $\mathbb{P}(X_{n+m} = i \mid X_n = i) > 0$.
- **positive recurrent** if starting from $i \in$ I we are certain to return to that same state i with finite average time

We also need to define invariant probability distributions.

Definition 1.18. Let X be a Markov chain with state space I with transition matrix P. A probability distribution $\pi = (\pi_i)_{i \in I}$ is said to be invariant if $\pi = \pi$. In particular, we have the following equivalence

$$\pi_i = \sum_{j \in I} \pi_j p_{ji}$$

If we are able to show that the Markov chain $(\mathrm{X}_{\mathrm{A}_m})_n$ with state space $\mathrm{I}_{\mathrm{A}_m}$ satisfy these properties for any $m \in \mathbb{N}^*$, we could then use the following result to show that $(\mathrm{X}_{\mathrm{A}_m})_n$converges towards the uniform distribution $\mathcal{U}(2^{n(n+1)/2})$.

Theorem 1.19. *If a Markov Chain* X *is:*

1. *irreducible*
2. *aperiodic*
3. *positive recurrent*

Then there exists a unique invariant probability distribution π on the state space and X_n converges in distribution towards π when $n \to \infty$.

The proof of the irreducibility of the tiling is somewhat techical. It consists in defining a function r which assigns a positive integer to any tiling in a way that if $\mathcal{T}, \mathcal{T}'$ are adjacent, $r(\mathcal{T}) = r(\mathcal{T}') \pm 1$ and such that r admits its unique minimum for the all-horizontal tiling. By showing that it is always possible to perform a rotation in order to decrease the value of the function r up to its minimum, we can demonstrate that it is possible to reach the minimal configuration from any tiling. The complete demonstration of the irreducibility can be found in the references [2].

Since the probability of remaining still $\mathbb{P}(X_{n+1} = i \mid X_n = i)$ is not null for any state, it is clear that the chain is aperiodic. Indeed, it is possible (while unrealistic) to remain in the same state for any arbitrary length of time. Moreover, since the state space is finite and that the chain is irreducible, the chain is also positive recurring.
Let us now remark that the uniform distribution is an invariant probability distribution. Indeed, if $N = 2^{n(n+1)/2}$ denotes the number of possible states for a random tiling of order n, then for any state i we get $\pi_i = \frac{1}{N}$, and:

$$\sum_{j\in I} \pi_j p_{ji} = \frac{1}{N}\sum_{j\in I} p_{ji} = \frac{1}{N} = \pi_i$$

Therefore, the uniform distribution is the unique invariant probability distribution, and X_n converges in distribution towards it.

2 Simulations and Observations

2.1 Technical Implementation of the Random Tilings

We will discuss in this section our implementation of the random tilings. The full implementation can be found in the appendix. We decided to use object oriented programming in Python, for the sake of simplicity and usability.

2.1.1 Creation of an "Organized" Tiling

Let us make a few remarks before presenting the code. A tiling of order n can be seen as a $2n \times 2n$ grid of 1×1 squares.

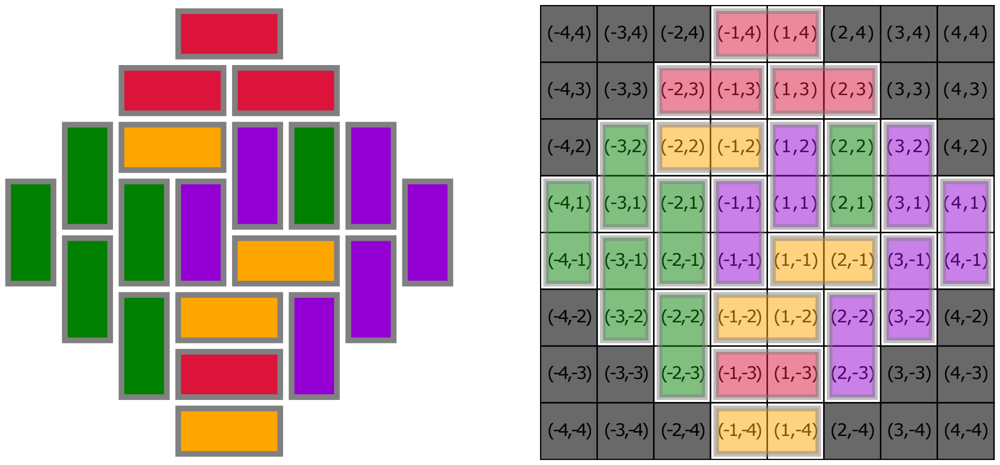


Figure 16: On the left, random tiling of order 4. On the right, representation of the tiling as dominoes laid onto a grid of 1×1 squares

For any tiling $\mathcal{T}$, every domino is placed on two adjacent squares, so that each "active" square (squares that are not greyed out in the above figure) holds exactly one half of a domino. We can encode the domino configuration as an attribute of the squares on which it lays: we say that the square itself is horizontal if the (half) domino placed onto it is oriented horizontally, and vertical otherwise. Let us add an additional bit of information: let the "dominant" part of the domino be the top half when oriented vertically, or the left half when laid horizontally. By adding this data point, the configuration of the domino can be deduced from the information contained in a single square. For example, if we know that a given case holds the dominant part of a domino oriented vertically (i.e. its top half), we know that the domino is placed onto this square and the one right below. By doing so, we do not need to implement the domino themselves as a class.
Let us construct such a grid from the ground up. The first class that we need to create is the `Square` class. The squares are identified by a coordinate system $(\pm x, \pm y)$ such that $x, y \in \mathbb{N}^2, 1 \leq x \leq n, 1 \leq y \leq n$.
Let us note that we do not attribute any coordinate of the form $(0, x)$ or $(y, 0)$. This create some complication when trying to determine its adjacent squares, but it ensures a form of symmetry that will simplify the tiling of an empty grid.
Squares are considered active if they can (and will) hold a domino square. This attribute is computed based on condition (3)

$$|x| + |y| \leq n + 1 \tag{3}$$

The configuration of the domino laid onto the square is also implemented as an attribute of the `square`, as discussed above.

