AC-demo

June 29, 2023

1 Autômatos Celulares

Os autômatos celulares são uma classe de modelos computacionais que consistem em uma grade ou rede de células interconectadas que evoluem ao longo do tempo de acordo com um conjunto de regras predefinidas. Essas células geralmente têm um estado ou valor que pode ser atualizado com base nos valores de seus vizinhos imediatos.

Modelos de autômatos celulares são definidos por:

- Uma **matriz de células**, cada qual possuindo um **estado** (ou **valor**), havendo um número finito de estados possíveis para as células;
- Uma **regra de transição**, que define como os **estados** destas células serão atualizados a cada **iteração**.

Todas as células atualizam seus estados simultaneamente de acordo com a regra de transição definida para a simulação. Por exemplo, eis uma regra de transição para autômatos celulares unidimensionais e algumas gerações após a definição do estado inicial das células:

Adicionalmente, as seguintes caractereísticas são válidas ou importantes para a maioria dos autômatos celulares:

- *As regras de transição em geral são dadas em função do estado atual da célula e o de suas **vizinhas***, para alguma definição de vizinhança pré-estabelecida. Por exemplo, a seguir estão algumas vizinhanças comumente adotadas:
- É necessário definir alguma **topologia** para a matriz de células. Na prática, isso significa estabelecer quais são as vizinhas das células nas extremidades da matriz. A topologia mais comum é considerar uma topologia toroidal (também conhecida por "periódica"), onde as células em cada extremidade consideram as células das extremidades opostas da matriz suas vizinhas:

Também é comum definir uma topologia onde as células das extremidades não têm vizinhos para além da extremidade.

Autômatos celulares são muito úteis para a modelagem de sistemas dinâmicos que apresentam dependência espacial, *porque são por definição discretos no **tempo** e no **espaço***, sendo suas menores unidades a **célula** e a **iteração**, respectivamente. Isso simplifica muito sua implementação e análise em computador, e os permite fazer uso das capacidades de multiprocessamento da máquina, pois diferentes conjuntos de células podem ser atualizados em paralelo por cada unidade de processamento. Além do mais, são interessantes no sentido de que *regras em geral muito simples dão origem a comportamentos complexos e inesperados em uma grande escala*, como poderão ver pelas demonstrações a seguir.

Termos importantes:

- **Autômatos Celulares**: Modelo computacional descrito no texto acima. Abreviado por **AC**.
- **Célula**: Unidade fundamental de espaço, corresponde aos elementos em uma matriz, com determinada posição e valor.
- **Estado**: Também chamado de "valor", é um número (por exemplo) guardado por cada célula que pode mudar a cada iteração. Por definição, existe um número finito de valores possíveis para as células em um AC.
- **Tabuleiro**: Se refere à matriz de células, com seus valores e posições e regra de transição definidas. Não é um termo comum na literatura de ACs, mas usado no contexto do tomato-engine.
- **Regra de transição**: Também referida simplesmente por "regra". Define como os estados das células serão atualizados para gerar a próxima iteração do tabuleiro, e é geralmente dada em função do estado da célula e de suas vizinhas.
- **Iteração**: Corresponde a uma aplicação da regra de transição em todas as células. Lembrando que todas as células têm seus estados atualizados **simultaneamente**.

2 Tomato-engine

O **tomato-engine** é uma biblioteca que busca **facilitar** ao máximo a implementação e exibição de quaisquer regras de ACs em Python.

Ao usar o tomato-engine, você não precisa criar um sistema que execute regras de ACs do zero e pode se concentrar em estudar os ACs em si, implementando novas regras e as modificando rapidamente. Também te dá a vantagem de poder desfrutar de todo o ambiente do Python, que conta com bibliotecas muito robustas para análise de dados, criação de gráficos e matemática, para seus trabalhos com ACs.

2.1 Modelo do tomato-engine

Basicamente, o tomato-engine trabalha com dois arquivos (chamados "módulos" em Python) e duas classes:

- Um módulo de **regra**, onde é definida a classe **Cell**, que representa as células na simulação e define a **regra de transição** a ser usada.
- Um módulo de **simulação**, onde a classe **Board**, que representa o **tabuleiro** (definido na seção acima), é inicializada com o **arquivo de regra** e a matriz de **estados iniciais das células**.

No caso de uso em um notebook do Jupyter, costuma ser mais prático ter uma **célula de regra** e uma **célula de simulação** ao invés de arquivos separados (**não confundir com as células do autômato celular, estamos falando de células do jupyter aqui**). As regras demonstradas neste notebook estão declaradas desta maneira.

A classe Cell possui **atributos** e **métodos**, que são váriaveis e funções, respectivamente (em programação, chama-se variáveis e funções que pertencem a classes de "atributos" e métodos"). Os atributos são os seguintes:

- **self.pos**: uma tupla de 2 números que armazena a **posição da célula no tabuleiro** na forma de **linha** e **coluna** (nesta ordem). Este atributo é definido automaticamente para cada

célula no começo da simulação, então na prática é uma constante que você pode usar em suas regras, mas nunca modificar.
- **self.val**: um número que representa o **estado** da célula. Você pode adotar quaisquer números para representar os estados em seu modelo, e de preferência deve escolhê-los de modo a facilitar a programação e interpretação do modelo.
- **self.state_matrix**: uma matriz de Numpy que contém os **estados de todas as células** na geração atual. Você pode consultar o estado de qualquer célula nesta matriz indexando-a com índices absolutos, ou usando a **linha** e **coluna** da célula em questão para fazer indexação relativa.
- **self.neighbors**: uma *lista com os valores das células na vizinhança considerada pela regra.* O tomato-engine define várias das vizinhanças mais comuns (por exemplo as apresentadas na figura 2) por padrão, de modo que se você quiser usar alguma delas, basta especificar qual no código.

Você também pode definir qualquer número de atributos auxiliares para a criação de sua regra. O "self" indica que estes atributos dizem respeito à célula, ao invés de serem globais. Por exemplo, como cada célula tem seu próprio estado, para a célula consultar o *seu próprio estado* devemos usar **self.val**, e o mesmo vale para os demais atributos. Variáveis globais, ou seja, que não dizem respeito a nenhuma célula específica, podem ser definidas com `Cell.<nome-da-variavel>`.

E os métodos mais importantes são:

- **update**: **atualiza o estado da célula**. Este é o método mais importante, e corresponde à aplicação da regra de transição. É executado por todas as células simultanemanete para dar origem a uma nova geração.
- **display**: retorna uma **tupla de valores RGB** (ou seja, uma **cor**) para cada estado possível da célula. Não interfere na simulação em si, é usado somente para a visualização do tabuleiro. Por padrão, ele retorna `(0, 0, 0)` (preto) para `self.val = 0` e (255, 255, 255) para `self.val != 0`, mas quase sempre você precisará redefinir este método para se adequar ao número de estados dos quais o seu modelo faz uso.
- **from_display**: É a função inversa de `display`, ou seja, retorna o estado da célula que corresponde a cada cor. É usado somente para inicializar a simulação a partir de uma imagem contendo os estados iniciais das células.
- **init**: Método executado por cada célula somente uma vez, no início da simulação. Útil para auxiliar na definição de valores iniciais.
- **simulation_start**: Método executado pelo **tabuleiro** somente uma vez antes da primeira geração. Não é executado por cada célula, portanto, sua utilidade é definir constantes e valores iniciais.

A classe Board também tem seus próprios métodos e atributos. Você deve saber de pelo menos os seguintes métodos:

- **load_state**: carrega o estado inicial da simulação, seja diretamente a partir de uma matriz com os valores iniciais de cada célula, ou a partir de uma imagem, usando o método `from_display` apresentado acima. Este método não inicia a simulação imediatamente.
- **start**: carrega o estado inicial da simulação **e a inicia**, por padrão exibindo cada geração em uma janela ou no próprio notebook do jupyter.
- **update**: executa o método `update` de cada célula. Ou seja, este é o método que gera cada nova geração do modelo.

O seguinte diagrama ilustra todas estas informações:

Dado o volume de informações, pode parecer muito difícil trabalhar com autômatos celulares. Na prática, é bem mais simples do que parece – você reaproveitar a maior parte do código de exemplos e mudar somente os métodos que interessam, principalmente o `update`. *O tomato-engine faz com que modelos simples sejam fáceis de implementar e modelos complexos sejam possíveis.*

Há uma série de vídeo-tutoriais no canal da Comunidade de Software Livre de Ribeirão Preto, na plataforma Odysee.

2.2 Instalação

Assim como a maioria das bibliotecas de Python, o tomato-engine pode ser instalado por meio do **pip**. A célula abaixo instalará o tomato-engine ou o atualizará caso já esteja instalado:

```
[ ]: !pip install --upgrade tomato-engine
```

Para usar o tomato-engine de maneira integrada em um notebook do Jupyter, é necessário instalar o **matplotlib**:

```
[ ]: !pip install --upgrade matplotlib
```

Note que não é necessário executar estas células de instalação todas as vezes que for usar o tomato-engine

2.3 Observações

A prioridade no desenvolvimento do tomato-engine foi (e é) ser fácil de usar, principalmente para criar regras novas e fazer experimentos com as regras já existentes. Também priorizamos a máxima versatilidade possível, de modo que *qualquer regra de autômato celular pode ser implementada no tomato-engine*. Na prática isso implica em alguns compromissos, principalmente quanto ao **desempenho**. O tomato-engine não é adequado para executar simulações acima da ordem de grandeza de 10 mil células (que equivale a uma matriz 100x100), tanto devido ao consumo de RAM quanto à velocidade da simulação (o paralelismo no tomato-engine ainda é muito limitado).

