

## 1 Introdução

Este capítulo apresenta a construção e uso do laboratório de simulações do Game of Life de John Conway para a realização de experimentos que tem por objetivo investigar a hipótese causal de como as regras de sobrevivência e nascimento de células influenciam na estabilidade de um sistema aleatório do Game of Life.

## 2 O Fenômeno do Mundo Real

O Game of Life é um sistema de autômatos celulares, inventado por John Conway na década de 70, que tentam de alguma forma representar a vida, onde a sobrevivência ou nascimento de uma célula depende da quantidade de células vizinhas.

O jogo original, possui as seguintes regras:

1. Se uma célula viva possui exatamente (2 ou 3) vizinhos vivos, ela permanece viva, caso contrário ela morre.
2. Se uma célula morta possui exatamente (3) vizinhos vivos, ela nasce, caso contrário ela permanece morta.

É válido ressaltar que existem diversas variações para o Game of Life com regras diferentes, como visto em [https://en.wikipedia.org/wiki/Conway%27s\\_Game\\_of\\_Life#Variations](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life#Variations), e a definição das regras é normalmente padronizada como uma string. Por exemplo, a regra do jogo padrão é escrita como "B3/S23", por que é preciso 3 células vizinhas vivas para que uma célula nasça (B de "born"), e que 2 ou 3 células vizinhas estejam vivas para que uma célula sobreviva (S de survive).

## 3 O Laboratório Mesa Game of Life

O laboratório busca simular o comportamento do Game Of Life. A partir das regras em 2, podemos construir listas que definem as regras sobre as células vivas (2 e 3 no exemplo acima) e regras sobre células mortas (3).

A ideia é que quanto mais próximo a média dessas listas forem, mais estável o fenômeno é, onde nós podemos definir que um sistema é estável quando uma das seguintes forem satisfeitas:

1. O estado atual é igual ao estado anterior.
2. O sistema entra em um loop de  $n$  ciclos, onde o passo  $n+1$  é igual ao primeiro passo.

No exemplo clássico do John Conway a média da lista de células vivas é 2,5  $((2+3)/2)$  enquanto a de células mortas é 3  $(3/1)$ , portanto podemos ver o quão próximo essas médias são em dividindo uma pela outra, e quanto mais próximo de 1, maior a semelhança das médias.

### 3.1 O Conceito da Simulação

O modelo MESA Game of life é um modelo de simulação do Game of Life, baseado no exemplo disponível em <https://github.com/projectmesa/mesa-examples/blob/master/examples/ConwaysGameOfLife/> , fornecido no repositório do framework MESA.

### 3.2 O Simulador

O novo modelo adiciona a possibilidade de alterar as regras básicas do John Conway, no arquivo *"rules.TXT"* do repositório:

"B3/S23"

#### 3.2.1 Variáveis Independentes ou de Controle

São as seguintes as variáveis Independentes ou de Controle, manipuláveis na interface do simulador:

**Largura** Largura do Grid.

**Altura** Altura do Grid.

**Rule** String que representa o conjunto de regras do Game Of Life, o padrão é 'B3/S23'.

**Survive** Lista que contém inteiros definindo o número de vizinhos necessários para uma célula sobreviver.

**Density** Probabilidade de cada célula nascer na primeira iteração, fixado em 0,5.

#### 3.2.2 Variáveis Dependentes

**Semelhanca** Semelhança entre o estado atual e o último estado em porcentagem.

**Average rule** Variável para armazenar a relação entre a média da soma das regras, utilizado para ver se há relação com a semelhança.

**Alive cells** Porcentagem de células vivas.

**Dead cells** Porcentagem de células mortas.

### 3.3 O Código do Simulador

Nesta seção serão apresentadas todas as classes que fazem parte do código do simulador e justificar o funcionamento do mesmo.

### 3.3.1 GameOfLifeModel

Classe que representa o Modelo do experimento, disponível em `src/model.py`

Para criação de um objeto a partir dessa classe, são necessários os seguintes parâmetros: Largura, Altura, Rules e Density, descritos em 3.2.1.