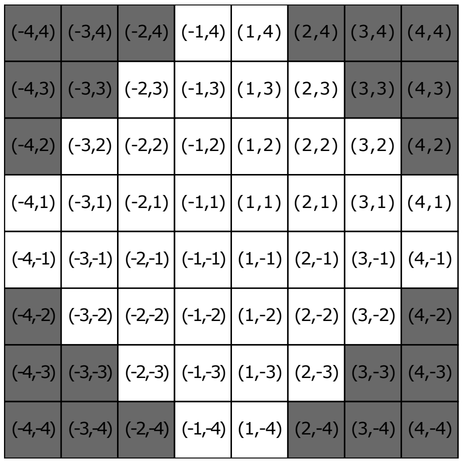
The code for the square class is quite straightforward:

```python
class Square:
    def __init__(self, x, y, n, horizontal, dominant):
        self.x = x # X axis
        self.y = y # Y axis
        self.is_active = (abs(x) + abs(y) <= n +1) # is square in grid ?
        self.horizontal = horizontal # orientation of the domino in grid
        self.dominant = dominant # based on orientation, does grid hold dominant side
            of domino ?
```

Two squares of coordinates (x, y) and (x', y') are adjacent in the following cases

$$\begin{cases} x = x', y = y' \pm 1 & \text{if } y \notin \{-1, 1\} \\ x = x', y = y' + 2 & \text{if } y = -1 \\ x = x', y = y' - 2 & \text{if } y = 1 \\ y = y', x = x' \pm 1 & \text{if } x \notin \{-1, 1\} \\ y = y', x = x' + 2 & \text{if } x = -1 \\ y = y', x = x' - 2 & \text{if } x = 1 \end{cases}$$

Therefore, the creation of every square of coordinates $(\pm x, \pm y)$ for $x, y \in \mathbb{N}$ such that $1 \leq x \leq n, 1 \leq y \leq n$ is equivalent to generating an empty grid.



(-4,4)	(-3,4)	(-2,4)	(-1,4)	(1,4)	(2,4)	(3,4)	(4,4)
(-4,3)	(-3,3)	(-2,3)	(-1,3)	(1,3)	(2,3)	(3,3)	(4,3)
(-4,2)	(-3,2)	(-2,2)	(-1,2)	(1,2)	(2,2)	(3,2)	(4,2)
(-4,1)	(-3,1)	(-2,1)	(-1,1)	(1,1)	(2,1)	(3,1)	(4,1)
(-4,-1)	(-3,-1)	(-2,-1)	(-1,-1)	(1,-1)	(2,-1)	(3,-1)	(4,-1)
(-4,-2)	(-3,-2)	(-2,-2)	(-1,-2)	(1,-2)	(2,-2)	(3,-2)	(4,-2)
(-4,-3)	(-3,-3)	(-2,-3)	(-1,-3)	(1,-3)	(2,-3)	(3,-3)	(4,-3)
(-4,-4)	(-3,-4)	(-2,-4)	(-1,-4)	(1,-4)	(2,-4)	(3,-4)	(4,-4)

Figure 17: An empty grid of order 4. Grey squares are deactivated and cannot hold dominoes.

Before discussing the implementation of the grid, we shall introduce the concept of rotation boxes. Two dominoes can be rotated only if they are placed onto a 2×2 box, which is precisely four active squares sharing a corner. Let us call such a structure a `Rotation_Box`
The following condition allows us to verify if a rotation is possible on a box, based on the attributes of its squares.

Proposition 2.1. *It is possible to perform a rotation on a 2×2 box if, and only if:*

1. *The top left square and the bottom right square of the box have the same orientation.*
2. *The top left square is dominant and the bottom right square is nondominant.*

Proof. It is clear that if the top left and bottom right square do not share the same orientation, it is not possible to rotate the box.

Assume that the square can be rotated. If both squares are oriented horizontally, the top left square must hold the left side of a domino, and the bottom right square must hold the right side of another domino. If both dominoes are oriented vertically, the upper left square must hold the top side of the first domino and the bottom right square must hold the bottom half of the second domino. In both cases, the dominance condition is respected. Moreover, this configuration requires that both the top right and bottom left squares share the same orientation as the top left and bottom right.
Now, assume that both square share the same orientation but the dominance condition 2. is not respected. If they are oriented vertically, we can assume that the top left square is nondominant. Then it holds that the bottom side of a domino, and the top side will be one the square located above it, which is outside the box. It remains 3 squares for a single domino, which contradicts the fact that the box holds exactly two dominoes, and thus rotation is not possible. The same argument holds if the bottom right corner does not respect the dominance condition, but also in the horizontal case since, if the top left square is nondominant, it holds the left side of a domino, whose left half will be placed outside the box. □

We can perform a rotation within the box as follows:

1. We swap the orientation of every square in the box. Horizontal becomes vertical, vertical becomes horizontal.

2. We swap the dominance attribute of the top right and bottom left squares. Dominant becomes non-dominant, non-dominant becomes dominant.

Indeed, it is required to update the orientation of all the squares in the box. Since the rotation can be reverted, the dominance attributes of the top left and bottom right squares must remain the same. However, if the top right corner was holding the right side of a vertical domino, it will hold after rotation the top side of a vertical domino. Therefore its dominance only needs to be swapped, and for the same reason, so does the dominance attribute of the bottom left square.
We can therefore implement an object called `Rotation_Box` which consists in a 2×2 box. Its attributes will be the four squares it is composed of, a boolean variable which will confirm if the box can or cannot be rotated, and the functions described above to verify the possibility of a rotation, and to perform the rotation within the box. . The `Rotation_Box` objects will also be given coordinates based on the coordinate of their top left square.

Based on the information discussed abouve, we implemented the `Rotation_Box` class as:

```python
class Rotation_Box:
    def check_rotatability(self): # resets can_be_rotated value
        self.can_be_rotated = ( # can square be rotated ?
            (self.UL.horizontal == self.BR.horizontal) and
            (self.UL.dominant) and
            (not self.BR.dominant) and
            self.is_active
            )
        return self.can_be_rotated

    def __init__(self, UL, UR, BR, BL):
        self.UL = UL #upper left square
        self.UR = UR #upper right square
        self.BR = BR #bottom right square
        self.BL = BL #bottom left square
        self.Squares = {self.UL,self.UR,self.BR,self.BL} # Set containing all squres
            of box
        self.can_be_rotated = self.check_rotatability()
        self.is_active = all([C.is_active for C in self.Squares])

    def rotate(self): # updates orientation and dominance
        if self.can_be_rotated:
            ns = not self.UL.horizontal
            self.UL.horizontal = self.UR.horizontal = self.BR.horizontal = self.BL.
                horizontal = ns
            self.UR.dominant = not self.UR.dominant
            self.BL.dominant = not self.BL.dominant
            return True
        else:
            return False
```

We can now discuss the implementation of the `Grid` object. First of all, an object of this class will contain the following informations:

- a collection of `Square` objects with coordinates $(\pm x, \pm y), x, y \in \mathbb{N}, 1 \leq x \leq n, 1 \leq y \leq n$.
- a collection of `Rotation_Box` objects which correspond to all 2×2 boxes.