Ao executar uma simulação no tomato-engine por meio do método `start`, ela continuará executando até que o usuário feche a janela ou interrompa o *kernel* no jupyter. Portanto, se estiver usando o Jupyter, lembre-se de sempre clicar no *quadrado* ao lado do botão de executar a célula (o *play*) na parte de cima da interface do Jupyter para parar a simulação (pode ser necessário clicar neste botão várias vezes), ou especificar o número máximo de gerações da simulação, após as quais ela parará automaticamente. Para tal use o argumento `generations=<número-de-gerações>` no método `board.start`.

3 Algumas regras

A seguir estão alguns exemplos de regras e suas explicações. Recomendo fortemente ler suas descrições e executá-las na ordem em que são apresentadas.

3.1 Game of Life

O clássico Game of Life de Conway.

Estados possíveis:

- Vivo: 1, branco
- Morto: 0, preto

Regra:

- Procriação: Célula morta fica viva na próxima geração se tiver 3 vizinhos vivos
- Sobrevivência: Célula viva sobrevive para a próxima geração se tiver 2 ou 3 vizinhos vivos

Esta regra bastante simples gera vários padrões interessantes e surpreendentes, de modo que a visualização da simulação parece um sistema vivo. Você pode experimentar alterar as condições de sobrevivência e procriação e ver as diferenças. Experimente também alterar a vizinhança.

O Game of Life é bastante estudado por matemáticos e cientistas da computação, e possui sua própria Wiki: conwaylife.com.

[ ]:
```
from tomato.classes import cell

class LifeCell(cell.CellTemplate):
    # {{{
    def update(self, state_matrix):
        self.state_matrix = state_matrix

        # Célula morta:
        if self.value == 0:
            if self.live_neighbors == 3:
                self.value = 1
            else:
                self.value = 0
        # Célula viva:
        else:
            if self.live_neighbors in (2, 3):
                self.value = 1
            else:
                self.value = 0

    # Definição de vizinhança. Usa uma das vizinhanças padrão que vêm com o␣
↪tomato-engine
    @property
    def neighbors(self):
        # Experimente alterar a vizinhança. Vizinhanças disponíveis:
        # self.moore_neighborhood
        # self.neumann_neighborhood
        # self.hexagonal_neighborhood
        # self.triangular_edges_neighborhood
        # self.triangular_vertices_neighborhood
        # self.triangular_1vertex_neighborhood
        # Para informações sobre vizinhanças: https://conwaylife.com/wiki/
↪Neighbourhood
        return self.moore_neighborhood
```

```
    # Função auxiliar que conta o número de vizinhos vivos
    @property
    def live_neighbors(self):
        return sum(self.neighbors)


# }}}
```

[ ]:
```
import tomato as tt
from tomato.functions import utils

rule = LifeCell

# Tamanho da célula na visualização
CELL_SIZE = 5

# Dimensões da simulação em número de linhas e colunas de células
DIMENSIONS = (120, 120)

# Criando uma matriz aleatória para servir de estado inicial
initial_state = utils.random_int_matrix(DIMENSIONS, 2, seed=12_1_14_7_20_15_14)

# Inicializa o tabuleiro com a regra
board = tt.Board(rule, cell_size=CELL_SIZE)

# Inicia a simulação. Se inline=False, vai abrir uma janela com a simulação. Se␣
 ↪True, vai mostrar no próprio notebook do Jupyter.
# A simulação geralmente roda mais rápido com inline=False. Mas se você está␣
 ↪usando o colab, inline tem que ser True.
board.start(initial_state, inline=True)
```

3.2 Sequência de palitos de dente

Gera um padrão idêntico ao resultante de uma "sequência de palitos de dente".

Estados possíveis:

- Vivo: 1, branco
- Morto: 0, preto

Regra:

- Células mortas ficam vivas se possuem exatamente 1 vizinha viva. Células vivas nunca morrem

Veja por si mesmo o padrão interessante que uma regra tão simples gera um fractal tão intricado. Se você tem um computador bom (ou muita paciência), experimente aumentar as dimensões e diminuir o `cell_size` para ver melhor o caráter fractal deste padrão.

Vale a pena experimentar com diferentes vizinhanças e mudar a condição de nascimento das células.

Reproduza este padrão com os palitos de dente da próxima vez que for a um restaurante e me mande a foto, e receba meu contentamento e congratulações.

[ ]:
```
from tomato.classes import cell


class ToothpickCell(cell.CellTemplate):
    # {{{
    # Método executado somente uma vez no início da simulação. Neste caso, ele só facilita mudar a vizinhança
    @classmethod
    def simulation_start(cls, state_matrix, cell_args):
        neighborhood = getattr(ToothpickCell, f"{cell_args}_neighborhood")
        setattr(ToothpickCell, "neighbors", neighborhood)

    def update(self, state_matrix):
        self.state_matrix = state_matrix

        if sum(self.neighbors) == 1:  # somente um vizinho vivo
            self.value = 1


# }}}
```

[ ]:
```
import tomato as tt
from tomato.functions import utils

rule = ToothpickCell

CELL_SIZE = 5
DIMENSIONS = (120, 120)

# Experimente alterar a vizinhança. Vizinhanças disponíveis:
# "moore"
# "neumann"
# "hexagonal"
# "triangular_edges"
# "triangular_vertices"
# "triangular_1vertex"
# Para informações sobre vizinhanças: https://conwaylife.com/wiki/Neighbourhood
NEIGHBORHOOD = "neumann"

# Matriz inicial é totalmente = 0, exceto no elemento do meio, que é = 1
initial_state = utils.point_matrix(DIMENSIONS, 1)

board = tt.Board(rule, cell_size=CELL_SIZE)
board.start(initial_state, cell_args=NEIGHBORHOOD, inline=True)
```

3.3 Autômatos cíclicos

Autômatos celulares cíclicos

Estados possíveis:

- Inteiros entre 0 e `NUM_STATES - 1`, definido na célula de simulação.
  - Suas cores são gradações de cinza calculadas a partir do número de estados possíveis.

Regra:

- Cada célula verifica se possui algum vizinho cujo valor é igual ao seu prório valor + 1. Se sim, ela assume este valor.

Esta regra espontaneamente gera vórtices que se propagam com velocidade e frequência definidas se inicializada a partir de uma matriz aleatória. É interessante mudar o número de valores possíveis e a definição de vizinhança para ver os efeitos.

Este modelo foi estudado pela primeira vez pelo próprio Alan Turing, no período em que ele se interessou pelas oportunidades de estudo que a computação oferecia à química. Em particular, isso busca modelar a dissipação de pigmentos na pele de animais (referência).

[ ]:
```
from tomato.classes import cell

class CyclicCell(cell.CellTemplate):
    # {{{

    # Método executado somente uma vez no início da simulação. Neste caso, ele␣
 ↪facilita mudar o número de estados possíveis e a vizinhança na célula de␣
 ↪simulação.
    @classmethod
    def simulation_start(cls, state_matrix, cell_args):

        num_states = cell_args.get("num_states", 12)
        neighborhood = cell_args.get("neighborhood", "neumann")

        CyclicCell.num_states = num_states
        CyclicCell.shades = tuple(n*(256//CyclicCell.num_states) for n in␣
 ↪range(CyclicCell.num_states))

        neighborhood = getattr(CyclicCell, f"{neighborhood}_neighborhood")
        setattr(CyclicCell, "neighbors", neighborhood)


    def update(self, state_matrix):
        self.state_matrix = state_matrix

        next_val = (self.value + 1) % CyclicCell.num_states

        if next_val in self.neighbors:
            self.value = next_val
```

```
    @staticmethod
    def display(value):
        return CyclicCell.shades[value]

    @staticmethod
    def from_display(rgb):
        return rgb[0] * (256 // CyclicCell.num_states)


# }}}
```

[ ]:
```
import tomato as tt
from tomato.functions import utils

rule = CyclicCell

CELL_SIZE = 5
DIMENSIONS = (120, 120)

# Experimente alterar o número de estados e a vizinhança
NUM_STATES = 10
NEIGHBORHOOD = "neumann"
CELL_ARGS = {"num_states": NUM_STATES, "neighborhood": NEIGHBORHOOD}

initial_state = utils.random_int_matrix(DIMENSIONS, NUM_STATES)

board = tt.Board(rule, cell_size=CELL_SIZE)
board.start(initial_state, cell_args=CELL_ARGS, inline=True)
```

3.4 Autômatos Celulares Elementares

A classe de autômatos celulares unidimensionais que Stephen Wolfram estudou em sua obra A New Kind of Science

Estados possíveis:

- Vivo: 1, branco
- Morto: 0, preto

Regra:

- Este módulo de regra permite executar qualquer regra de ACs elementares definida na notação de Wolfram, que já foi apresentada aqui

O tomato-engine foi projetado para ACs bidimensionais, mas como os AC unidimensionais podem ser vistos como um caso particular destes, eles também podem ser implementados sem muita dificuldade. Neste caso, aproveitamos as linhas do tabuleiro para exibir o histórico das gerações.