Ao inicializarmos a classe, são criados dois *grids*, isso é feito para que a cada passo, possamos analisar a diferença do estado atual com o estado anterior, é importante ressaltar que usamos o argumento `Torus` como `"False"` por que a ideia inicial é que as células de cada ponta não sejam vizinhos da outra, como é descrito em <https://web.stanford.edu/class/sts145/Library/life.pdf>:

```
# novo grid com a altura e largura passado na construçao do objeto
self.grid = Grid(largura, altura, torus=False)
# mantemos um grid do ultimo passo para fazer o
# calculo da diferenca entre estados.
self.lastGrid = Grid(largura, altura, torus=False)
```

Depois disso, é feito um analisador sintático da variável `"rules"` para que ela possa ser passada como lista de inteiros para cada agente (podemos entender melhor o por quê disso em 3.3.2). Por exemplo, a string `"B3/S23"` é convertida em duas listas, a primeira chamada `born` que é igual a `[3]` e a segunda é chamada `survive`, que é igual a `[2,3]`.

# making the list of integers to use in rule:

```
#split string into a list. ex.: "B3/S23" -> ["B3", "S23"]
string_split = rule.split('/')

# REMOVE LETTER B from first part of string_list
born_int = string_split[0].replace('B', '')
# make each element after that an integer
born = [int(x) for x in born_int]

# REMOVE LETTER A from first part of string_list
survive_int = string_split[1].replace('S', '')
# make each element after that an integer
survive = [int(x) for x in str(survive_int)]

# get average of each rule (sum of each element/ length array)
born_average = get_average(born)
survive_average = get_average(survive)
```

Após a criação das listas, é feito uma iteração pelo grid e é inicializado um agente em cada posição, com o estado inicial de Morto. O comportamento e a coleta de cada dado passo a passo é melhor definido em 3.3.3.

Importante destacar que `GameOfLifeModel` herda os métodos da classe `Model` do framework `Mesa` <https://mesa.readthedocs.io/en/latest/apis/init.html>.

### 3.3.2 GameOfLifeAgent

Classe que representa os agentes do experimento, disponível em `src/agent.py`.

Para criação de um objeto a partir dessa classe, são necessários os seguintes parâmetros: posição, modelo, survive, born, estado inicial. As variáveis serão descritas a seguir:

**Posicao** Tupla contendo a posição (x,y) do agente no *grid*.

**Modelo** Objeto modelo em que o agente será utilizado (GameOfLifeModel).

**Estado Inicial** Estado Inicial da célula (vivo ou morto, 0 ou 1), o *default* é MORTO.

**Survive** Lista contendo inteiros referentes ao numero de vizinhos necessários para que uma célula sobreviva, ex.: [2,3].

**Born** Lista contendo inteiros referentes ao numero de vizinhos necessários para que uma célula nasça, ex.[3]:.

Importante destacar que GameOfLifeAgent herda os métodos da classe Agent do framework Mesa <https://mesa.readthedocs.io/en/latest/apis/init.html>.

A definição das regras em 2, no *framework* MESA, é feita a cada passo, na função `step()`:

```
def step(self):
    """Define se a celula estara viva ou morta no proximo passo
    """
    numero_vizinhos_vivos = 0
    # contando os vizinhos vivos:
    for vizinho in self.getVizinhos:
        if vizinho.estaVivo:
            numero_vizinhos_vivos += 1

    #aqui guardamos a posicao da celula no ultimo estado.
    self.model.lastGrid[self.x][self.y].estado = self.estado

    # game of life rules:

    # if cell is alive:
    if self.estaVivo:
        # se o numero de vizinhos nao for igual a algum numero
        if numero_vizinhos_vivos not in self.survive:
            self.proximoEstado = self.MORTO
        else:
            self.proximoEstado = self.VIVO
    # if cell is dead:
```

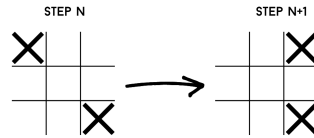


Figure 1: Duas etapas em uma simulação.

```

else:
    if numero_vizinhos_vivos in self.born:
        self.proximoEstado = self.VIVO

```

### 3.3.3 Comportamento do Modelo

Agora que entendemos melhor as classes que compõem o laboratório, podemos explicar melhor o seu código, com ênfase nas funções que coletam e calculam as variáveis Dependentes 3.2.1.