We will design the class `Grid` so that upon creation, all the objects of class `Square` and `Rotation_Box` will be generated. This requires to compute the orientation and the dominance for all squares in the grid. As we want all dominoes to be oriented horizontally in the initial configuration, we simply set the orientation of each active square to `Horizontal = True`.
Let us focus on the top left quadrant of the grid with coordinates $(-x, y)$ to identify a pattern which will allow us to compute the dominance. The top-most square located in $(-1, n)$, always needs to be dominant. In particular, we observe the following relation: the square is dominant if $x + y - n$ is odd and so, regardless of the parity of n.
We observe that this relation is reversed on the top right quadrant (squares of coordinates (x, y).) Therefore, on the top half of the grid, the dominance can be computed as follows:

$$f(x,y) = \begin{cases} 1 & \text{if } x + y - n \equiv 1 \ [2] \text{ and } x < 0 \\ 1 & \text{if } x + y - n \equiv 0 \ [2] \text{ and } x > 0 \\ -1 & \text{otherwise} \end{cases}$$

This function was implemented as follows:

```python
def f(x,y,n): # Defines the logic for dominance attribute
    if (x + y - n) % 2:
        return - sign(x)
    return sign(x)
```

This dominance of the square of coordinates (x, y) can therefore be computed on the upper half of the grid by evaluating $f(x, y)$: it is dominant if $f(x, y) = 1$. We can then use the use the symmetry along the central vertical line to show that the same logic is valid for squares of coordinates $(\pm x, -y)$.

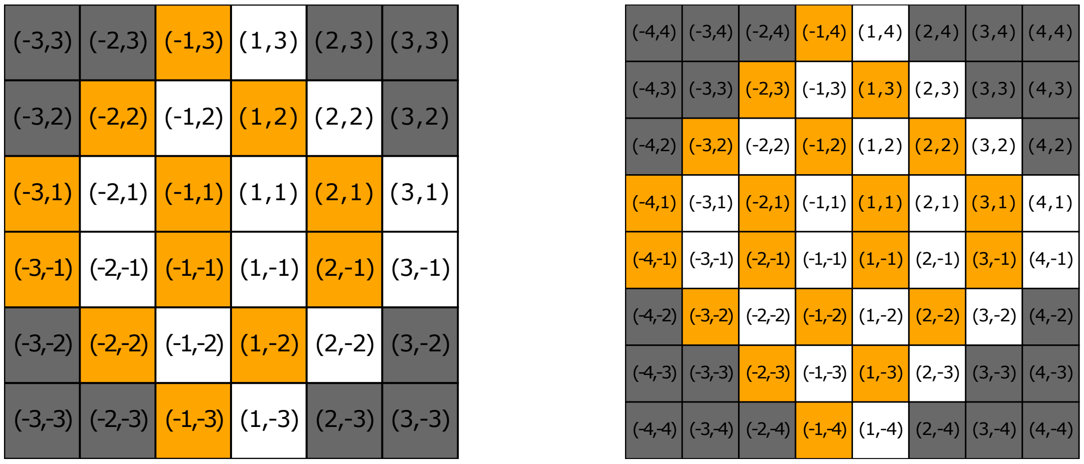


Figure 18: Computation of dominant squres in odd (here $n = 3$, on the left) and even ($n = 4$, on the right) ordered tilings. The orange squares are computed as dominant based on the function defined above.

We can now implement the class `Grid`. When creating an object of this class, it will create all `square` and `Rotation_box` objects, and store them in dictionaries. Additionally, it also creates a set containing all boxes for which a rotation is possible.

```python
class Grid:
    def __init__(self, n:int, m:int):
        self.n = n
        self.m = m
        self.Squares = {} # dictionary of grid’s tiles
        self.Rotation_Boxes = {} # dictionary of grid’s squares
        self.Turnable_Boxes = set() # set of all rotatable squares

        for j in range(-m,m+1): # Generates the squares
            if j==0: # tiles with null y axis are not defined
                continue
            for i in range(-n,n+1):
                if i == 0: # tiles with null x axis are not defined
                    continue
                self.Squares[i,j] = Square(i,j,n, True, (f(i,j,n) == 1)) # populate
                    Tiles dictionary

        for j in range(n,-n,-1): #Generates the boxes
            if j == 0: #squares with null y axis are not defined
                continue
            for i in range(-m, m):
                if i==0: #squares with null x axis are not defined
                    continue
                self.Rotation_Boxes[i,j] = Rotation_Box( # populate Squares dictionary
                self.Squares[i,j], # upper left tile
                self.Squares[i+1+(i==-1), j], # upper right tile
                self.Squares[i+1+(i==-1), j-1-(j==1)], # bottom right tile
                self.Squares[i, j-1-(j==1)] # bottom left tile
                )
                if (self.Rotation_Boxes[i,j].can_be_rotated):
                    self.Turnable_Boxes.add(self.Rotation_Boxes[i,j]) # populates
                        rotatable set
```

2.1.2 Representation of the Grid

Let us consider the following matrix $(M_{i,j})$ where $i, j \in \{-n, -n+1, \dots, -2, -1, 1, 2 \dots, m-1, n\}$ are the coordinates such that (i, j) is a square on the grid:

$$M_{i,j} = \begin{cases} 2 & \text{if the square } (i,j) \text{ is vertical dominant.} \\ 1 & \text{if the square } (i,j) \text{ is horizontal dominant.} \\ 0 & \text{if the square } (i,j) \text{ is inactive.} \\ -1 & \text{if the square } (i,j) \text{ is horizontal non-dominant.} \\ -2 & \text{if the square } (i,j) \text{ is vertical non dominant.} \end{cases}$$

Any tiling can be uniquely represented as such a $2n \times 2n$ matrix. We can save this matrix as a comma separated value (csv) file for further manipulation and graphic representation.

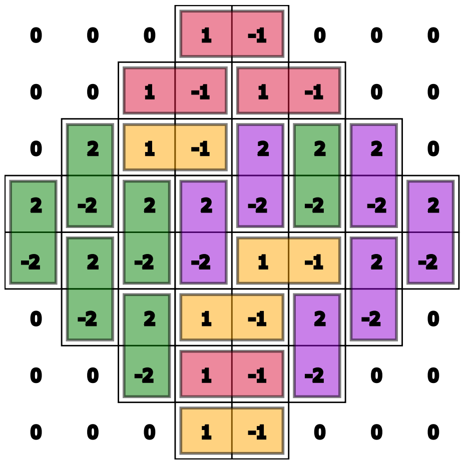


Figure 19: Matrix representation of a tiling

We can use the following snippet to encode the orientation and dominance of a square. Considering that the orientation and dominance are stored as a boolean `horizontal`, `dominant` $\in$ {True, False}.

```python
def render(C:Square): # Defines value when generating CSV
    return C.is_active * (C.dominant*1 - (not C.dominant)*1) * (1+ (not C.horizontal)
        *1)
```

