Aplicação: autômatos celulares

Modelos de autômatos celulares definem ferramentas computacionais muito importantes para a simulação e estudo de sistemas complexos. Um sistema é dito complexo quando suas propriedades não são uma consequência natural de seus elementos constituintes vistos isoladamente. As propriedades emergentes de um sistema complexo decorrem em grande parte da relação não-linear entre as partes. Costuma-se dizer de um sistema complexo que o todo é mais que a soma das partes. Exemplos de sistemas complexos incluem redes sociais, colônias de animais, o clima e a economia.

Uma pergunta recorrente no estudo de tais sistemas é: porque e como padrões complexos emergem a partir da interação entre os elementos? Como esses padrões evoluem com o tempo? Respostas a essas perguntas não são totalmente conhecidas, mas o estudo de modelos de autômatos celulares nos auxiliam no estudo e análise de tais sistemas. Aplicações práticas são muitas e incluem:
- Autômatos celulares e composição musical
- Autômatos celulares e modelagem urbana
- Autômatos celulares e propagação de epidemias
- Autômatos celulares e crescimento de câncer

Os primeiros modelos de autômatos celulares foram propostos originalmente na década de 40 por John Von Neumann e tinham como objetivos principais:
- Representar matematicamente a evolução natural em sistemas complexos
- Desenvolver máquinas de auto-replicação, através de um conjunto de regras matemáticas objetivas
- Estudar a auto-organização em sistemas complexos

Segundo Wolfram, autômatos celulares são formados por uma rede de células que possuem seus estados alterados num tempo disscreto de acordo com seu estado anterior e o estado de suas células vizinhas. Algumas características importantes e comuns a todos os autômatos celulares são:
- Homogeneidade: as regras são iguais para todas as células
- Estados discretos: cada célula pode estar em um dos finitos estados
- Interações locais: o estado de uma célula depende apenas das células mais próximas (vizinhas)
- Processo dinâmico: a cada instante de tempo as células podem sofrer uma atualização de estado

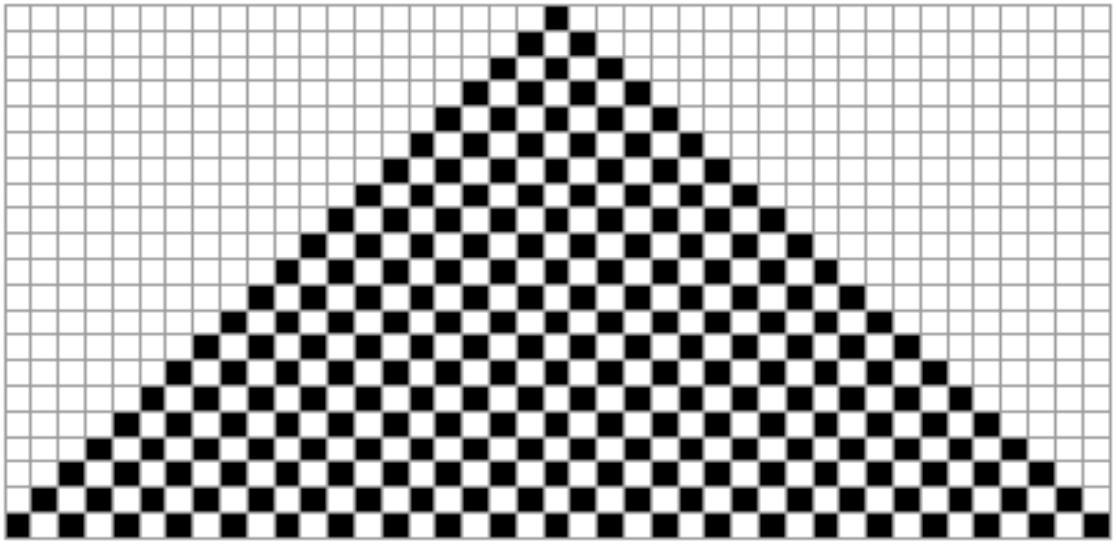
Def: Um autômatos celular é definido por uma 5-tupla de elementos

$A = (R, S, S_0, V, F)$ onde

R é a grade de células (pode ser 1D, 2D, 3D,…)
S é o conjunto de estados de uma célula c_i
S_0 é o estado inicial do sistema
V é conjunto vizinhança (define quem são os vizinhos de cada célula)
F é a função de transição (regras de governam a evolução do sistema no tempo)