[ ]:
```
from tomato.classes import cell
import numpy as np

class Cell1D(cell.CellTemplate):
    # {{{

    # Todas as combinações possíveis de estados da célula e das vizinhas, de acordo com a notação de Wolfram.
    possible_states = [# {{{
        [1,1,1],    # rule_array[0]
        [1,1,0],    # rule_array[1]
        [1,0,1],    # rule_array[2]
        [1,0,0],    # rule_array[3]
        [0,1,1],    # rule_array[4]
        [0,1,0],    # rule_array[5]
        [0,0,1],    # rule_array[6]
        [0,0,0],    # rule_array[7]
    ]# }}}

    @classmethod
    def simulation_start(cls, state_matrix, cell_args):
        Cell1D.rule_tuple = cell_args

        Cell1D.generation = 0

    def update(self, state_matrix):
# {{{
        self.state_matrix = state_matrix

        if self.lin == self.generation + 1:
            for i, state in enumerate(Cell1D.possible_states):
                if np.array_equal(state, self.neighbors):
                    self.value = Cell1D.rule_tuple[i]
        self.generation += 1
# }}}

    # Este é o primeiro exemplo de definição própria da vizinhança. Note como usamos
    # self.lin e self.col (linha e coluna da célula) para encontrar as suas vizinhas
    @property
    def neighbors(self):
# {{{
        state_matrix = self.state_matrix
        m, n = state_matrix.shape

        prev_lin = self.lin - 1
```

```
        # Dividimos por n (número de colunas) aqui para que as células das␣
 ↪bordas
        # considerem como vizinhas as células no outro lado da matriz. Ou seja,
        # topologia toroidal
        prev_col = (self.col - 1) % n
        next_col = (self.col + 1) % n

        return [
            state_matrix[prev_lin, prev_col],
            state_matrix[prev_lin, self.col],
            state_matrix[prev_lin, next_col],
        ]
# }}}

# }}}
```

[ ]:
```
import tomato as tt
import numpy as np

rule = Cell1D

# Função de conveniência que permite se referir a uma regra na notação de␣
 ↪Wolfram
# por sua representação decimal, que é a mais comum
def by_decimal_num(num):
    return tuple(bin(num).replace("0b", "").zfill(8))

cell_size = 4
size = (100, 100)

# Algumas regras que valem a pena verificar:
# 13, 17, 18, 28, 30, 45, 57, 60, 62, 73, 75, 89
# 102, 101, 105, 110, 129, 131, 135, 150 and 225
rule_num = 30

# Começando com uma célula viva no centro e as demais mortas. É interessante
# testar cada regra com estados iniciais diferentes também
state_matrix = np.zeros(size)
state_matrix[0, size[1] // 2] = 1

board = tt.Board(rule, cell_size=cell_size)
board.start(
    state_matrix,
    cell_args=by_decimal_num(rule_num),
    generations=size[1], # simulação vai parar após executar a iteração n. <num.
 ↪ de colunas>
```

```
    inline=True,
)
```

3.5 Passeio Aleatório

Um exemplo de como implementar modelos com **Agentes**, isto é, células que podem se mover pelo tabuleiro.

Estados possíveis: - Para as células: - rastro dos agentes: 1 a 100 - fundo preto: 0 - Para os agentes: - a cor do agente na visualização: tripla RGB entre 0 e 255
- posição: tupla de linha e coluna

Regra:

A cada iteração, o agente escolhe dois números, dirlin e dircol, que podem ser -1, 0 ou 1 cada. Eles adicionam estes números às suas variáveis lin e col, que representam a linha e coluna que ocupam no tabuleiro. Assim, eles escolhem uma direção aleatória e dão um passo.

As células verificam se existe um Agente em sua posição, e mudam de cor para pintar o "rastro" da caminhada dos agentes.

Esta regra está próxima de ser o exemplo mais simples possível de uma regra com Agentes. Leia os comentários com atenção para ver as diferenças na implementação de Agentes e Células.

[ ]:
```
from random import choice

from tomato.classes import agent, cell # Vamos usar o módulo *agent* dessa vez, além do *cell*
from collections import namedtuple

# Tupla que será usada para criar a lista de estados iniciais dos agentes
WalkerTuple = namedtuple("WalkerTuple", ["val", "pos"])

class BackgroundCell(cell.CellTemplate): # <- Note o cell.CellTemplate
    # {{{
    """
    Classe para as células de "fundo" no tabuleiro. Tudo o que elas fazem neste
    modelo é mostrar o rastro de por onde os agentes andam.
    """

    # Para modelos com agentes, precisamos de um argumento *state_list* na
    # função update, além do state_matrix de sempre
    def update(self, state_matrix, state_list):

        # O state_list é uma lista com os agentes. Eles possuem os mesmos
        # atributos que a WalkerTuple definida na linha 7
        self.agent_list = state_list

        # self.agent_above retorna o agente na "acima" da célula, ou seja,
        # na mesma posição em linhas e colunas. Se não houver nenhum agente
```

```
        # "em cima" da célula, retorna False. Por isso este *if* funciona.
        if self.agent_above:

            # O valor da célula é a intensidade de sua cor, e mostra o rastro
            # que os agentes deixam pelo tabuleiro
            self.value = 100
        else:

            # Se não houver um agente acima, o rastro esmaecerá
            self.value -= 2
            if self.value < 0:
                self.value = 0

    @staticmethod
    def display(value):
        return (value, value, value)

# }}}


class Walker(agent.AgentTemplate): # <- Note o agent.AgentTemplate
# {{{
    """
    Classe para os agentes. Eles andam aleatoriamente pelo tabuleiro.
    Note o agent.AgentTemplate na linha 46. Isso diz ao tomato-engine
    que esta é uma classe para Agentes, e não Células.
    """

    # Para modelos com agentes, precisamos de um argumento *state_list* na
    # função update, além do state_matrix de sempre
    def update(self, state_matrix, state_list):

        # Seleciona aleatoriamente uma direção para o próximo passo
        dirlin = choice((-1, 0, 1))
        dircol = choice((-1, 0, 1))

        # Dá o próximo passo. O % implementa a topologia toroidal.
        m, n = state_matrix.shape
        self.lin = (self.lin + dirlin) % m
        self.col = (self.col + dircol) % n

    # O valor de cada agente é uma tupla de valores R, G, B que será a
    # a cor deles na simulação.
    @staticmethod
    def display(value):
        return value
# }}}
```

[ ]:
```
import numpy as np
import tomato as tt

agent_class = Walker
cell_class = BackgroundCell

cell_size = 6
dimensions = (100, 100)

# O número de agentes e seus valores iniciais devem estar em uma
# lista, assim como o das células deve estar em uma matriz.
# Definimos a WalkerTuple na célula acima. O primeiro elemento é
# o valor inicial do agente e o segundo sua posição em (lin, col).
walker_list = [
    WalkerTuple((255, 0, 0), (20, 20)),
    WalkerTuple((0, 255, 255), (20, 80)),
    WalkerTuple((255, 255, 0), (80, 20)),
    WalkerTuple((255, 0, 255), (80, 80)),
]

state_matrix = np.zeros(dimensions)

# Precisamos passar a classe da Célula e do Agente para a Board
board = tt.Board(cell_class, agent_class, cell_size=cell_size, max_fps=200)

board.start(state_matrix, walker_list, inline=True)
```

3.6 Formiga de Langton

Outro famoso modelo de autômatos celulares, criado pelo célebre Christopher Langton. Outro exemplo envolvendo uma classe para Agentes.

Estados possíveis: - Para as células: - Célula branca: 1 - Célula preta: 0 - Para a formiga: - Direção: tupla de 2 elementos, que representam as componentes do vetor direção da formiga num sistema de cordenadas em que **y** aponta para cima e **x** para a esquerda - Posição: tupla de 2 elementos, com a posição inicial da formiga em linha e coluna

Regra:

- A cada iteração, a formiga checa o estado da célula sobre a qual está:
  - 1: a formiga muda o estado da célula para 0, move uma célula no sentido do seu vetor direção, e gira esse vetor em 90 graus no sentido **horário**
  - 0: a formiga muda o estado da célula para 0, move uma célula no sentido do seu vetor direção, e gira esse vetor em 90 graus no sentido **anti-horário**

Deixada sobre uma matriz cheia de células pretas, a formiga começa desenhando um padrão aparentemente caótico, até espontaneamente chegar a um estado estacionário em que ela faz um **"túnel"**. Ela permanece construíndo este túnel até encontrar alguma "perturbação" (ou seja, alguma célula branca) em seu caminho.

Esta nova versão (dia 29/02) é mais fácil de entender que a original, e permite ainda que várias formigas existam no tabuleiro, e com direções e passos diferentes.