We can now add the function `save` within the class `Grid` to save the `Grid` as a comma seperated value (csv) document. The paramater `path` is used to indicate the location and name of the csv file.

```python
def save(self,dest):
    """ stores the grid as as CSV file """
    f = open(dest,'w')
    for j in range(self.m,-self.m-1,-1):
        if j == 0:
            continue
        for i in range(-self.n, self.n+1):
            if i == 0:
                continue
            f.write(str(render(self.Squares[i,j])))
            if i < self.n:
                f.write(",")
        f.write('\n')
    f.close()
```

From this csv file, we can create a visual representation of the tilings using scalar vector graphics (svg). For each dominant square in the tiling (i.e. coefficients of the matrix $M_{i,j}$ which are either 1 or 2), a line is added to a svg file, with the following pattern:

```
<rect x="482" y="2" width="36" height="16" fill="orange"/>
```

Let us detail how the values are calculated. Allow some padding around tiling and between each domino of a variable `p`, and condiser the width of a square `Sqsize` we loop through each value of the csv document. Let i, j be value in i-th column of the j-th line.

1. `x = p + i * Sqsize`
2. `y = p + j * Sqsize`

3. The `width` and `height` are calculated as `Sqsize*k -2*spacing`, with the value of `k` depending on the orientation of the domino: if a domino is oriented horizontally, $k = 2$ for the width and $k = 1$ for the height, and inversely for vertically oriented dominoes.

4. The color attribute `fill` is computed based on the position of the dominant half of the domino, and its orientation. For example, horizontal dominoes will either be `orange` or `crimson` if the domino is oriented horizontally. The choice between the two is computed as the sum of the coordinates of its dominant domino, modulo 2.

2.1.3 Rotations

For the moment we can perform the rotation of dominoes on a single `Rotation_Box`. Moreover, we introduced a set `Turnable_Boxes` containing all the `Rotation_Box` for which a rotation is possible. The process now consists in:

1. Pick at random one of the `Rotation_Box` from the `Turnable_Boxes` set.
2. Perform a rotation on this box.
3. Update the `Turnable_Boxes` set to add `Rotation_Box` objects that could not be rotaded previously, but now; or remove the ones that could be rotated but for which a rotation is no longer possible.

We can demonstrate that, after a rotation, it suffices to verify if a rotation is possible on the `Rotation_boxes` located directly above, below, to the left and to the right of it.

Figure 20: Rotation Boxes update

The following functions were added to the `Grid` class, which allow for a deterministic number of rotations.

```python
def check_if_turnable(self,i:int,j:int): # after rotation, checks if adjacent squares can be rotated
    """ After rotation, checks if adjacent squares can be rotated """
    tile = self.Rotation_Boxes.get((i,j)) # prevents out of bound.
    if tile:
        if tile.check_rotatability(): # adds square to rotatable set if possible
            self.Turnable_Boxes.add(tile)
        else:
            if tile in self.Turnable_Boxes: # remove square from rotatable set if not possible
                self.Turnable_Boxes.remove(tile)

def rotate(self, i:int ,j:int):
""" Performs rotation on square with top left square in (i,j), then updates rotatability of adjacent squares. """
    n = self.n; m = self.m
    if (self.Rotation_Boxes[i,j].rotate()): # attempts rotation and updates rotatable set
        # print("Rotation of square (",i,j,") completed.")
        self.check_if_turnable(i+1+(i==-1),j)    # right box
        self.check_if_turnable(i-1-(i==1),j)     # left box
        self.check_if_turnable(i,j+1+(j==-1))    # above box
        self.check_if_turnable(i,j-1-(j==1))     # below box
    else:
        print("Rotation of square (",i,j,") is not possible")

def randomize(self,n:int):
    for i in range(1,n+1):
        Sq = sample(self.Turnable_Boxes,1)[0]
        x = Sq.UL.x
        y = Sq.UL.y
        self.rotate(x,y)
```

Let us note that the implementation does not technically correspond to an aperiodic Markov chain, since we do not allow the tilings to stay constant when calling the function `randomize`. Each call will result in (at least) one rotation. It is however possible implement the aperiodicity, without causing loss in program runtime. In order to do so, we introduce two new attributes for the `Grid` object:

- the `actual_rotation` attribute stores the number of times a rotation was actually completed. We increment this attribute by 1 each time the `rotate` function is called.
- the `rotation` stores a virtual number of attempted rotations, which correspond to the time n of the Markov chain X. After each rotation, this attribute is incremented by G where G is a random variable following a geometric distribution $\mathcal{G}(p)$, where p is the probability of changing state. In particular, $G = k > 1$ is equivalent to staying put $k-1$ times, then moving to a different state.

This alternative version of the randomize function was implemented:

```
def randomize_limit(self,s:int):
    while self.rotations < s:
        self.rotations += random.geometric(p=(len(self.Turnable_Boxes))/self.SqLen)
        self.actual_Rotations += 1
        Sq = sample(self.Turnable_Boxes,1)[0]
        x = Sq.UL.x
        y = Sq.UL.y
        self.rotate(x,y)
    return self.breakingPoint()
```

We can therefore generate a random tiling of order n using the command.

```
n = 20 #order of the tilings
r = 1000000 # We want X_r and attempt r=1,000,000 rotations.
from RTClass3 import *
K = Grid(n,n)
K.randomize_limit(r)
K.save('data/Tilings_001.csv')
```

The following command line in the terminal will then convert the csv into a svg:

```
    python code/SVGwriter3.py [LOCATION_OF_CSV] [LOCATION_OF_SVG]
```

2.2 Some observations

We qualify a domino of well-tiled with respect to one of the corners of the aztec diamond if that domino could be a part of a “brick wall” fashionned tiling. By “brick wall” fashionned, we mean that each corner of the tiling have the same orientation the upper and lower polar regions are both horizontally tiled whereas the left and right polar regions are vertically tiled. Red dominoes in figure 3 are well tiled with regards to the top, while orange dominoes are well tiled with regards to the bottom.
The dominoes laid on a case of the top left border (that is the ones with coordinates $(-n+1-i, i)$ for $i \in \{0, n\}$) are either the right side of an horizontal domino, or the top side of a vertical domino.