**Compute Likeness** A função "compute likeness" recebe o modelo 3.3.1 e retorna um float que corresponde a semelhança em porcentagem do estado atual, comparado ao último.

Considere o exemplo 1. onde X representa uma célula viva, e vazio representa uma célula morta:

Podemos ver que duas células mudaram de estado, a célula (1,1) e a célula (1,3). Logo, nosso algoritmo calculará quais células estão em estados diferentes para poder calcular a porcentagem de semelhança com o estado atual a partir da equação a seguir:

**S** Semelhança entre estado atual e o último estado em porcentagem.

**TC** Total de Células no jogo.

**CD** Número de células com estado diferente do estado passado.

$$S = (TC - CD)/TC$$

No exemplo em 1 teríamos

$$(9 - 2)/9 = 0.777..$$

O código a seguir representa como esse valor é calculado dentro da função compute likeness:

```
def compute_likeness(model):

    """Função que retorna a semelhança em porcentagem do
    estado atual do modelo,
    comparado ao estado passado."""

    soma = 0
    #iteracao pelo grid:
    for grid_content, x, y in model.grid.coord_iter():
        celula_atual_estado = grid_content.estado
        #se o estado do grid passado é diferente do estado atual, somamos 1
        celula_anterior_estado = model.lastGrid.grid[x][y].estado
        if celula_atual_estado != celula_anterior_estado:
            soma = soma + 1
    number_of_cells = model.largura * model.altura
    semelhanca = (number_of_cells - soma)/number_of_cells
    return semelhanca
```

Podemos ver o funcionamento do gráfico em um jogo de regras relativamente estáveis, como a "B3/S23" em 2. A medida que os elementos vão ficando estáveis, ou seja, param de mudar, a semelhança entre cada estado aumenta.

A primeira iteração deve ser desconsiderada uma vez que o grafo é inicializado em todos estados como 0, então a primeira semelhança sempre será o número de novos elementos que receberam o estado VIVO, que é passado pela densidade (fixado em 0.5).

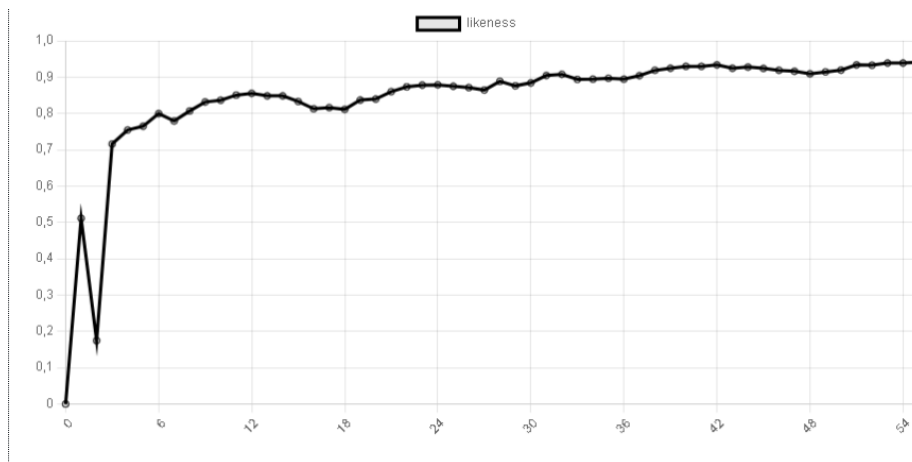


Figure 2: Gráfico da evolução da variável semelhança ao decorrer de uma simulação.

**Alive and Dead Cells** Essas funções são mais simples e retornam a percentagem de células vivas e células mortas no jogo, descritas no código a seguir:

```
def compute_alive_cells(model):
    """Função que retorna a percentagem de células vivas (variável dependente)"""

    alive_cells = 0
    # numero de células
    number_of_cells = model.largura * model.altura

    for grid_content, x, y in model.grid.coord_iter():
        if grid_content.estado == grid_content.VIVO:
            alive_cells = alive_cells + 1
    # percentagem de células vivas = células_vivas/total_células
    alive_cells_percent = alive_cells / number_of_cells
    return alive_cells_percent

def compute_dead_cells(model):
    """Função que retorna a percentagem de células mortas (variável dependente)"""
    dead_cells = 0
    # numero de células
    number_of_cells = model.largura * model.altura

    for grid_content, x, y in model.grid.coord_iter():
        if grid_content.estado == grid_content.MORTO:
            dead_cells = dead_cells + 1
    # percentagem de células mortas = células_mortas/total_células
    dead_cells_percent = dead_cells / number_of_cells

    return dead_cells_percent
```

É feito uma iteração pelo *grid* e somamos as células que estão vivas ou mortas, no final dividimos esse valor pelo número total de células para permanecemos em um valor de 0 a 1.