[ ]:
```python
from collections import namedtuple

from tomato.classes import agent, cell

# Tupla que será usada para criar a lista de estados iniciais dos agentes.
# O primeiro elemento é o vetor direção da formiga, que é uma tupla de 2,
# números inteiros.
AntTuple = namedtuple("AntTuple", ["dir", "pos"])


class BackgroundCell(cell.CellTemplate):
    # {{{
    """
    Classe para as células de "fundo" no tabuleiro. Elas só verificam se
    existe uma formiga em cima delas, e caso sim, e alternam seus valores.
    """

    def update(self, state_matrix, agent_list):
        self.agent_list = agent_list

        # A AntTuple com a mesma posição da célula, caso exista. Se não
        # existir, é False.
        agent_above = self.agent_above

        # É importante sempre usar um *if* desse para verificar se agent_above
        # não é False antes de fazer qualquer outra coisa com esta variável.
        if agent_above:

            # Alterna seu valor. Se é 0, fica 1, e vice-versa. Isso funciona
            # porque em python 1 == True e 0 == False.
            self.value = not self.value


# }}}


class Ant(agent.AgentTemplate):
    # {{{
    """
    Classe para as formigas.

    Note o agent.AgentTemplate na linha 34. Isso diz ao tomato-engine
    que esta é uma classe para Agentes, e não Células.
    """
```

```
    # Quando se trabalha com modelos que envolvem Agentes, é necessário
    # que todas as funções *update* tenham como argumento state_matrix e
    # agent_list, mesmo que uma (ou ambas) não seja usada pela função em si
    # (como neste caso, onde agent_list não é usada).
    def update(self, state_matrix, agent_list):
        self.state_matrix = state_matrix

        # A célula "abaixo" do Agente, ou seja, com a mesma posição em
        # linhas e colunas. Ao contrário da agent_above para Células,
        # esta variável sempre retorna o estado de uma célula, e nunca False
        # (porque sempre existe uma célula abaixo de qualquer Agente).
        cell_below = self.cell_below
        if self.cell_below:  # célula branca
            self.value = (  # virar no sentido anti-horário
                self.value[1],
                -self.value[0],
            )

        else:  # célula preta
            self.value = ( # virar no sentido horário
                -self.value[1],
                self.value[0],
            )

        self.move(state_matrix)

    def move(self, state_matrix):
        m, n = state_matrix.shape

        self.lin = (self.lin + self.value[0]) % m
        self.col = (self.col + self.value[1]) % n

    @staticmethod
    def display(value):
        # Nesta regra as formigas sempre são vermelhas
        return (255, 0, 0)


# }}}
```

[ ]:
```
import numpy as np
import tomato as tt

agent_class = Ant
cell_class = BackgroundCell
```

```
cell_size = 4
dimensions = (100, 200)

# Vamos colocar três formigas no tabuleiro, cada uma com uma
# direção inicial diferente
ant_list = [
    AntTuple((1, 0), (50, 50)),
    AntTuple((0, 2), (50, 100)),
    AntTuple((1, 2), (50, 150)),
]

state_matrix = np.ones(dimensions)

# A ordem em que você passa a cell_class e agent_class importa.
# cell_class sempre vem primeiro.
board = tt.Board(cell_class, agent_class, cell_size=cell_size)
board.start(
    state_matrix,
    ant_list,
    inline=True,
)
```

3.7 Turmites

Generalização da formiga de Langton, adicionando mais estados possíveis à formiga e às células.

Estados possíveis: - Para as células: - Números inteiros entre 0 e `e o tamanho do nome da formiga` (explicado a seguir): - 0: preto - 1: branco - 2: vermelho - 3: verde - 4: azul - 5: rosa - 6: amarelo - 7: ciano - Valores n $>$ 7 são representados pela cor do valor n % 7. - Para a formiga: - Direção: tupla de 2 elementos, que representam as componentes do vetor direção da formiga num sistema de cordenadas em que **y** aponta para cima e **x** para a esquerda - Posição: tupla de 2 elementos, com a posição inicial da formiga em linha e coluna

Regra:

- Mesma ideia da formiga de Langton, exceto que o sentido para o qual a formiga girará seu vetor posição ao ver cada estado é definido pelo *"nome"* da formiga
  - O nome da formiga é uma string de Ls e Rs, onde R significa "girar no sentido horário ao ver este valor" e L, "girar no sentido anti-horário ao ver este valor"
  - Por exemplo, a formiga de Langton tem nome "RL"

Experimente diferentes nomes para a formiga e veja os padrões gerados. O que acontece se o nome da formiga for um *palíndromo*?

Se definir um nome acima de 7 caracteres, lembre-se de definir `loop_colors: True`.

Obs: Esta regra deve ser atualizada para fazer uso das novas funcionalidades do tomato-engine para modelos com agentes.

[ ]:
```
from tomato.classes import cell

class TurmiteCell(cell.CellTemplate):
    # {{{

    # fmt: off
    colors = (
        (0, 0, 0),          # Black
        (255, 255, 255),    # White
        (255, 0, 0),        # Red
        (0, 255, 0),        # Green
        (0, 0, 255),        # Blue
        (255, 0, 255),      # Pink
        (255, 255, 0),      # Yellow
        (0, 255, 255),      # Cyan
    )
    # fmt: on

    def __init__(self, val, pos, cell_args):

        global ant_pos
        global ant_dir

        self.value = val
        self.lin, self.col = pos

        ant_pos = cell_args["pos"]
        ant_dir = cell_args["dir"]
        TurmiteCell.name = cell_args["name"]
        TurmiteCell.colors = cell_args.get("colors", TurmiteCell.colors)

        if cell_args.get("loop_colors", False):
            TurmiteCell.display = TurmiteCell.display_looping


    def update(self, state_matrix):

        global ant_pos
        global ant_dir

        if ant_pos == self.pos:
            self.value = (self.value + 1) % len(TurmiteCell.name)

            if TurmiteCell.name[self.value] == 'R':
                ant_dir = (
                    -ant_dir[1],
                    ant_dir[0],
```

```
                )
            elif TurmiteCell.name[self.value] == 'L':
                ant_dir = (
                    ant_dir[1],
                    -ant_dir[0],
                )

            # Wrap around the board when close to its edge
            m, n = state_matrix.shape
            ant_pos = (
                (ant_dir[0] + ant_pos[0]) % m,
                (ant_dir[1] + ant_pos[1]) % n,
            )

    @staticmethod
    def display(value):
        return TurmiteCell.colors[value]

    @staticmethod
    def display_looping(value):
        # Wraps around the colors tuple. Note that it makes the
        # simulation slower and loading from a png will not work
        # as expected.
        return TurmiteCell.colors[value % len(TurmiteCell.colors)]

    @staticmethod
    def from_display(value):
        return TurmiteCell.colors.index(tuple(value))

# }}}
```

[ ]:
```
import tomato as tt
from numpy import zeros

rule = TurmiteCell

cell_size = 5
dimensions = (100, 100)
ant_args = {
    "pos": (dimensions[0] // 2, dimensions[1] // 2),
    "dir": (0, -1),
    "name": tuple("RLR"), # Nome da formiga
    "loop_colors": False, # Coloque True aqui se o nome for acima de 8␣
 ↪caracteres. Isso torna a simulação um pouco mais lenta
}

state_matrix = zeros(dimensions, dtype='uint8')
```

```
board = tt.Board(rule, cell_size=cell_size, max_fps=120)
board.start(state_matrix, cell_args=ant_args, inline=True)
```

3.8 Tomato-life

Três variações do Game of Life competem entre si

Estados possíveis:

- Preto (morto): 0
- Branco: 1
- Vermelho: 2
- Verde: 3

Regra:

- Cada "espécie" de célula segue uma regra:
  - Branco: Nasce com 3 vizinhos brancos, sobrevive com 2 ou 3 (B3/S23)
  - Vermelho: Nasce com 3 vizinhos vermelhos, sobrevive com 2, 3 ou 8 (B3/S238)
  - Verde: Nasce com 3 ou 6 vizinhos verdes, sobrevive com 2 ou 3 (B36/S23)
- Há uma ordem de "pedra-papel-tesoura" para a prioridade de cada célula nascer, caso a condição seja cumprida para mais de uma espécie:
  - Branco vence verde
  - Verde vence vermelho
  - Vermelho vence branco
- No caso de uma célula morta, a ordem de prioridade é verde -> vermelho -> branco

Essa regra serve de exemplo para as funções `display`, `from_display`, `save_png`, e `start` a partir de uma imagem.

Ao usar uma imagem como estado inicial, *é muito importante se atentar ao* `cell_size`, pois ele dirá qual o tamanho das células na imagem de entrada. Por exemplo, se `cell_size = 4`, o tomato-engine entenderá que cada célula é um quadrado de 4x4 pixels na imagem.