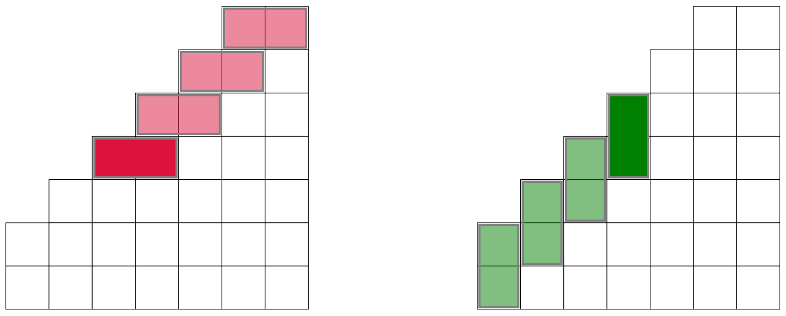


Figure 21: Possible position for a domino on the first diagonal border. The transparent dominoes are necessary in the sense that no other configuration is possible.

The representation in figure 21 shows that either of those configurations has a repercution on the rest of the border: if the domino is laid horizontally (i.e. well tiled with regards to the top), then all dominoes on the first diagonal border will also be well tiled from the top down to the domino. Conversely, if it is laid vertically (i.e. well tiled with regards to the left base), the same disposition can be observed for all leftward dominoes.
We can conclude that we have a limited number of possible cases. The first one is that all the dominoes on the first diagonal border have the same disposition (either all vertical and well tiled with regards to the left, or all horizontal and well tiled with regards to the top).
The second possibility is that they do not have the same disposition. In this case, the top and left side are both well tiled with regards to their own base, and there exist a unique point in which the diagonal and vertical dominoes meet. Such a point must be one of the cases displayed in figure 22.

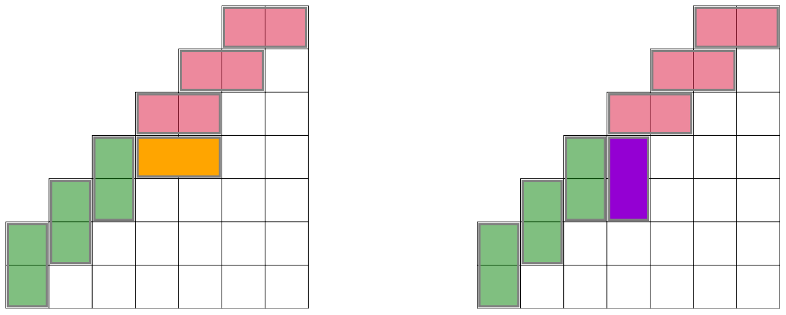


Figure 22: Point at which vertical and horizonal dominoes meet on the first diagonal border.

We can observe from this figure that dominoes laid on this point cannot be well tiled with regards to the top, nor to the left. They therefore create a first point at which the dominoes starts to deorganize. As stated previously, if it exists, such a point is unique on each border.
The existence of this point is not guaranteed. However, we can compute the probability that this point does not exist, which we shall do later.

Proposition 2.2. *The probability that the top corner is not well tiled is* 2^{-n}.

Proof. Let us assume that the top corner is not well tiled. This implies that the dominoes laid on the topmost cases of the tilings are set vertically. Then all the dominoes on the top left and top right dominoes must be set vertically, down to the left and right corners.

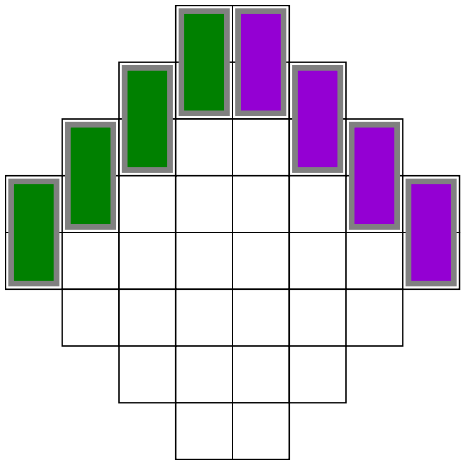


Figure 23: The top corner is not well tiled, thus the dominoes on the top borders are all laid vertically.

The remaining free cases compose a grid for a tilings of order $n-1$. This is the unique possibility for a tilings of order n. Therefore we have:

$$\begin{aligned}
\mathbb{P}\left\{\text{the top corner is not well tiled}\right\} = \frac{2^{(n-1)n/2}}{2^{n(n+1)/2}} &= 2^{((n-1)n-n(n+1))/2} \\
&= 2^{n(n-1-n-1)/2} \\
&= 2^{-2n/2} \\
&= 2^{-n}
\end{aligned}$$

□

Proposition 2.3. *The probability that neither the top, nor the bottom corners are well tiled is* 2^{-2n+1}.

Proof. Let us assume that the top corner is not well tiled. By symmetry, the probability that the bottom corner is also not well tiled is the same as the probability that the top corner is not well tiled in a $n-1$ order tilings, that is:

$$\mathbb{P}\left\{\text{Neither top nor bottom corners are well tiled}\right\} = 2^{-n} \cdot 2^{-(n-1)} = 2^{-2n+1}$$

□

3 Mixing time

Our first attempt to generate a random sample of large order was not satisfactory. A hundred million rotations were not sufficient to deconstruct the organized tiling, and its polar circle was flatter than expected.