**Average Rule** Por fim mantemos essa variável que é simplesmente a divisão da média das regras. No exemplo da regra "B3/S23", temos que a média da lista Born é 3 e da lista Survive é 2,5. Logo, é feito a divisão desses dois números e quanto mais próximo de 1, maior a semelhança entre eles.

Uma observação importante é que esse número não vai necessariamente de 0 a 1. Podemos ter uma média de survive maior que a média de born e consequentemente teremos um número maior que um, logo, se um número da regra S é muito maior que da regra B, teremos uma média alta, e se tivermos uma média próximo de 0, teremos que a média de B é muito maior que a de S. Podemos utilizar essas ideias para conjecturar hipóteses depois.

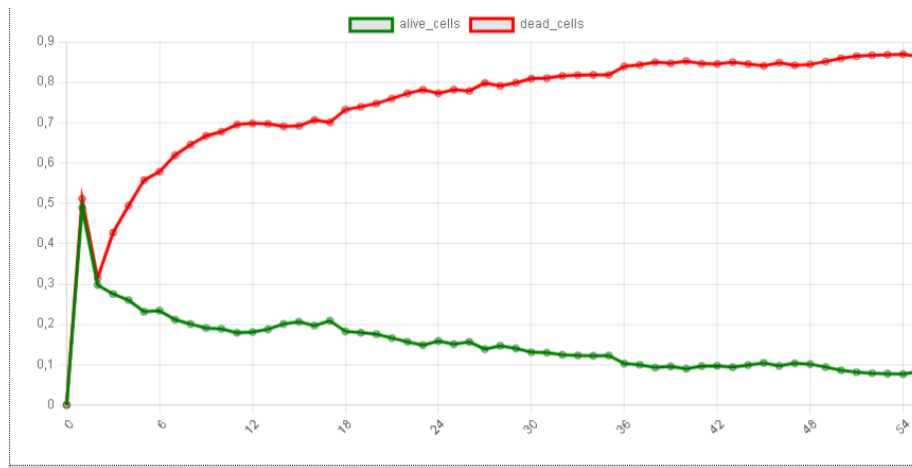


Figure 3: Gráfico da evolução da porcentagem de células vivas(verde) e células mortas(vermelho) ao decorrer de uma simulação.

```
...
#code inside GameOfLifeModel.init()
born_average = get_average(born)
survive_average = get_average(survive)
self.average_rule = survive_average / born_average
...

#code outside of GameOfLifeModel class
def get_average(list):
    """Get average of integers list"""
    list_total = 0
    for number in list:
        list_total = list_total + number
    return list_total / len(list)
```

### 3.4 A Hipótese Causal

A hipótese que me veio a cabeça é que quanto mais proporcional são as regras de sobrevivência e nascimento, mais estável é o fenômeno. Já discutimos brevemente o que representa estabilidade em 3. No código escrito estamos calculando a diferença entre o estado atual e estado anterior, ou seja, se não há diferença entre esses dois estados, o sistema não muda e consequentemente, ele é estável.

Infelizmente não foi possível codificar um algoritmo que calcule se o sistema está estável em um *loop* ou ciclo. Mas a ideia é que quanto menos as células mudem, mais léxicos estáveis são formados, portanto é estabelecido uma semel-



hança maior entre dois estados e a porcentagem é mais próxima de 100.

## 4 Os Experimentos Realizados

Alguns gráficos do comportamento de variáveis independentes como semelhança e porcentagem de células vivas ou mortas já foram apresentadas na descrição de 2 e 3, entretanto, queremos ter a possibilidade de realizar diversos experimentos de forma rápida e coletar esses dados de alguma maneira para futura análise. Isso é feito com a instancição de uma classe do *framework* MESA chamada de **BatchRun**.