[ ]:
```
from tomato.classes import cell

class TomatoCell(cell.CellTemplate):
    # {{{

    # Dead: 0
    # White: 1, B3/S23
    # Red: 2, B3/S238
    # Green: 3, B36/S23

    # Dead: white > red > green
    # Live: white > red > green > white

    def update(self, state_matrix):
        self.state_matrix = state_matrix
```

```
        # Dead cell
        if self.value == 0:
            if self.live_green_neighbors in (3, 6):
                self.value = 3
            elif self.live_red_neighbors == 3:
                self.value = 2
            elif self.live_white_neighbors == 3:
                self.value = 1
            else:
                self.value = 0

        # White cell
        elif self.value == 1:
            if self.live_red_neighbors == 3:
                self.value = 2
            elif self.live_white_neighbors in (2, 3):
                self.value = 1
            elif self.live_green_neighbors in (3, 6):
                self.value = 3
            else:
                self.value = 0

        # Red cell
        elif self.value == 2:
            if self.live_green_neighbors in (3, 6):
                self.value = 3
            elif self.live_red_neighbors in (2, 3, 8):
                self.value = 2
            elif self.live_white_neighbors == 3:
                self.value = 1
            else:
                self.value = 0

        # Green cell
        elif self.value == 3:
            if self.live_white_neighbors == 3:
                self.value = 1
            elif self.live_green_neighbors in (2, 3):
                self.value = 3
            elif self.live_red_neighbors == 3:
                self.value = 2
            else:
                self.value = 0

    @property
    def neighbors(self):
```

```
        return self.moore_neighborhood

    @property
    def live_white_neighbors(self):
        return self.neighbors.count(1)

    @property
    def live_green_neighbors(self):
        return self.neighbors.count(3)

    @property
    def live_red_neighbors(self):
        return self.neighbors.count(2)

    @staticmethod
    def display(value):
        if value == 1:
            return (255, 255, 255)
        elif value == 2:
            return (255, 0, 0)
        elif value == 3:
            return (0, 255, 0)
        else:
            return (0, 0, 0)

    @staticmethod
    def from_display(value):
        if (value == (255, 255, 255)).all():
            return 1
        elif (value == (255, 0, 0)).all():
            return 2
        elif (value == (0, 255, 0)).all():
            return 3
        else:
            return 0


# }}}
```

[ ]:
```
import tomato as tt
from tomato.functions import utils

rule = TomatoCell

# Neste caso o CELL_SIZE é bastante importante. Experimente mudar este valor e␣
 ↪observar o efeito no estado inicial da simulação.
CELL_SIZE = 4
```

```
dimensions = (100, 100)
state_matrix = utils.random_choice_matrix(dimensions, (0, 2, 1, 3))

board = tt.Board(rule, img_cell_size=CELL_SIZE)

# A simulação vai parar após 100 gerações e salvar seu estado na imagem
 ↪tomato-life-100.png
board.start("tomato-logo.png", inline=True, generations=100)
#board.start(state_matrix, inline=True, generations=100) # Descomente isso se
 ↪quiser usar a matriz aleatória

board.save_png("tomato-life-100.png")
```

3.9 Maze of Chaos

Um caso surpreendente de organização emergente e caos.

Estados possíveis:

- Branco: 0
- Vermelho: 1
- Verde: 2
- Morto: 3 ou mais

Regra:

- Nesta regra também há 3 *espécies* (brancas, vermelhas, verdes). Mas desta vez todas seguem a mesma regra:
  - Células vivas são **convertidas** se possuem 2 ou 3 vizinhos de uma mesma cor
  - Células mortas **nascem** se possuem 3 ou mais vizinhos de uma mesma cor
- As mesmas ordem de prioridade do tomato-life valem aqui:
  - Células vivas: branca -> vermelha -> verde -> branca
  - Células mortas: branca -> vermelha -> verde

Ao iniciar a simulação, provavelmente haverão *colônias* de células de uma espécie que crescem até encontrar colônias de outras espécie. Observe o que acontece quando colônias de duas espécies diferentes se encontram, e quando três espécies diferentes se encontram.

Para explorar este sistema, você pode mudar o parâmetro prob_dead (percentual de células mortas na matriz aleatória), pode mudar a vizinhança considerada e a regra. Recomendo mudar a condição de conversão para "mais de 2 vizinhos da mesma cor", e também testar outras vizinhanças.

[ ]:
```
from tomato.classes import cell

# Checa se a condição de conversão é satisfeita para um dado número
# de vizinhos. Está aí para facilitar mudar este número de uma só
# vez em todo o código.
def conversion_condition(num_neighbors):
    return num_neighbors in (2, 3)
```

```
# Checa se a condição de conversão é satisfeita para um dado número
# de vizinhos. Está aí para facilitar mudar este número de uma só
# vez em todo o código.
def birth_condition(num_neighbors):
    return num_neighbors > 2

class ChaosMazeCell(cell.CellTemplate):
    # {{{

    # branco = 1
    # vermelho = 2
    # verde = 3
    # morto = 0 ou maior que 3

    def update(self, state_matrix):
        self.state_matrix = state_matrix

        if self.value == 3:
            if conversion_condition(self.live_red_neighbors):
                self.value = 2
            elif conversion_condition(self.live_green_neighbors):
                self.value = 3
            elif conversion_condition(self.live_white_neighbors):
                self.value = 1
        elif self.value == 1:
            if conversion_condition(self.live_green_neighbors):
                self.value = 3
            elif conversion_condition(self.live_white_neighbors):
                self.value = 1
            elif conversion_condition(self.live_red_neighbors):
                self.value = 2
        elif self.value == 2:
            if conversion_condition(self.live_white_neighbors):
                self.value = 1
            elif conversion_condition(self.live_red_neighbors):
                self.value = 2
            elif conversion_condition(self.live_green_neighbors):
                self.value = 3
        else:
            if birth_condition(self.live_white_neighbors):
                self.value = 1
            elif birth_condition(self.live_red_neighbors):
                self.value = 2
            elif birth_condition(self.live_green_neighbors):
                self.value = 3
```

```python
    @property
    def neighbors(self):
        # Teste outras vizinhanças também!
        return self.moore_neighborhood

    @property
    def live_neighbors(self):
        return (
            self.live_white_neighbors,
            self.live_red_neighbors,
            self.live_green_neighbors,
        )

    @property
    def live_white_neighbors(self):
        return self.neighbors.count(1)

    @property
    def live_green_neighbors(self):
        return self.neighbors.count(3)

    @property
    def live_red_neighbors(self):
        return self.neighbors.count(2)

    @staticmethod
    def display(value):
        if value == 1:
            return (255, 255, 255)
        elif value == 2:
            return (255, 0, 0)
        elif value == 3:
            return (0, 255, 0)
        else:
            return (0, 0, 0)

    @staticmethod
    def from_display(value):
        if (value == (255, 255, 255)).all():
            return 1
        elif (value == (255, 0, 0)).all():
            return 2
        elif (value == (0, 255, 0)).all():
            return 3
        else:
            return 0
```

```
# }}}
```

[ ]:
```
import tomato as tt
import numpy as np

rule = ChaosMazeCell

board_dimensions = (120, 120)

cell_size = 4

# Probabilidades de uma célula morta ou viva (respectivamente) serem␣
 ↪selecionadas
# em cada elemento da matriz de estados iniciais. Tente variar estes parâmetros.
prob_dead = 0.96
prob_live = (1 - prob_dead) / 3

state_matrix = np.random.choice(
    (0, 1, 2, 3), size=board_dimensions, p=(prob_dead, prob_live, prob_live,␣
 ↪prob_live)
)

# Como a matriz inicial é aleatória, recomendo executar esta célula várias vezes
# para um mesmo conjunto de parâmetros
board = tt.Board(rule, cell_size=cell_size)
board.start(state_matrix, inline=True)
```

3.10 Agora os ACs aplicados à modelagem de algum sistema:

3.11 Modelo Eden

Um modelo para crescimento de aglomerados, como as células em um tumor, ou uma colônia de bactérias.

Página da wikipedia

Um artigo matemático sobre o modelo

Um artigo mais aplicado na área de física dos materiais sobre este modelo

Estados possíveis:

- Vazio: 0, preto
- Ocupado: 1, branco

Regra:

- A vizinhança adotada é a de Moore
- Neste modelo, as regras não dependem do estado da própria célula, mas somente dos estados de suas vizinhas:

- Se a célula tem 8 vizinhos ocupados, ela estará ocupada na próxima geração
- Se a célula tem entre 2 e 8 vizinhos ocupados, ela tem probabilidade de 50% de estar ocupada na próxima geração
- Se a célula tem menos de 2 vizinhos ocupados, ela estará vazia na próxima geração

Neste modelo, usamos a função `cross_matrix` para gerar espaço de células vazias, exceto por 5 células em formato de "+" no centro. Este é nosso estado inicial.

Recomendo experimentar diferentes condições e probabilidade de nascimento de ocupação das células na regra, bem como diferentes vizinhanças.

Como este modelo é estocástico, você pode ter de executar a simulação algumas vezes até ver um aglomerado crescendo.

[ ]:
```
from tomato.classes import cell
from random import choice

class EdenCell(cell.CellTemplate):
    # {{{
    def update(self, state_matrix):
        self.state_matrix = state_matrix
        live_neighbors = self.live_neighbors

        if live_neighbors > 7:
            self.value = 1
        elif live_neighbors > 1:
            zz = choice((0, 1))
            if zz:
                self.value = 0  # normal
            else:
                self.value = 1  # growth
        else:
            self.value = 0

    @property
    def neighbors(self):
        return self.moore_neighborhood

    @property
    def live_neighbors(self):
        return sum(self.neighbors)


# }}}
```

[ ]:
```
import tomato as tt
from tomato.functions import utils

rule = EdenCell
```

```
cell_size = 4
dimensions = 100

# Uma matriz de zeros com uma cruz de 1s no meio
state_matrix = utils.cross_matrix(dimensions, 1)

board = tt.Board(rule, cell_size=cell_size)
board.start(state_matrix, inline=True)
```

3.12 Disseminação de opinião

Modelo de disseminação de opiniões. Inicialmente concebido especificamente para "fake news". Foi apresentado em 2022 como trabalho da disciplina de Física Computacional pela Luiza Licarião e Bruno Febrônio.