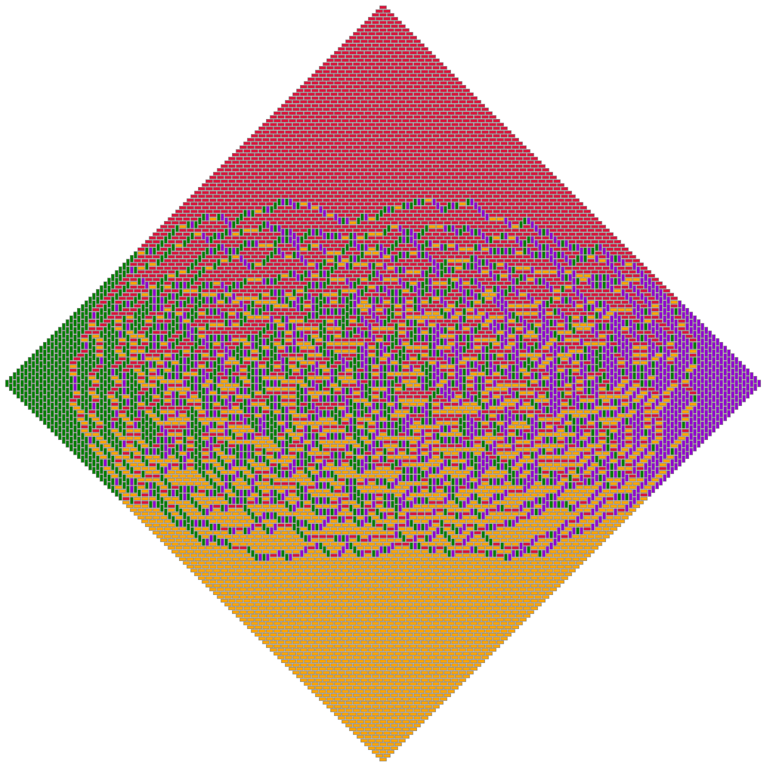


Figure 24: Tiling of order 100, generated after 100 000 000 rotations

Clearly, we needed more than a hundred million rotations for a tiling of order 100. This raised the following problem: how many rotations would be needed to generate a random tiling more representative of the uniform distribution? Let us start with the following observation: we know that most of the possible configurations will have a unique point on each border at which horizontal and vertical dominoes will revert. For a random tiling $\mathcal{T}$, let us denote the height (i.e. the y coordinate) of this point (we will refer to this point as the "breaking point") on the top left border by $H_{TL}(\mathcal{T}) \in \{0, n\}$ where n is the order of the tiling.
Suppose that $H_{TL}(\mathcal{T}) = k$. By symmetry along the diagonal axis $y = -x$, we can construct another tiling $\mathcal{T}'$ whose height will be $H_{TL}(\mathcal{T}') = n - k$. We conclude that for any tiling of height k, there exists a symmetric tiling of height $n - k$: the average height of a random tilings must therefore be on the center of the border: $\mathbb{E}\left[H_{TL}(\mathcal{T})\right] = \frac{n}{2}$.
We need to perform a large enough number of rotations so that the average height of this point will will lay at half the order. Let us find a lower bound on the number of rotations to meet this requirement. We will start by implementing a function which computes the height of the breaking point. The following snippet was included within the `Grid` class definition:

```python
def breakingPoint(self):
    for i in range(1,self.n+1):
        if (not self.Squares[-i,self.n+1-i].horizontal):
            return abs(i-1)
    return self.n
```

The function loops through all the the squares of the top left border and returns the y coordinate of the first non-horizontal square. We note that if the first square is vertical, the function returns n that if all squares of the border are horizontal, the function is set to return n.
Using this function, let us perform the following experiment:

1. We generate m organized tiling of order n, $\mathcal{T}_1^{(0)}, \ldots, \mathcal{T}_m^{(0)}$.

2. We perform k rotations on each tiling, and measure the average height of the breaking point and get $\mathcal{T}_1^{(k)}, \dots, \mathcal{T}_{100}^{(k)}$. Let us for example introduce the function $H_n : \mathbb{N} \to \mathbb{R}_+$

$$H_n(k) = \frac{1}{m} \sum_{i=1}^{m} H_{TL}\left(\mathcal{T}_i^{(k)}\right)$$

The following figure shows the evolution of $H_n(k)$ for $n \in \{10, 15, 20, 25\}$ and $k \leq 300000$.

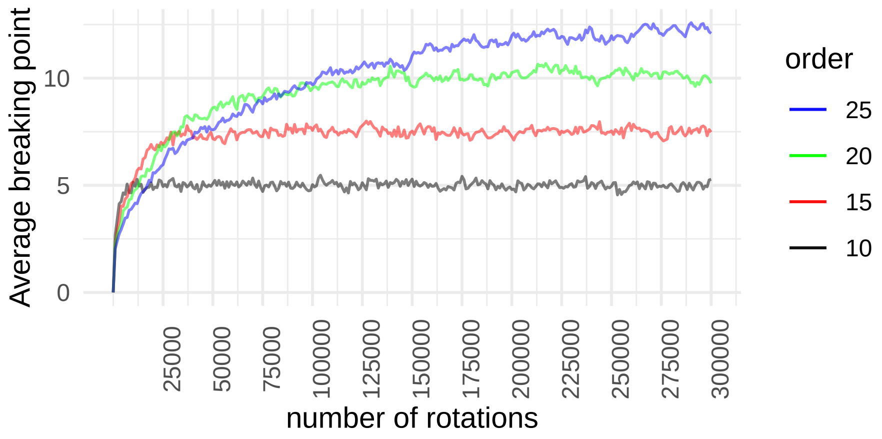


Figure 25: Average height of breaking point for a random tilings of order 10, 100 tilings

We can observe that the average height of the breaking point stabilized around 5 for the tilings of order $n = 10$, and so very early. Around 10,000 rotations were sufficient to obtain an observed average close to the expected value. For tilings of order 20, it took over 10 times this number of rotations to reach $H_{20} \approx 10 = \frac{20}{2}$. We performed the same processus for tilings of larger orders (up to 45). The following table shows, for our simulations, the number of actual rotation it took for the average height to stabilize at $\frac{n}{2}$.

order	number of rotations
10	10,000
15	50,000
20	100,000
25	220,000
30	450,000
35	750,000
40	150,000
45	250,000

Let us plot this data.

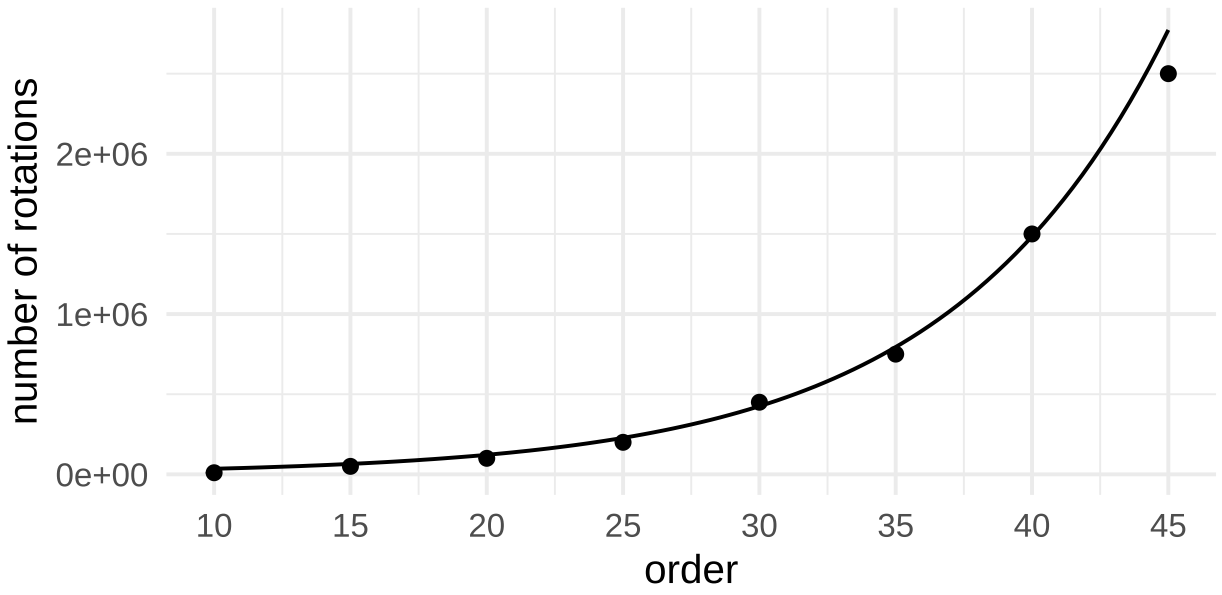


Figure 26: Number of actual rotations needed to reach an average breaking point height of $\frac{n}{2}$
.