### 4.1 BatchRun

Para codificação do módulo BatchRun, foi criado um novo arquivo (`batch_run.py`) que é executado separadamente do módulo do servidor (`run.py`).

A codificação do módulo pode ser vista com comentários a seguir:

```
import mesa
import pandas as pd
import numpy as np
from datetime import datetime
from src.model import GameOfLifeModel

#parametros da classe (utilizado para fazer o resultado
# do arquivo .csv dinâmico):

experiments_per_parameter_configuration = 100
max_steps_per_simulation = 300
max_steps = 500
iterations = 10

# parametros do modelo: vamos iterar por diferentes regras para análise
params = {
    "largura": 50,
    "altura": 50,
    "density": .5,
    "rule": ["B2/S34", "B3/S23", "B6/S16", "B36/S23", "B1357/S1357", "B34/S34", "B15/S28"]
}

if __name__ == "__main__":
    # running the test and storing it in results
    results= mesa.batch_run(
        GameOfLifeModel,
```

```

parameters=params,
iterations=iterations,
max_steps=max_steps,
data_collection_period=-1,
display_progress=True
)

# converting results to csv
results_df=pd.DataFrame(results)
now = str(datetime.now()).replace(":", "-").replace(" ", "-")
file_name_suffix = ("_iter_" + str(experiments_per_parameter_configuration) + "_steps_"
results_df.to_csv("results/Game_of_life_model_data" + file_name_suffix + ".csv", sep=';')

```

Pela variável "params" são passadas diversas regras diferentes para que seja possível *plotar* gráficos e fazer análises a respeito da nossa hipótese 3.4.

A escolha dessas regras não foi aleatória, esses padrões são versões diferentes do Game of Life comuns, descritos em <https://cs.stanford.edu/people/eroberts/courses/soco/projects/2008-09/modeling-natural-systems/gameOfLife2.html>. Algumas conhecidas por serem estáveis, outras por terem um grande crescimento de células, entre outros padrões.

#### 4.1.1 Problemas e dificuldades na execução dos experimentos

Infelizmente, foi necessário a realização de muita fatoração para que o modelo consiga realizar testes iguais aos argumentos do módulo BatchRun. Anteriormente, as variáveis das regras de born e survive 3.2.1 eram passadas como lista, por exemplo, `born=[3] survive=[2,3]`, porém, o módulo de testes considerou isso como duas cargas de teste diferentes por estarem sendo passadas em um *array*.

A solução foi passar a regra como *string*, que foi descrita em 3.3.1. O ponto positivo é que esse tipo de padrão é utilizado em diversos artigos então facilita no entendimento geral do código.

## 4.2 Resultados do Experimento

Foram realizados 70 iterações, 10 para cada regra (7 regras diferentes). Cada iteração consistia em 500 passos, isso foi feito após analisar que a média da semelhança tendia a ser uma função crescente, e as vezes, após 400 passos ou mais o sistema finalmente ficava estável (semelhança = 1).

```

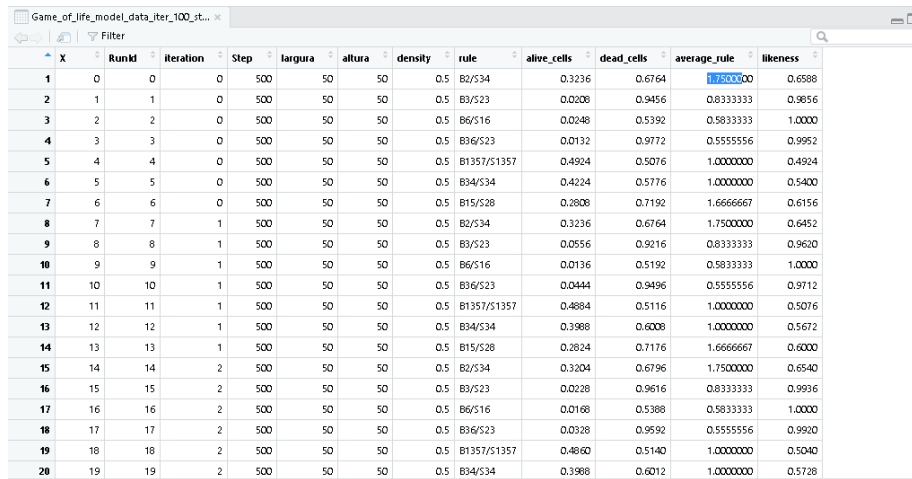
params = {
    "largura": 50,
    "altura": 50,
    "density": .5,
    "rule": ["B2/S34", "B3/S23", "B6/S16",