As referências utilizadas foram as seguintes:

Cognitive cascades: How to model (and potentially counter) the spread of fake news

A systematic mapping on automatic classification of fake news in social medi

Hadoop Cellular Automata for Identifying Rumor in Social Networks

Cellular Automaton-Based Controlling Simulation of Fake Information Diffusion in Public Crisis

Estados possíveis:

Neutro (U) - cor: marrom areia, - valor: 0, - explicação: Pessoas que ainda não ouviram as notícias ou formaram uma opinião sobre elas.

Crente (B) - cor: azul céu claro, - valor: 1, - explicação: Pessoas que acreditam na desinformação, mas não estão tentando disseminá-la ativamente.

Disseminador Crente (BD) - cor: cinza escuro, - valor: 2, - explicação: Pessoas que acreditam na desinformação e estão ativamente tentando disseminá-la.

Verificadores de Fatos (FC) - cor: verde maçã, - valor: 3, - explicação: Um estado transitório em que a pessoa está decidindo se a notícia é verdadeira ou não.

Cético (S) - cor: rosa claro, - valor: 4, - explicação: Pessoas que não acreditam na desinformação, mas não estão promovendo ativamente o ceticismo para outras pessoas.

Disseminador Cético (SD) - cor: cáqui, - valor: 5, - explicação: Pessoas que não acreditam na desinformação e estão ativamente tentando convencer outras pessoas de sua falsidade.

Regra:

Confira o arquivo `trabalho-fake-news.pdf` para uma explicação detalhada dos modelos estudados no trabalho.

A regra contida neste notebook é a versão final do modelo, contendo inclusive as melhorias discutidas na seção **Apêndice** do PDF.

[ ]:
```
import numpy as np
from numpy import random
from tomato.classes import cell


class FakeNewsCell(cell.CellTemplate):
    # {{{
    colors = (
        (244, 164, 96),  # N "sandy brown" 0
        (135, 206, 250),  # B "light sky blue" 1
        (169, 169, 169),  # DB "dark gray" 2
        (144, 238, 144),  # FC "granny smith apple" 3
        (255, 182, 193),  # S "light pink" 4
        (240, 230, 140),  # DS "khaki" 5
    )

    def __init__(self, val, pos, cell_args):
        self.lin, self.col = pos
        self.value = val

        self.prob = cell_args[0][self.lin, self.col]  # the individual's␣
 ↪credulity
        self.alpha = cell_args[1]  # preference for immediate neighbors

        FakeNewsCell.rng = np.random.default_rng(cell_args[2])
        FakeNewsCell.random_neighbors_num = cell_args[3]

    def update(self, state_matrix):
        self.state_matrix = state_matrix

        if self.value == 0:
            if self.believing_ratio >= 0.5:
                self.value = 1
            elif self.believing_ratio < 0.5 and self.believing_ratio >= 0.1:
                if random.rand() > 0.7:
                    self.value = 3
                else:
                    self.value = 0
            else:
                self.value = 4

        elif self.value == 1:
            if self.believing_ratio >= 0.5:
                if random.rand() > 0.7:
                    self.value = 2
                else:
                    self.value = 1
```

```
            elif self.believing_ratio < 0.5 and self.believing_ratio >= 0.3:
                self.value = 1
            else:
                self.value = 3

        elif self.value == 3:
            if self.believing_ratio >= 0.8:
                self.value = 1
            elif self.believing_ratio >= 0.6 and self.believing_ratio < 0.8:
                self.value = 0
            else:
                if random.rand() > 0.5:
                    self.value = 4
                else:
                    self.value = 3

        elif self.value == 4:
            if self.believing_ratio <= 0.3:
                if random.rand() > 0.8:
                    self.value = 5
                else:
                    self.value = 4
            elif self.believing_ratio > 0.6:
                self.value = 3
            else:
                self.value = 4

    @property
    def neighbors(self):
        return self.moore_neighborhood

    @property
    # Retorna um número n de vizinhos aleatórios pelo tabulero
    def rand_neighbors(self):
        return FakeNewsCell.rng.choice(self.state_matrix.flatten(),␣
↪FakeNewsCell.random_neighbors_num)

    @property
    # Retorna o valor que será utilizado para determinar a mudança de estado
    def believing_ratio(self):
        B = self.neighbors.count(1) + list(self.rand_neighbors).count(1)
        BD = self.neighbors.count(2) + list(self.rand_neighbors).count(2)
        SD = self.neighbors.count(5) + list(self.rand_neighbors).count(5)
        S = self.neighbors.count(4) + list(self.rand_neighbors).count(4)
        P = self.prob
        alfa = self.alpha
        viz = (alfa / (8)) * (2 * BD + B - 2 * SD)
```

```
        prob = (1 - alfa) * P
        if viz >= 0:
            return viz + prob
        else:
            return prob

    @staticmethod
    def display(value):
        return FakeNewsCell.colors[value]

    @staticmethod
    def from_display(value):
        return FakeNewsCell.colors.index(tuple(value))


# }}}
```

[ ]:
```
import numpy as np
import tomato as tt

rule = FakeNewsCell

cell_size = 6
board_dimensions = (80, 80)
seed = 5

np.random.seed(seed)
prob_matrix = np.random.rand(100, 100)

# "a" são os estados possíveis e "p" são as probabilidades de cada um
state_matrix = np.random.choice(
    a=[0, 1, 2, 3, 4, 5], size=board_dimensions, p=[0.7, 0.05, 0.1, 0.0, 0.05,␣
 ↪0.1]
)

board = tt.Board(rule, cell_size=cell_size)
board.start(state_matrix, cell_args=[prob_matrix, 0.5, seed, 4], inline=True)
```

3.13 Dengue

Modelo de propagação da dengue em zonas urbanas, criado para fazer simulações em diferentes cidades.

Estados e Regras: Este modelo faz uso de números complexos e conta com 6 tipos de células, cujos parâmetros podem ser alterados por meio dos cell_args:

1. Mosquitos: cor: branca, valor: imaginário entre 0 e 3.99 (inclusivo) Andam aleatoriamente pelo tabuleiro. Tem tempo de vida dado por cell_args['mosquito_lifespan']. Quando passam

por cima de uma casa, a infecta com probabilidade cell_args['infection_prob'], morrendo no processo caso cell_args['kamikaze_mosquito'] é True.

2. Casa: cor: verde a vermelho, valor: 2.0 (suscetível) a 3.0 (infectado) Quando infectada, morre com probabidade dada por cell_args['death_prob']. Caso sobreviva, se recupera em cell_args['heal_time'] gerações. Casas só podem ser infectadas quando suscetíveis (valor = 2.0, cor verde).
3. Água: cor: azul a ciano, valor: 4.0 (água 'mexida') a 5.0 (água parada) Quando parada, cria um mosquito com probabilidade cell_args['spawn_prob']. Difunde seu valor com outros vizinhos 'águas': valor = 0,75^(n. de vizinhos 'água') + 0,25*(valor de cada vizinho) + cell_args['water_still_factor'].
4. Inspetor sanitário: cor: magenta, valor: imaginário entre 4.0 e 7.0 (inclusivo) Anda pelas ruas, virando sempre à esquerda. Mata os mosquitos em seu caminho. "Mexe" as águas em sua vizinhança.
5. Rua: cinza, valor: 7.0 Não faz nada, é simplesmente as células sob as quais o inspetor pode andar.
6. Fundo: preto, valor: 8.0 Não faz nada.

Este é de longe o modelo mais complicado já implementado no tomato-engine, pois ele depende de vários **agentes** que se movem pelo tabuleiro e não podem se encontrar (pois, neste caso, ambos "desapareceriam" ou só um restaria, pois uma célula não pode ter mais de um estado). O uso de números complexos é nada mais que uma *gambiarra*: desta maneira foi possível fazer com que cada célula tivesse um valor de *terreno* (ruas, casas, água, fundo) e outro de *agente* (inspetor e mosquito), de modo a um não sobrescrever o outro. Em geral, não recomendo usar nenhum modelo que dependa de agentes que movem pelo tabuleiro.

Esta regra foi criada para ser usada com layouts de cidades criadas em algum programa de edição de imagens e carregadas como estado inicial. Experimente usar os dois layouts inclusos: `city.png` e `city2.png`.

Você também pode facilmente criar seus próprios layouts. Basta usar um programa de imagens e colorir pixel por pixel de uma imagem com as cores *exatas* que correspondem aos diferentes estados das células. Depois, você pode redimensionar a imagem para facilitar a visualização, mas atente-se ao *cell_size* – ele deve ser redimensionado pelo mesmo fator.