Based on the shape of the graph, it appears that the minimum number or rotations follows an exponential function. The line in the above figure above the graph of the function $x \mapsto 10^5 \cdot \exp\left(\frac{x}{8}\right)$.
Returnining to our initial problem, if we assume that the minimal number of rotations follows $10^5 \cdot \exp\left(\frac{\text{order}}{8}\right)$, it would require over 2.5 billion actual rotations in order to obtain a representing tiling of order 100. Let us emphasize that this is the number of actual rotations on a non-aperiodic Markov Chain. The number of the aperiodic Markov chain equivalent must be even larger. Indeed, we need the state of the chain in X_n for which n is large enough so that the actual number rotations $k \leq n$ exceeds 2.5 billion, which is unrealistic.
Finally, let us review the process to generate a random tiling following the aperiodic Markov Chain. We can decide on a large number of rotations r so that the generated tilings represents X_n for a $n >> r$:

1. We start with $X_0 = \mathcal{T}_0$, the all-horizontal tiling.

2. We start the random process defined for the aperiodic Markov Chain $X_1, X_2, \ldots$, allowing the chain to remain on the same state with some probability (as discussed in section 1.2). We ensure that the number of actual rotations exceeds the lower bound defined above. Let us denote as T the first time that this condition is met.

3. We then continue with r additional steps, and stop the process as soon as the time exceeds $T + r$. The end result will be the state of X_n with $n \geq T + r$.

The process defined above ensures that a sufficient number of rotations will be performed to satisfy an expected height of the breaking point equivalent to expectation of the uniform distribution, while maintaining the aperiodicity of the Markov chain. This last point is important since it is a necessary condition for the limit distribution to be the uniform distribution.

Appendixes

A Code

```python
# import numpy as np
from numpy import sign, random
from random import sample

def dom(x,y,n): # Defines the logic for is_active status
    if (x+ y - n) %2:
        return - sign(x)
    return sign(x)

def render(C): # Defines when value when generating CSV
    return C.is_active * (C.dominant*1 - (not C.dominant)*1) * (1+ (not C.horizontal)
        *1)

class Square:
    def __init__(self, x, y, n, horizontal, dominant):
        self.x = x # X axis
        self.y = y # Y axis
        self.is_active = (abs(x) + abs(y) <= n +1) # is square in grid ?
        self.horizontal = horizontal # orientation of the domino in grid
        self.dominant = dominant # based on orientation, does grid hold dominant side
            of domino ?

class Rotation_Box:
    def check_rotatability(self): # resets can_be_rotated value
        self.can_be_rotated = ( # can square be rotated ?
            (self.UL.horizontal == self.BR.horizontal) and #might not be needed
            (self.UL.dominant) and
            (not self.BR.dominant) and
            all([C.is_active for C in self.Squares])
            )
        return self.can_be_rotated

    def __init__(self, UL, UR, BR, BL):
        self.UL = UL
        self.UR = UR
        self.BR = BR
        self.BL = BL
        self.Squares = {self.UL,self.UR,self.BR,self.BL} # Set containing all squares
            of box
        self.can_be_rotated = self.check_rotatability()
        self.is_active = all([C.is_active for C in self.Squares])

    def rotate(self): # updates orientation and dominance
        if self.can_be_rotated:
            ns = not self.UL.horizontal
            self.UL.horizontal = self.UR.horizontal = self.BR.horizontal = self
                .BL.horizontal = ns
            self.UR.dominant = not self.UR.dominant
            self.BL.dominant = not self.BL.dominant
            return True
        else:
            return False

class Grid:

    def __init__(self, n:int, m:int):
        self.n = n
        self.m = m
        self.rotations = 0
        self.actual_Rotations = 0
        self.Squares = {} # dictionary of grid's tiles
        self.Rotation_Boxes = {} # dictionary of grid's squares
        self.Turnable_Boxes = set() # set of all rotatable squares

        for j in range(-m,m+1): # Generates tiles in grid
            if j==0: # tiles with null y axis are not defined
                continue
            for i in range(-n,n+1):
                if i == 0: # tiles with null x axis are not defined
                    continue
                self.Squares[i,j] = Square(i,j,n, True, (dom(i,j,n) == 1)) # populate
                    Tiles dictionary

        for j in range(n,-n,-1): #Generates the squares
            if j == 0: #squares with null y axis are not defined
                continue
            for i in range(-m, m):
                if i==0: #squares with null x axis are not defined
                    continue
                self.Rotation_Boxes[i,j] = Rotation_Box( # populate Squares dictionary
                    self.Squares[i,j], # upper left tile
                    self.Squares[i+1+(i==-1), j], # upper right tile
                    self.Squares[i+1+(i==-1), j-1-(j==1)], # bottom right tile
                    self.Squares[i, j-1-(j==1)] # bottom left tile
                )
                if (self.Rotation_Boxes[i,j].can_be_rotated): self.Turnable_Boxes.add(
                    self.Rotation_Boxes[i,j]) # populates rotatable set

            self.SqLen = sum([1 for k,v in self.Rotation_Boxes.items() if v.is_active
                ])

    def check_if_turnable(self,i:int,j:int): # after rotation, checks if adjacent
        squares can be rotated
        """ After rotation, checks if adjacent squares can be rotated """
        tile = self.Rotation_Boxes.get((i,j)) # prevents out of bound.
        if tile:
            if tile.check_rotatability(): # adds square to rotatable set if possible
                self.Turnable_Boxes.add(tile)
            else:
                if tile in self.Turnable_Boxes: # remove square from rotatable set if
                    not possible
                    self.Turnable_Boxes.remove(tile)

    def rotate(self, i:int ,j:int):
        """ Performs rotation on square with top left square in (i,j), then updates
            rotatability of adjacent squares. """
        n = self.n; m = self.m
```

```python
        if (self.Rotation_Boxes[i,j].rotate()): # attempts rotation and updates rotatable set
            # print("Rotation of square (",i,j,") completed.")
            self.check_if_turnable(i+1+(i==-1),j)   # right square
            self.check_if_turnable(i-1-(i==1),j)    # left square
            self.check_if_turnable(i,j+1+(j==-1))   # above square
            self.check_if_turnable(i,j-1-(j==1))    # below square
        else:
            print("Rotation of square (",i,j,") is not possible")

    def display(self):
        """ Print the grid in the terminal """
        for j in range(self.m,-self.m-1,-1):
            if j == 0:
                continue
            # print([(Tiles[i,j].x,Tiles[i,j].y) for i in range(-self.n, self.n+1) if i != 0])
            print([render(self.Squares[i,j]) for i in range(-self.n, self.n+1) if i != 0])

    def save(self,dest):
        """ stores the grid as as CSV file """
        f = open(dest,'w')
        for j in range(self.m,-self.m-1,-1):
            if j == 0:
                continue
            for i in range(-self.n, self.n+1):
                if i == 0:
                    continue
                f.write(str(render(self.Squares[i,j])))
                if i < self.n:
                    f.write(",")
            f.write('\n')
        f.close()

    def randomize(self,n:int):
        # SqLen = len(self.Rotation_Boxes)
        for i in range(1,n+1):
            self.rotations += random.geometric(p=(len(self.Turnable_Boxes))/self.SqLen)
            self.actual_Rotations += 1
            Sq = sample(self.Turnable_Boxes,1)[0]
            x = Sq.UL.x
            y = Sq.UL.y
            self.rotate(x,y)

    def breakingPoint(self):
        # BK = 0
        for i in range(1,self.n+1):
            if (not self.Squares[-i,self.n+1-i].horizontal):
                return abs(self.n+1-i)
        return 0

    def randomize_limit(self,s:int):
        # SqLen = len(self.Rotation_Boxes)
        while self.rotations < s:
            self.rotations += random.geometric(p=(len(self.Turnable_Boxes))/self.SqLen
                )
            self.actual_Rotations += 1
            Sq = sample(self.Turnable_Boxes,1)[0]
            x = Sq.UL.x
            y = Sq.UL.y
            self.rotate(x,y)
        return self.breakingPoint()

if __name__ == '__main__':
    K = Grid(4,4)
    K.rotate(-4,1)
    print(K.Turnable_Boxes)
```

```python

def addDomino(x,y,horizontal=True):
    col = (horizontal and Hcol) or Vcol # assign relevant color palette
    mod = (horizontal and [1,0]) or [0,1] # assign relevant modifier
    f.write('\t<path \n\t\td="')
    # Draw a rectangle from start position M and subsequent points
    f.write('M ' + str(cWid*x +spacing) + " " + str(cWid*y + spacing))
    f.write(' L ' + str(cWid*(x+1+mod[0])-spacing) + " " + str(spacing+cWid*y))
    f.write(' L ' + str(cWid*(x+1+mod[0]) - spacing) + " " + str(cWid*(y+1+mod[1])-
        spacing))
    f.write(' L ' + str(cWid*x + spacing) + " " + str(cWid*(y+1+mod[1])-spacing))
    f.write(' Z" \n \t\tfill="'+col[(x+y)%2]+'" /> \n ')
```

```python
# Generates a SVG file based on source CSV file
import csv
import sys
# from svglib.svglib import svg2rlg
# from reportlab.graphics import renderPDF, renderPM

# import Surface

# manual entries for source and dest files, relative path from RandomTilings folder
source = 'data/Test25.csv'
dest = 'images/svg/Test25.svg'
pdfdest = 'images/svg/Test25.pdf'
convert2pdf = True
# BK = Surface.BK(source)

cWid = 10 # set width of a case
Hcol = ["orange","crimson"] # set color pair for horizontal dominos
Vcol = ["darkviolet","green"] # set color pair for vertical dominos
spacing = 1 # set space between dominos


def addDomino(x,y,horizontal=True):
    col = (horizontal and Hcol) or Vcol # assign relevant color palette
    mod = (horizontal and [1,0]) or [0,1] # assign relevant modifier
    f.write('\t<path \n\t\td="')
    # Draw a rectangle from start position M and subsequent points
    f.write('M ' + str(cWid*x +spacing) + " " + str(cWid*y + spacing))
    f.write(' L ' + str(cWid*(x+1+mod[0])-spacing) + " " + str(spacing+cWid*y))
    f.write(' L ' + str(cWid*(x+1+mod[0]) - spacing) + " " + str(cWid*(y+1+mod[1])-
        spacing))
    f.write(' L ' + str(cWid*x + spacing) + " " + str(cWid*(y+1+mod[1])-spacing))
    f.write(' Z" \n \t\tfill="'+col[(x+y)%2]+'" /> \n ')


with open(dest,'w') as f:
    with open(source,'r') as input:
        nrow = sum(1 for line in csv.reader(input))
        ncol = nrow
        f.write('<svg xmlns="http://www.w3.org/2000/svg" height="' + str(nrow*cWid) +'
            " width="' + str((ncol)*cWid) + '">\n')
        f.write('\t<g stroke="gray" stroke-width="1">\n')
        input.close()
    with open(source,'r') as input:
        y = 0
        for line in csv.reader(input):
            x = 0
            for n in line:
                if int(n) == 1:
                    addDomino(x,y,True)
                elif int(n) == 2:
                    addDomino(x,y,False)
                x+=1
            y += 1
    #     for k in BK:
    #         f.write ('<circle cx="' + str(k[0]*cWid + cWid/2) + '" cy = "' + str(k[1]*cWid + cWid/2) + '" r="' + str(cWid/2) + '" fill = "black" /> \n')
    f.write("\t</g>\n</svg>")

# if convert2pdf:
#     drawing = svg2rlg(dest)
#     renderPDF.drawToFile(drawing, pdfdest)
```

```python

def addDomino(x,y,horizontal=True):
    col = (horizontal and Hcol) or Vcol # assign relevant color palette
    mod = (horizontal and [1,0]) or [0,1] # assign relevant modifier
    f.write('\t<path \n\t\td="')
    # Draw a rectangle from start position M and subsequent points
    f.write('M ' + str(cWid*x +spacing) + " " + str(cWid*y + spacing))
    f.write(' L ' + str(cWid*(x+1+mod[0])-spacing) + " " + str(spacing+cWid*y))
    f.write(' L ' + str(cWid*(x+1+mod[0]) - spacing) + " " + str(cWid*(y+1+mod[1])-
        spacing))
    f.write(' L ' + str(cWid*x + spacing) + " " + str(cWid*(y+1+mod[1])-spacing))
    f.write(' Z" \n \t\tfill="'+col[(x+y)%2]+'" /> \n ')
```

References

[1] Manuel Fendler, Daniel Grieser. *A New Simple Proof of the Aztec Diamond Theorem*, 2014

[2] Noam Elkies, Greg Kuperberg, Michael Larsen, James Propp. *Alternating Sign Matrices and Domino Tilings*, 1991

[3] Pierre Perruchaud.*Chaînes de Markov*, 2022