Autômatos Celulares Elementares

Um autômato celular é dito elementar se o reticulado de células R é unidimensional (ou seja, pode ser representado por um vetor), o conjunto de estados S = {0, 1} e o conjunto vizinhança engloba apenas duas células: a anterior (i-1) e a posterior (i+1). Tipicamente, se uma célula assume estado 0 dizemos que ela está morta e se ela assume estado 1 dizemos que está viva. A figura a seguir ilustra as primeiras 20 gerações de um autômato celular elementar, em que no início apenas uma célula está viva (preto = vivo, branco = morto)

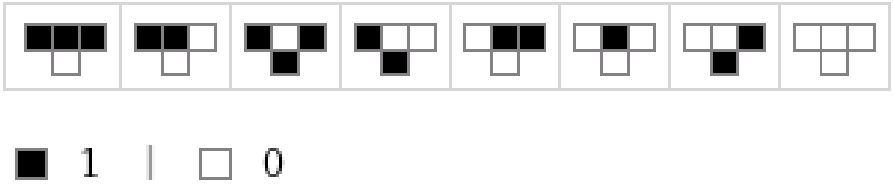


A questão é: como evoluir uma configuração de forma a construir esse padrão? Que regras são aplicadas para definir quais células vivem ou morrem na próxima geração?

A função de transição do autômato da figura é dada pela seguinte tabela.

$x_t(i-1)$	$x_t(i)$	$x_t(i+1)$	$x_{t+1}(i)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Uma forma de resumir toda essa tabela é através da seguinte representação:



O que essa regra nos diz pode ser sumarizado em 8 fatos:
1) sempre que a célula i for morta e a (i-1) e (i+1) também forem, a célula i permanecerá morta na próxima geração.
2) sempre que a célula i for morta, a (i-1) for morta e a (i+1) for viva, a célula i viverá na próxima geração.
3) sempre que a célula i for viva e a (i-1) e a (i+1) forem mortas, a célula i morrerá na próxima geração.
4) sempre que a célula i for viva, a (i-1) for morta e a (i+1) for viva, a célula i morrerá na próxima geração.

5) sempre que a célula i for morta, a (i-1) for viva e a (i+1) for morta, a célula i viverá na próxima geração.
6) sempre que a célula i for morta e ambas (i-1) e (i+1) forem vivas, a célula i viverá na próxima geração.
7) sempre que a célula i for viva, a (i-1) for viva e a (i+1) for morta, a célula i morrerá na próxima geração.
8) sempre que a célula i for viva e ambas (i-1) e (i+1) forem vivas, a célula i morrerá na próxima geração.

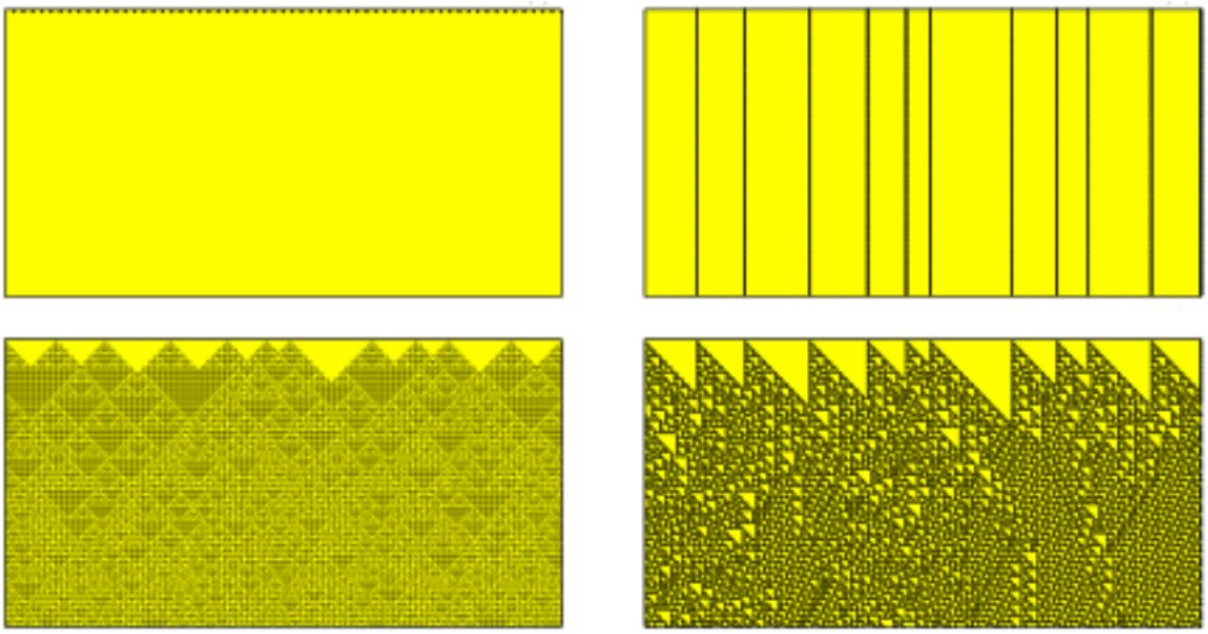
Note que não existem mais combinações possíveis de 0's e 1's usando apenas 3 bits, pois conseguimos contar em binário de 0 a 7, o que resulta em 8 possibilidades. Essa regra tem um nome: é a regra 50, pois o número binário correspondente a última coluna da função de transição vale 00110010, que em binário é justamente o número 50. Sendo assim, quantas possíveis regras existem para um autômato celular elementar? Basta computar 2^8, que resulta em 256. Portanto, o número total de regras distintas é 256. Por essa razão dizemos que existem 256 autômatos celulares elementares distintos, um para cada regra. O interessante é estudar e simular o que acontece com cada um desses autômatos durante sua evolução. De acordo com Wolfram, existem 4 classes de regras para um autômato celular elementar:

- Classe 1: Estado Homogêneo
  Todas as células chegarão num mesmo estado após um número finito de estados

- Classe 2: Estável simples
  As células não possuirão todas o mesmo estado, mas eles se repetem com a evolução temporal

- Classe 3: Padrão irregular
  Não possui padrão reconhecível

- Classe 4: Estrutura complexa
  Estruturas complexas que evoluem imprevisivelmente



As regras mais interessantes são as da classe 4, pois definem um sistema complexo com propriedades dinâmicas interessantes, sendo algumas delas capazes até de simular máquinas de Turing, que são modelos computacionais capazes de serem programadas para realizar diferentes tarefas computacionais. Um exemplo de regra com essa característica é a regra 110. (Referências: https://en.wikipedia.org/wiki/Rule_110, http://www.complex-systems.com/pdf/15-1-1.pdf)

Exercício: Construa a função de transição do autômato celular elementar definido pela regra 30. Aplique a regra para evoluir a condição inicial idêntica a da figura da regra 50 (apenas uma célula viva) por 20 gerações. Repita o exercício mas agora para a regra 110.

A seguir é apresentado um algoritmo para a simulação de autômatos celulares elementares.

```
geração = vetor(N) (N é o número de células)

nova_geração = vetor(N)

evolução = matriz(MAX, N) (MAX é o número de gerações)

Inicializar geração (setar configuração inicial)

para i = 1 até MAX
     evolução[i,:] = geração
     # Percorre cada célula da geração atual
     para j = 1 até N
          Aplicar regra de transição na célula j, gerando nova_geração
     geração = nova_geração

Plotar resultados
```

Ex: Baseado no algoritmo anterior, implementar um script em Python que, dado uma regra (número de 0 a 255), evolua uma configuração inicial de tamanho N = 1000 até a geração 500.

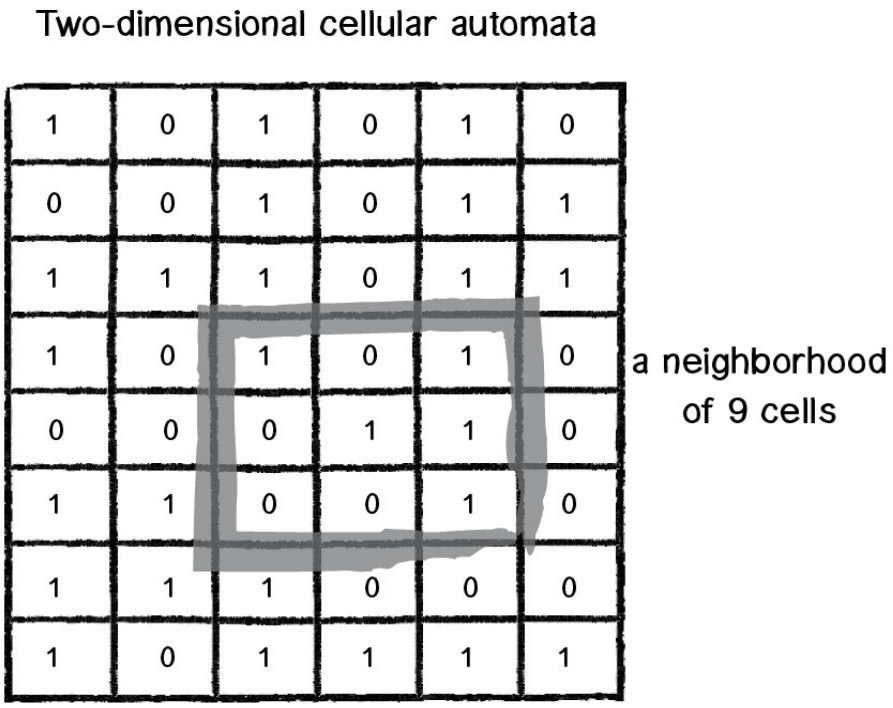
```
import numpy as np
import matplotlib.pyplot as plt

# converte um número inteiro para sua representação binária (0-255)
def converte_binario(numero):
     binario = bin(numero)
     binario = binario[2:]
     if len(binario) < 8:
          zeros = [0]*(8-len(binario))
     binario = zeros + list(binario)
     return list(binario)


# Início do script
MAX = 500
g = np.zeros(1000)
ng = np.zeros(1000)

regra = int(input('Entre com o número da regra: '))
codigo = converte_binario(regra)
```

```
# Matriz em que cada linha armazena uma geraçao do autômato
matriz_evolucao = np.zeros((MAX, len(g)))

# Define geração inicial
g[len(g)//2] = 1

# Laço principal: atualiza as gerações
for i in range(MAX):

     matriz_evolucao[i,:] = g

     # Percorre células da geração atual
     for j in range(len(g)):

          if (g[j-1] == 0 and g[j] == 0 and g[(j+1)%len(g)] == 0):
               ng[j] = int(codigo[7])
          elif (g[j-1] == 0 and g[j] == 0 and g[(j+1)%len(g)] == 1):
               ng[j] = int(codigo[6])
          elif (g[j-1] == 0 and g[j] == 1 and g[(j+1)%len(g)] == 0):
               ng[j] = int(codigo[5])
          elif (g[j-1] == 0 and g[j] == 1 and g[(j+1)%len(g)] == 1):
               ng[j] = int(codigo[4])
          elif (g[j-1] == 1 and g[j] == 0 and g[(j+1)%len(g)] == 0):
               ng[j] = int(codigo[3])
          elif (g[j-1] == 1 and g[j] == 0 and g[(j+1)%len(g)] == 1):
               ng[j] = int(codigo[2])
          elif (g[j-1] == 1 and g[j] == 1 and g[(j+1)%len(g)] == 0):
               ng[j] = int(codigo[1])
          elif (g[j-1] == 1 and g[j] == 1 and g[(j+1)%len(g)] == 1):
               ng[j] = int(codigo[0])

     g = ng.copy()  # se não usar copy ambos vetores tornam-se o mesmo

# plota matriz resultante como imagem
plt.figure(1)
plt.axis('off')
plt.imshow(matriz_evolucao, cmap='gray')
plt.savefig('Automata.png', dpi=300)
plt.show()
```

Autômatos celulares 2D

O Jogo da Vida

O autômato 2D mais conhecido sem dúvida é o jogo da vida, criado por Conway para simular a evolução de sistemas complexos a partir de regras determinísticas. O reticulado 2D é representado computacionalmente por uma matriz geralmente quadrada de células que podem estar vivas ou mortas. A função de vizinhança é definida pela vizinhança de Moore, ou seja, pelas 8 células mais próximas a uma dada célula i, conforme ilustra a figura a seguir.
A função de transição do jogo da vida tem como conceito imitar processos de nascimento de morte. A ideia básica é que um ser vivo necessita de outros seres vivos para sobreviver e procriar, mas um excesso de densidade populacional provoca a morte do ser vivo devido à escassez de recursos.



São 4 regras básicas:

R1 (Sobrevivência) – uma célula viva com 2 ou 3 células vizinhas vivas, permanece viva na próxima geração.

R2 (Morte por isolamento) – uma célula viva com 0 ou 1 vizinho vivo morre de solidão na próxima geração.

R3 (Morte por sufocamento) – uma célula viva co 4 ou mais vizinhos vivos morre por sufocamento na próxima geração.

R4 (Renascimento) – uma célula morta com exatamente 3 vizinhos vivos, renasce na próxima geração.