[ ]:
```
from random import choice, random, seed

import numpy as np
from tomato.classes import cell

class DengueCell(cell.CellTemplate):
    # mosquito: x+0j ... x+3j -> white (real: on top of, complex: direction)
    # house: 2.0 ... 3.0 -> green (healthy) to red (infected)
    # water: 4.0 .. 5.0 -> blue (stirred) to cyan (still)
    # inspector: x+4j ... x+7j -> magenta (real: on top of, complex: direction)
    # street: 7.0 -> grey
    # background/walls: 8 -> black
```

```
    def __init__(self, val, pos, cell_args):
        # {{{
        global move

        self.value = val
        self.lin, self.col = pos

        seed(cell_args.get("seed", None))
        DengueCell.heal_time = cell_args.get("heal_time", 200)
        DengueCell.mosquito_lifespan = cell_args.get("mosquito_lifespan", 90)
        DengueCell.water_still_factor = cell_args.get("water_still_factor", 0.
↪001)
        DengueCell.spawn_prob = cell_args.get("spawn_prob", 0.001)
        DengueCell.infection_prob = cell_args.get("infection_prob", 1)
        DengueCell.kamikaze_mosquito = cell_args.get("kamikaze_mosquito", False)
        DengueCell.death_prob = cell_args.get("death_prob", 0.0)

        move = True

    # }}}

    @classmethod
    def generation_start(cls, cell_args):
        # {{{
        # All this does is alternate the "move" flag at the start of each␣
↪generation.
        global move

        move = not move

    # }}}

    def update(self, state_matrix):
        # {{{
        global move
        self.state_matrix = state_matrix

        if not move:
            if self.imagv < 8:  # a nosquito or an inspector
                if self.imagv < 4:  # mosquito
                    self.imagv = self.mosquito_next_step() + np.modf(self.
↪imagv)[0]

                    self.imagv += 0.01
                    if (  # mosquito dies
                        np.around(100 * np.modf(self.imagv)[0], 3)
```

```
                        == DengueCell.mosquito_lifespan
                    ):
                        self.imagv = 8

                    # mosquito infects susceptible house
                    if np.isclose(self.value.real, 2.0):
                        if random() < DengueCell.infection_prob:
                            if random() < DengueCell.death_prob:  # house dies
                                self.realv = 8
                            else:
                                self.realv = 3.0
                            if DengueCell.kamikaze_mosquito:
                                self.imagv = 8

                else:  # inspector
                    self.imagv = self.inspector_next_step()

        else:
            if self.imagv < 8:  # a mosquito or an inspector
                self.imagv = 8

            if 2.0 < self.realv <= 3.0:
                # house slowly recovers
                self.realv = self.realv - 1 / DengueCell.heal_time
                if self.realv < 2.0:
                    self.realv = 2.0

            if 4.0 <= self.realv <= 5.0:
                # water diffuses its movement, gradually becoming still
                self.water_diffusion()
                if self.realv == 5.0:
                    if random() < DengueCell.spawn_prob:
                        self.imagv = choice((0, 1, 2, 3))

            for idx, val in enumerate(self.neighbors):
                # inspector in the neighborhood
                if 4 <= val.imag < 8:
                    if 4.0 <= self.realv <= 5.0:
                        self.realv = 4.0

                    # inspector will kill the mosquito if they land on the same␣
↪cell
                    if idx == int(val.imag + 2) % 4:
                        self.imagv = val.imag
                        break
                elif val.imag < 4:
                    if idx == int(val.imag + 2) % 4:
```

```
                    self.imagv = val.imag

    # }}}

    # Display and from display{{{
    @staticmethod
    def display(value):
        # {{{
        if value.imag < 4:  # mosquito
            return (255, 255, 255)
        elif value.imag < 8:  # inspector
            return (255, 0, 255)
        elif value.real == 8.0:  # background
            return (0, 0, 0)
        elif value.real == 7.0:  # street
            return (128, 128, 128)
        elif 2.0 <= value.real <= 3.0:  # house
            scaled_value = (value.real - 2) * 255
            return (scaled_value, 255 - scaled_value, 0)
        elif 4.0 <= value.real <= 5.0:  # water
            scaled_value = (value.real - 4) * 255
            return (0, scaled_value, 255)

    # }}}

    @staticmethod
    def from_display(value):
        # {{{
        if (value == (0, 0, 0)).all():  # background
            return 8 + 8j
        elif (value == (128, 128, 128)).all():  # street
            return 7 + 8j
        elif (value == (255, 0, 255)).all():  # inspector
            return 7 + 4j
        elif (value == (255, 255, 255)).all():  # mosquito
            return 8 + choice((0, 1, 2, 3)) * 1j
        elif value[2] == 255:  # water
            return ((value[1] / 255) + 4) + 8j
        else:  # house
            return ((value[0] / 255) + 2) + 8j

    # }}}
    # }}}

    # Neighborhoods{{{
    @property
    def neighbors(self):
```

```
        # {{{
        #   0
        # 3 w 1
        #   2
        return self.neumann_neighborhood

    # }}}

    @property
    def second_neighbors(self):
        # {{{
        #      0
        #
        # 3    w    1
        #
        #      2
        lin, col = self.lin, self.col

        state_matrix = self.state_matrix
        m, n = state_matrix.shape
        lin, col = self.lin, self.col
        prev_lin, next_lin = lin - 2, lin + 2
        prev_col, next_col = col - 2, col + 2
        try:
            neighbors = [
                state_matrix[prev_lin, col],
                state_matrix[lin, next_col],
                state_matrix[next_lin, col],
                state_matrix[lin, prev_col],
            ]
        except IndexError:
            prev_lin %= m
            next_lin %= m
            prev_col %= n
            next_col %= n

            neighbors = [
                state_matrix[prev_lin, col],
                state_matrix[lin, next_col],
                state_matrix[next_lin, col],
                state_matrix[lin, prev_col],
            ]
        return neighbors

    # }}}

    @property
```

```
    def diagonal_neighbors(self):
        # {{{
        # 0   1
        #   w
        # 3   2
        lin, col = self.lin, self.col

        state_matrix = self.state_matrix
        m, n = state_matrix.shape
        lin, col = self.lin, self.col
        prev_lin, next_lin = lin - 1, lin + 1
        prev_col, next_col = col - 1, col + 1
        try:
            neighbors = [
                state_matrix[prev_lin, prev_col],
                state_matrix[prev_lin, next_col],
                state_matrix[next_lin, next_col],
                state_matrix[next_lin, prev_col],
            ]
        except IndexError:
            prev_lin %= m
            next_lin %= m
            prev_col %= n
            next_col %= n

            neighbors = [
                state_matrix[prev_lin, prev_col],
                state_matrix[prev_lin, next_col],
                state_matrix[next_lin, next_col],
                state_matrix[next_lin, prev_col],
            ]
        return neighbors

    # }}}
    # }}}

    # Convenience methods for complex number manipulation{{{
    @property
    def imagv(self):
        return self.value.imag

    @imagv.setter
    def imagv(self, val):
        self.value = self.value.real + val * 1j

    @property
    def realv(self):
```

```
        return self.value.real

    @realv.setter
    def realv(self, val):
        self.value = val + self.value.imag * 1j

    # }}}

    # 'Next step' functions{{{
    def mosquito_next_step(self):
        # {{{
        # This function chooses the next value (and therefore movement␣
↪direction) for
        # the random walker. it has to take into account all cells within 1 and␣
↪2 units
        # of Neumann distance to make sure it won't choose a move that will␣
↪result in it
        # landing on the same cell as another random walker.

        free_second_neighbors = set(
            x for x in range(4) if self.second_neighbors[x].imag >= 4
        )
        free_diag_neighbors = set(
            x
            for x in range(4)
            if self.diagonal_neighbors[x].imag >= 4
            and self.diagonal_neighbors[(x + 1) % 4].imag >= 4
        )
        free_neighbors = set(x for x in range(4) if self.neighbors[x].imag >= 4)
        try:
            chosen = choice(
                tuple(
                    free_neighbors.intersection(
                        free_second_neighbors, free_diag_neighbors
                    )
                )
            )
        except IndexError:
            # This will happen if the intersection of all "free_neighbors" sets␣
↪is
            # empty, which means any move may result in it bonking another␣
↪walker. In
            # this case it just chooses a random direction.
            return choice((0, 1, 2, 3))
        return chosen

    # }}}
```

```
    def inspector_next_step(self):
        # {{{
        # inspectors can only walk on roads, and always turn left at intersections

        if self.relative_neighbor(0).real != 7:
            for idx in range(4):
                if self.relative_neighbor(idx).real == 7:
                    return (self.imagv + idx) % 4 + 4
        elif self.relative_neighbor(3).real == 7:
            return (self.imagv + 3) % 4 + 4
        else:
            return self.imagv

    # }}}
    # }}}

    # Miscellaneous{{{
    def water_diffusion(self):
        # {{{
        water_neighbors = [
            0.25 * (x.real - 4) for x in self.neighbors if 4.0 <= x.real <= 5.0
        ]
        self.realv -= 4
        self.realv = np.power(0.75, len(water_neighbors)) * self.realv + sum(
            water_neighbors
        )
        self.realv = self.realv + 4 + DengueCell.water_still_factor
        if self.realv > 5.0:
            self.realv = 5.0

    # }}}

    def dir_to_index(self, direc):
        return int(self.imagv - 4 + direc) % 4

    def relative_neighbor(self, direc):
        # {{{
        # return the value of a neighbor found at a direction relative to the cell's
        # current direction
        # 0 - front; 1 - right; 2 - back; 3 - left
        return self.neighbors[self.dir_to_index(direc)]

    # }}}
```

```
    def relative_diag_neighbor(self, direc):
        # {{{
        # return the value of a diagonal neighbor found at a direction relative␣
 ↪to the
        # cell's current direction
        # 0 - front-left; 1 - front-right; 2 - back-right; 3 - back-left
        return self.diagonal_neighbors[self.dir_to_index(direc)]

    # }}}


# }}}
```

[ ]:
```
import numpy as np
import tomato as tt

rule = DengueCell

# Use os cell_args para alterar parâmetros no modelo
cell_args = {
    "seed": 1,  # semente para o gerador de números aleatórios
    "heal_time": 200,
    "mosquito_lifespan": 90,
    "water_still_factor": 0.001,
    "spawn_prob": 0.01,
    "infection_prob": 0.5,
    "kamikaze_mosquito": True,
    "death_prob": 0.1,
}
cell_size = 16
board = tt.Board(rule, cell_size=cell_size)

# Teste também com o city2.png
board.start("city.png", cell_args=cell_args, inline=True)
```

4 Análise quantitativa de um AC

Até agora vimos muitos exemplos de regras sendo executadas em uma animação, pela qual podemos fazer uma análise **qualitativa** do modelo. Para fazer uma análise **quantitativa**, nós manualmente executamos os métodos `load_state` e `update` da classe `Board` em um loop, ao invés de deixar que o método `start` faça isso por nós.