```

```
"B36/S23", "B1357/S1357",  
"B34/S34", "B15/S28"]}]}
```

#### 4.2.1 Utilizando o R Studio

Foi utilizado o ferramenta R Studio para análise e plotagem inicial dos gráficos. O dataset é convertido em ".csv", como especificado em 4.1 e depois importado no RStudio: Import Dataset - From text (base) - results/Game\_of.life\_model\_data\_iter\_100\_steps\_300\_2022-08-18-15-31-54.293149.



X	RunId	Iteration	Step	largura	altura	density	rule	alive_cells	dead_cells	average_rule	likeness
1	0	0	0	500	50	50	0.5 B2/S34	0.3236	0.6764	0.7500000	0.6588
2	1	1	0	500	50	50	0.5 B3/S23	0.0208	0.9456	0.8333333	0.9856
3	2	2	0	500	50	50	0.5 B6/S16	0.0248	0.5392	0.5833333	1.0000
4	3	3	0	500	50	50	0.5 B36/S23	0.0132	0.9772	0.5555556	0.9952
5	4	4	0	500	50	50	0.5 B1357/S1357	0.4924	0.5076	1.0000000	0.4924
6	5	5	0	500	50	50	0.5 B34/S34	0.4224	0.5776	1.0000000	0.5400
7	6	6	0	500	50	50	0.5 B15/S28	0.2808	0.7192	1.6666667	0.6156
8	7	7	1	500	50	50	0.5 B2/S34	0.3236	0.6764	1.7500000	0.6452
9	8	8	1	500	50	50	0.5 B3/S23	0.0556	0.9216	0.8333333	0.9620
10	9	9	1	500	50	50	0.5 B6/S16	0.0136	0.5192	0.5833333	1.0000
11	10	10	1	500	50	50	0.5 B36/S23	0.0444	0.9496	0.5555556	0.9712
12	11	11	1	500	50	50	0.5 B1357/S1357	0.4884	0.5116	1.0000000	0.5076
13	12	12	1	500	50	50	0.5 B34/S34	0.3988	0.6008	1.0000000	0.5672
14	13	13	1	500	50	50	0.5 B15/S28	0.2824	0.7176	1.6666667	0.6000
15	14	14	2	500	50	50	0.5 B2/S34	0.3204	0.6796	1.7500000	0.6540
16	15	15	2	500	50	50	0.5 B3/S23	0.0228	0.9616	0.8333333	0.9936
17	16	16	2	500	50	50	0.5 B6/S16	0.0168	0.5388	0.5833333	1.0000
18	17	17	2	500	50	50	0.5 B36/S23	0.0328	0.9592	0.5555556	0.9920
19	18	18	2	500	50	50	0.5 B1357/S1357	0.4860	0.5140	1.0000000	0.5040
20	19	19	2	500	50	50	0.5 B34/S34	0.3988	0.6012	1.0000000	0.5728

Figure 4: Tabela das 20 primeiras iterações dos testes.

#### Código RStudio

```
# Import Datasets -> From text (base) -> Dataset - From text (base) #-> results/Game_of
dataset <-Game_of_life_model_data_iter_100_steps_300_2022.08.18.15.31.54.293149

hist(dataset$likeness)
hist(dataset$alive_cells)
hist(dataset$dead_cells)

library(ggplot2)
ggplot(new_metadata) +
  geom_point(aes(x = age_in_days, y= samplemeans))
```

A partir dessa importação, podemos começar a *plotar* gráficos e ter alguns *insights*. Os códigos da plotagem pode ser visto em 4.2.1

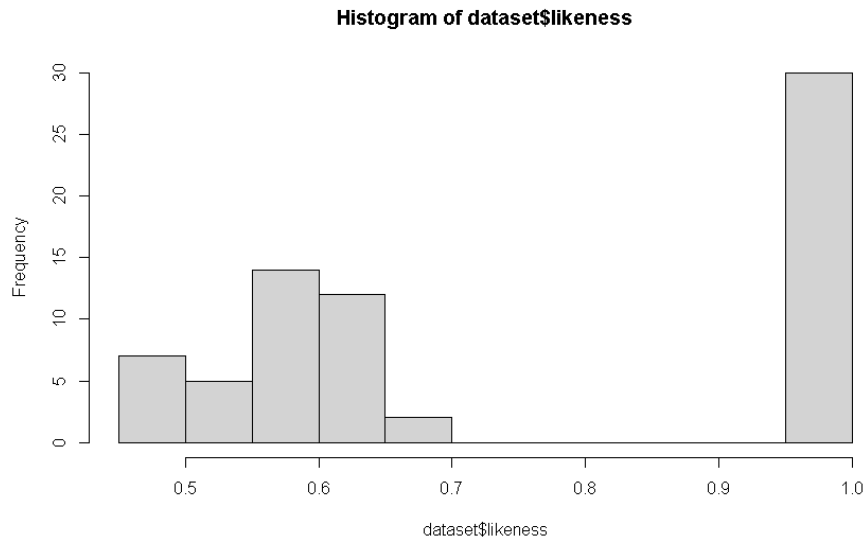


Figure 5: Histograma de Semelhança

**Histograma de Semelhança** No gráfico 5 podemos ver a frequência da semelhança de cada modelo ao final de 500 passos entre diferentes regras.

Aqui podemos ver que algumas são mais estáveis, com semelhança próximas ou iguais a 1 e outras são menos estáveis, formando aparentemente uma distribuição normal entre valores menores ou iguais a 0.7.

**Histograma de Células Vivas** Podemos ver que é muito mais comum ao final de 500 passos, que existam muitas poucas células vivas, próximas de 0, caso contrário ela aparenta estar uniformemente distribuída entre valores acima de 2.5.

A partir desse histograma já podemos imaginar que ao verificarmos o histograma da porcentagem de células mortas, teremos uma grande quantidade próximo a 1 e uma distribuição uniforme em valores menores.

**Histograma de Células Mortas** Como dito anteriormente, aparentemente há uma concentração entre 0.9 e 1 de células mortas enquanto o resto do gráfico é uniforme. Podemos ver uma exceção número maior entre 0.5 e 0.6 por ser referente ao intervalo que aparentava ser maior em 6 .

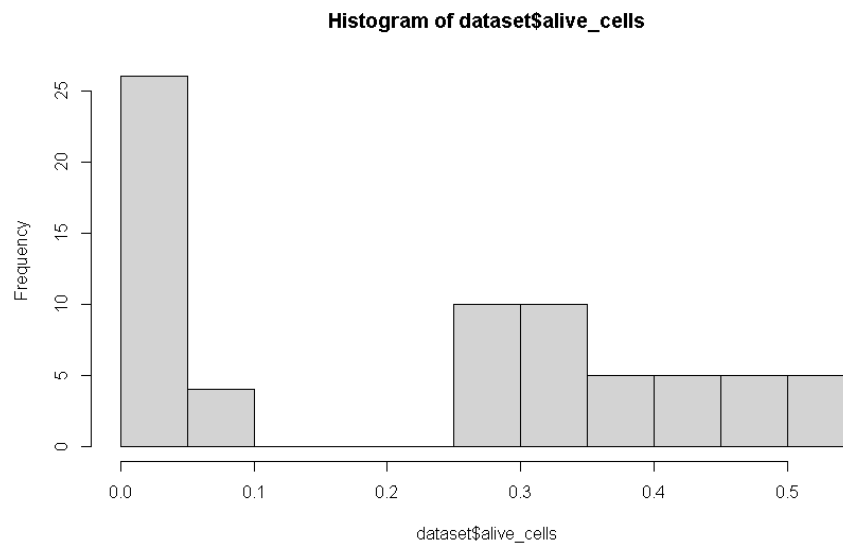


Figure 6: Histograma da Porcentagem de Células Vivas

**Gráfico Semelhança x Regra Média** No grafo em 8 podemos ver uma distribuição onde X é a Semelhança do modelo e Y é a média das listas de regras de inteiros 3.4 Apesar de ser possível analisar algumas relações como, que quando a regra se aproxima de 0.5 temos uma semelhança maior, não acredito que seja possível tirar nenhuma conclusão com o número de dados coletados (70 iterações).

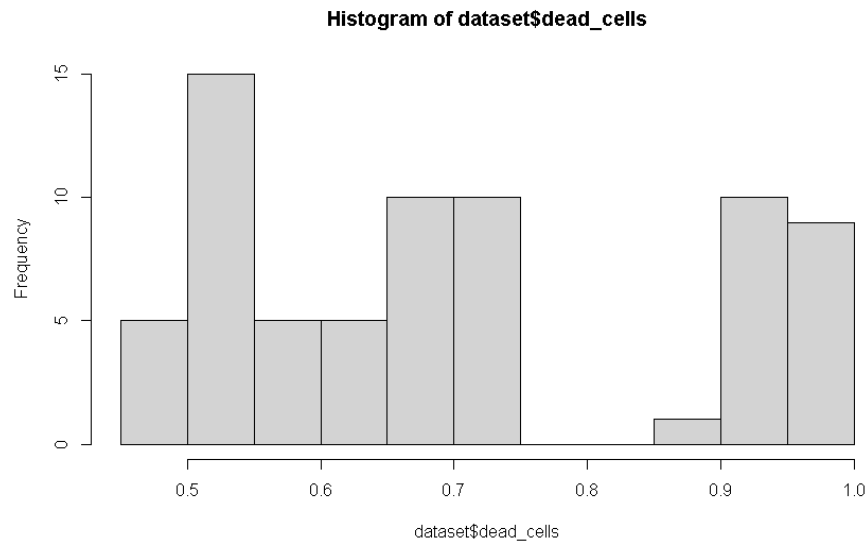


Figure 7: Histograma da Porcentagem de Células Mortas

## 5 Discussão e *insights* preliminares sobre as hipóteses

Como vimos em 8, não foi possível garantir nenhuma relação ou encontrar algum padrão na distribuição que relaciona a semelhança com a média entre as regras. Seria necessário aumentar o número de regras para que possamos ver se há alguma relação ou se é aleatório como aparenta ser.

Entretanto foram possíveis tirar boas conclusões respeito do funcionamento de células vivas e mortas e como a estabilidade desse sistema funciona ao passar do tempo.

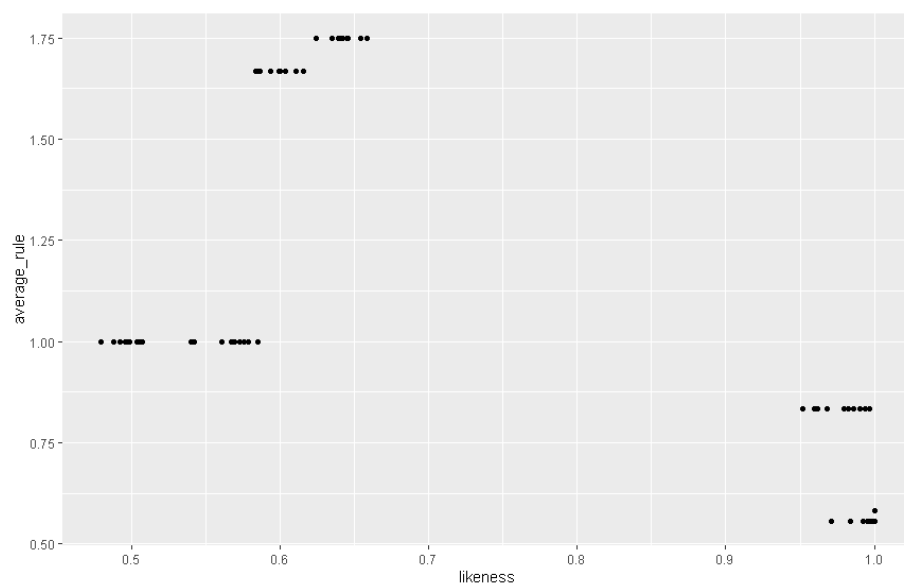


Figure 8: Gráfico Semelhança x Regra Média