Isso nos leva a regra de transição conhecida como B3S23, uma vez que no código proposto B significa born (nascer) e S significa survive (sobreviver). Em outras palavras, nesse autômato em particular, uma célula nasce sempre que possui 3 vizinhas vivas ao redor e sobrevive se possui 2 ou 3 vizinhas vivas ao redor. Outras variantes de regras incluem: B15S257, B147S256, B34S4567, etc. Cada uma das regras define um autômato diferente. Ao autômato cuja regra é B3S23 dá-se o nome de Jogo da Vida.

É interessante perceber que a regra B3S23 a partir de diversas inicializações simples exibe um comportamento altamente complexo, onde padrões complexos e “seres vivos” passam a interagir de maneira bastante inesperada. Trata-se de um conjunto de regras totalmente determinísticas que levam a um comportamento completamente imprevisível (ordem x caos).

Para uma coletânea de condições iniciais verifique o link:
https://jakevdp.github.io/blog/2013/08/07/conways-game-of-life/

Um problema comum que afeta simulações computacionais do jogo da vida é o chamado problema de valor de contorno. Isso nada mais é que uma falha ao se definir a função de transição para células na borda do sistema. Para se evitar essa indefinição, considera-se que o reticulado é na verdade um toro. Isso significa que não existem bordas, uma vez que a borda da esquerda é ligada a borda da direita, assim como a inferior é ligada a superior.

Ex: Implementar um script em Python que, dada uma configuração inicial, simule o jogo da Vida num tabuleiro de dimensões 100 x 100 por 200 gerações.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import time

# cria inicialização: rpentomino nas coordenadas i, j
def init_config(tabuleiro, padrao, i, j):
     linhas = padrao.shape[0]
     colunas = padrao.shape[1]
     tabuleiro[i:i+linhas, j:j+colunas] = padrao

# Início do script
inicio = time.time()
MAX = 200
SIZE = 100

geracao = np.zeros((SIZE, SIZE))
nova_geracao = np.zeros((SIZE, SIZE))

# Cubo de dados em que cada fatia representa uma geração
matriz_evolucao = np.zeros((SIZE, SIZE, MAX))

# Define geração inicial

unbounded = np.array([[1, 1, 1, 0, 1],
                      [1, 0, 0, 0, 0],
                      [0, 0, 0, 1, 1],
                      [0, 1, 1, 0, 1],
                      [1, 0, 1, 0, 1]])

glider = np.array([[1, 0, 0],
          [0, 1, 1],
          [1, 1, 0]])

r_pentomino = np.array([[0, 1, 1],
               [1, 1, 0],
               [0, 1, 0]])

diehard = np.array([[0, 0, 0, 0, 0, 0, 1, 0],
           [1, 1, 0, 0, 0, 0, 0, 0],
           [0, 1, 0, 0, 0, 1, 1, 1]])

acorn = np.array([[0, 1, 0, 0, 0, 0, 0],
         [0, 0, 0, 1, 0, 0, 0],
         [1, 1, 0, 0, 1, 1, 1]])

init_config(geracao, r_pentomino, SIZE//2, SIZE//2)
```

```
# Laço principal (atualiza gerações)
for k in range(MAX):

     print('Processando geração %d...' %k)
     matriz_evolucao[:,:,k] = geracao

     # Laço principal: atualiza as gerações
     for i in range(SIZE):

          for j in range(SIZE):

               vivos = geracao[i-1, j-1] + geracao[i-1, j] + \
               geracao[i-1, (j+1)%SIZE] + geracao[i, j-1] + \
               geracao[i, (j+1)%SIZE] + geracao[(i+1)%SIZE, j-1] + \
               geracao[(i+1)%SIZE, j] + geracao[(i+1)%SIZE, (j+1)%SIZE]

               if (geracao[i,j] == 1):
                    if (vivos == 2 or vivos == 3):
                         nova_geracao[i,j] = 1
                    else:
                         nova_geracao[i,j] = 0
               else:
                    if (vivos == 3):
                         nova_geracao[i,j] = 1

     geracao = nova_geracao.copy()

fim = time.time()
print('Tempo gasto na simulação: %.2f s' %(fim-inicio))

# Gera animação da evolução do sistema
fig = plt.figure(1)
plt.axis('off')
lista = []
for i in range(MAX):
     im = plt.imshow(matriz_evolucao[:,:,i], cmap='gray')
     lista.append([im])

ani  =  animation.ArtistAnimation(fig,  lista,  interval=100,  blit=True,
repeat_delay=1000)

ani.save('r_pentomino.gif', writer='imagemagick')

plt.show()
```

"Fear has a large shadow, but he himself is small."
(Ruth Gendler)