Desta forma, podemos inserir código para salvar alguma parâmetro da simulação que varia com as gerações em listas e plotar gráficos a partir delas.

No exemplo a seguir, vamos estudar 200 gerações da regra Maze of Chaos, salvando um gif dessa simulação e gerando uns gráficos.

[ ]:
```
from tomato.classes import cell

# Checa se a condição de conversão é satisfeita para um dado número
# de vizinhos. Está aí para facilitar mudar este número de uma só
# vez em todo o código.
def conversion_condition(num_neighbors):
    return num_neighbors > 2

# Checa se a condição de conversão é satisfeita para um dado número
# de vizinhos. Está aí para facilitar mudar este número de uma só
# vez em todo o código.
def birth_condition(num_neighbors):
    return num_neighbors > 2

class ChaosMazeCell(cell.CellTemplate):
    # {{{

    # branco = 1
    # vermelho = 2
    # verde = 3
    # morto = 0 ou maior que 3

    def update(self, state_matrix):
        self.state_matrix = state_matrix

        if self.value == 3:
            if conversion_condition(self.live_red_neighbors):
                self.value = 2
            elif conversion_condition(self.live_green_neighbors):
                self.value = 3
            elif conversion_condition(self.live_white_neighbors):
                self.value = 1
        elif self.value == 1:
            if conversion_condition(self.live_green_neighbors):
                self.value = 3
            elif conversion_condition(self.live_white_neighbors):
                self.value = 1
            elif conversion_condition(self.live_red_neighbors):
                self.value = 2
        elif self.value == 2:
            if conversion_condition(self.live_white_neighbors):
                self.value = 1
            elif conversion_condition(self.live_red_neighbors):
                self.value = 2
            elif conversion_condition(self.live_green_neighbors):
                self.value = 3
        else:
```

```
            if birth_condition(self.live_white_neighbors):
                self.value = 1
            elif birth_condition(self.live_red_neighbors):
                self.value = 2
            elif birth_condition(self.live_green_neighbors):
                self.value = 3


    @property
    def neighbors(self):
        # Teste outras vizinhanças também!
        return self.moore_neighborhood

    @property
    def live_neighbors(self):
        return (
            self.live_white_neighbors,
            self.live_red_neighbors,
            self.live_green_neighbors,
        )

    @property
    def live_white_neighbors(self):
        return self.neighbors.count(1)

    @property
    def live_green_neighbors(self):
        return self.neighbors.count(3)

    @property
    def live_red_neighbors(self):
        return self.neighbors.count(2)

    @staticmethod
    def display(value):
        if value == 1:
            return (255, 255, 255)
        elif value == 2:
            return (255, 0, 0)
        elif value == 3:
            return (0, 255, 0)
        else:
            return (0, 0, 0)

    @staticmethod
    def from_display(value):
        if (value == (255, 255, 255)).all():
```

```
            return 1
        elif (value == (255, 0, 0)).all():
            return 2
        elif (value == (0, 255, 0)).all():
            return 3
        else:
            return 0


# }}}
```

4.1 Parâmetros iniciais

Para fins de praticidade, vamos separar o módulo de simulação em algumas células.

Nesta primeira, vamos somente definir os **parâmetros** da simulação, inclusive a *matriz de estados iniciais*:

[ ]:
```
import tomato as tt

# Vamos fixar uma semente para o Gerador de Números Aleatórios desta vez
from numpy.random import default_rng
RNG = default_rng(8624)

rule = ChaosMazeCell

board_dimensions = (120, 120)

cell_size = 4

# Probabilidades de uma célula morta ou viva (respectivamente) serem␣
 ↪selecionadas
# em cada elemento da matriz de estados iniciais. Tente variar estes parâmetros.
prob_dead = 0.95
prob_live = (1 - prob_dead) / 3

initial_state_matrix = RNG.choice(
    (0, 1, 2, 3), size=board_dimensions, p=(prob_dead, prob_live, prob_live,␣
 ↪prob_live)
)

# número de gerações da simulação
max_generations = 300
```

4.2 Exibindo os ACs e salvando um gif

Primeiro vamos ver 200 gerações da simulação, para ter uma noção visual do que estamos estudando, e salvar um gif dessas gerações.

```
[ ]: board = tt.Board(rule, cell_size=cell_size)
     board.start(
         initial_state_matrix,
         inline=True,
         generations=max_generations, # Para criar o gif, é necessário limitear o␣
      ↪número de gerações da simulação
         generate_figures=True, # Criar uma pasta e salvar um png de cada geração
         generate_figures_dir="maze-of-chaos-simulation", # Especificar o nome da␣
      ↪pasta
         generate_gif=True # Criar um gif com as imagens ao final da execução
     )
```

4.3 Medindo parâmetros da simulação

Agora vamos carregar o estado e iterar a simulação *manualmente*, usando os métodos `board.load_state` e `board.update` dentro de um loop `While`. Desta forma, a simulação não será exibida, mas podemos encaixar código para coleta e análise de dados entre cada iteração (ou seja, entre cada `board.update`).

```
[ ]: import numpy as np

     # Criando outra Board e carregando o estado inicial
     board = tt.Board(rule, cell_size=cell_size)
     board.load_state(initial_state_matrix)

     # Criando vetores numpy para armazenar as populações de cada espécie de célula
     white_pop = np.zeros(max_generations)
     red_pop = np.zeros(max_generations)
     green_pop = np.zeros(max_generations)

     while board.generation < max_generations:
         # Matriz com os estados de cada célula:
         state_matrix = board.state_matrix

         # Contando o número de ocorrências de cada estado do AC na matriz e␣
      ↪armazenando
         # nos vetores apropriados.
         # Lembrando que neste modelo, (branco, vermelho, verde) = (1, 2, 3):
         white_pop[board.generation] = np.count_nonzero(state_matrix == 1)
         red_pop[board.generation] = np.count_nonzero(state_matrix == 2)
         green_pop[board.generation] = np.count_nonzero(state_matrix == 3)

         # Iterar a simulação depois de terminada a nossa coleta de dados da geração
         board.update()

     # Avisando que a simulação acabou e fazendo umas estatísticas básicas
     print("Prontinho!")
```

```
print(f"white_pop: initial {white_pop[0]} | final {white_pop[-1]} | avg {np.
 ↪mean(white_pop)}")
print(f"red_pop: initial {red_pop[0]} | final {red_pop[-1]} | avg {np.
 ↪mean(red_pop)}")
print(f"green_pop: initial {green_pop[0]} | final {green_pop[-1]} | avg {np.
 ↪mean(green_pop)}")
```

4.4 Plotando gráficos

Por fim, vamos plotar uns gráficos com os dados que obtivemos.

[ ]:
```
from matplotlib import pyplot as plt

fig, ax = plt.subplots()
ax.plot(white_pop, color="gray", label="brancas")
ax.plot(red_pop, color="red", label="vermelhas")
ax.plot(green_pop, color="green", label="verdes")

ax.set_xlabel("Geração")
ax.set_ylabel("População")
ax.legend()

# Lembre-se que muitas vezes é interessante ver o gráfico nas escalas log e␣
 ↪linear
# ax.set_yscale("log", base=2)
```

5 Novidades 29/06

5.1 Implementação de Agentes

Quando escrevi a primeira versão deste notebook, usei uma versão do tomato-engine que não suportava direito um tipo peculiar, mas ao mesmo tempo muito comum e útil, de autômato: o **Agente**.

A diferença do Agente para uma célula "normal" de um autômato celular é que o agente pode **mudar de posição** no tabuleiro. A implementação de agentes era possível mas muito complicada no tomato-engine, vide as regras *Dengue* e a versão antiga da *Formiga de Langton* acima.

Agora é possível implementar agentes de maneira muito mais simples no tomato-engine, por meio da classe **Agent**. Essa classe possui os mesmos métodos e atributos da classe Cell, exceto que suas instâncias (ou seja, os agentes) são armazenados numa **lista** (chamada `state_list`) ao invés de uma matriz, permitindo assim mudar livremente sua posição em linhas e colunas do tabuleiro. A maneira de implementar agentes está apresentada nos exemplos *Random Walk* e *Colorful Random Walk*, e na *Formiga de Langton*, que foi atualizada para fazer uso destas novas funcionalidades. No futuro pretendo atualizar o modelo *Dengue* e o *Turmites* também.

Essa atualização inclui outras mudanças no tomato-engine, que foram necessárias para incorporar as novas funcionalidades. Alguns exemplos de regras tiveram que ter o nome de algumas variáveis alterados.

Lembre-se de executar a célula de *Instalação* novamente para atualizar o tomato-engine.

[ ]: