Relatório Problema 2: O Jogo da Vida de John Conway

Gustavo Silva Ribeiro

[1]Bacharelado de Engenharia de Computação

Universidade Estadual de Feira de Santana (UEFS)

Av. Transnordestina, s/n, Novo Horizonte, Feira de Santana – BA, Brasil – 44036-900

tavoribeiro007@gmail.com

Resumo. *Este trabalho apresenta o desenvolvimento de uma aplicação interativa baseada no Jogo da Vida de John Conway. Implementado em Python e executado no terminal, o jogo permite ao usuário definir posições iniciais por meio de um cursor, simular gerações com base nas regras clássicas do autômato celular e encerrar automaticamente ao detectar padrões estáticos, extermínio ou oscilações. O código prioriza a clareza, simplicidade e fidelidade ao comportamento original do modelo matemático de Conway.*

1. Introdução

O Jogo da Vida de Conway é um autômato celular criado por John Conway em 1970 [1]. Trata-se de um sistema discreto e determinístico que simula a evolução de padrões em uma grade com base em regras simples de sobrevivência, nascimento e morte. Este projeto teve como objetivo desenvolver uma versão textual e interativa do jogo, utilizando conceitos de controle de fluxo, listas bidimensionais e manipulação de entrada no terminal. A proposta foi permitir não apenas a simulação do comportamento dos padrões, mas também sua terminação automática com base em comportamentos estáticos ou oscilatórios. Conforme destacado por Oliveira (2020) [5], o Jogo da Vida de Conway é um exemplo notável de como regras simples podem levar a comportamentos complexos, sendo uma ferramenta valiosa para o estudo de sistemas dinâmicos e computação.

2. Metodologia

O projeto foi desenvolvido utilizando Python e executado exclusivamente via terminal, com foco na experiência interativa do usuário e na fidelidade às regras do Jogo da Vida (de John Conway). A primeira decisão foi estruturar o tabuleiro como uma lista de listas, com dimensões fixas de 10x10. A representação de células vivas e mortas foi deixada a cargo do usuário no início da execução, reforçando a interatividade do sistema.

2.1. Questões e soluções

Nesta subseção serão descritos as questões que se mostraram presentes durante as sessões

PBL em sala de aula.

Como fazer uma matriz? Foi criado dois loops para gerar uma lista de lista, garantindo uma estrutura bidimensional .

Como analisar as condições de vivas e mortas? Para cada célula (x, y), conta quantos vizinhos estão vivos e então aplica as regras (Figura 1).

```
def contar_vizinhos_vivos(tabuleiro, x, y):
    deslocamento = [(-1, -1), (-1, 0), (-1, 1),
                    ( 0, -1),          ( 0, 1),
                    ( 1, -1), ( 1, 0), ( 1, 1)]
    c = 0
    for dx, dy in deslocamento:
        nx = x + dx
        ny = y + dy
        if 0 <= nx < 10 and 0 <= ny < 10:
            if tabuleiro[ny][nx] == viva:
                c += 1
    return c
```

Figura 1. Contar vizinhos vivos

Como fazer a simulação do jogo - iniciar e dar sequência às gerações? Estruturei a simulação em três passos principais:

1. Loop principal de gerações;
2. Função que gera a próxima geração;
3. E só então atualiza o histórico de gerações.

2.2. Descrição do Algoritmo

A estrutura do programa segue por três grandes blocos funcionais: a fase de configuração inicial do tabuleiro, a função de evolução das gerações segundo as regras de Conway, e o módulo de verificação de fim de jogo. O algoritmo foi planejado para operar inteiramente em ambiente de terminal.

2.2.1. Módulo de Configuração

Na fase inicial, o tabuleiro é representado como uma matriz 10x10, implementada por uma lista de listas em Python [6]. O usuário informa os símbolos que representarão as células vivas e mortas. O tabuleiro é preenchido com o símbolo das células mortas por padrão, utilizando dois laços aninhados para gerar a matriz.

Para interagir com o tabuleiro, o sistema oferece um cursor controlado por teclas: W (cima), A (esquerda), S (baixo) e D (direita), mapeadas a variáveis que ajustam a posição do cursor em coordenadas (x, y). O cursor é visualmente destacado no terminal usando colchetes ao redor da célula onde se encontra. Com a tecla V, o usuário ativa uma célula viva na posição do cursor. A estrutura condicional garante que a movimentação

fique dentro dos limites da matriz, validando os valores de x e y entre 0 e 9. Essa fase é interativa e visual, permitindo ao usuário configurar manualmente o estado inicial do tabuleiro. O processo é encerrado com a tecla !, que desativa o modo de entrada e inicia a fase de simulação do jogo (Figura 1).

```
def matriz_com_cursor(Matriz, pos_x, pos_y):
    for y in range(10):
        Matriz = []
        print()
        for x in range(10):
            if y == pos_y and x == pos_x:
                Matriz.append(f"[{tabuleiro[y][x]}]")
            else:
                Matriz.append(f"{tabuleiro[y][x]}")
        print(Matriz)
```

Figura 2. Função do cursor do tabuleiro

2.2.2. Módulo de Simulação da Evolução

A segunda parte do algoritmo consiste na simulação interativa das gerações segundo as regras do Jogo da Vida. Para cada célula da matriz, a função de contagem de vizinhos vivos avalia o estado das oito células adjacentes, usando deslocamentos em coordenadas relativas. O estado da próxima geração é então calculado com base nas regras:

- Solidão: com menos de 2 vizinhos vivos morre;
- Sobrevivência: com 2 ou 3 vizinhos vivos sobrevive;
- Superpopulação: com mais de 3 vizinhos vivos morre;
- Reprodução: uma célula morta com mais de 3 vizinhos vivos ganha vida.

O algoritmo cria uma nova matriz representando a próxima geração, baseada nas decisões tomadas individualmente para cada célula da matriz atual. Essa abordagem evita alterações diretas na matriz original durante a iteração, o que poderia comprometer os cálculos de vizinhança. Após o término do ciclo de análise, a nova matriz substitui a anterior, e o tabuleiro é exibido novamente. Para tornar o processo visível ao usuário, o terminal é limpo e o novo estado do tabuleiro é impresso com um intervalo de tempo entre as atualizações (definido com time.sleep [2]). Esse ciclo se repete até que uma das três condições de término seja satisfeita: todas as células estão mortas; o tabuleiro permanece inalterado em relação à geração anterior; ou a configuração atual é idêntica à de duas gerações atrás, indicando uma oscilação.

2.2.3. Módulo da Verificação de Fim de Jogo

A verificação do fim de jogo é um componente essencial para garantir que a simulação não entre em ciclos infinitos ou permaneça ativa sem necessidade. O sistema identifica três condições para encerrar a execução:

1. Se todas as células estão mortas;
2. Se a nova geração é igual à atual (indicando um padrão estático);
3. Se a nova geração é igual à geração anterior à atual, o que caracteriza um padrão oscilador de dois estados.

Assim como na Figura 2, essas verificações são realizadas por uma função que recebe a geração atual, a nova geração, e as duas gerações anteriores. Utiliza-se a função all() [3] para verificar se todas as células estão mortas, e operadores de comparação para detectar repetições.

```python
def finalizar_jogo(tabuleiro, nova_geracao, ultima_geracao, penultima_geracao):
    todas_mortas = all(celula != viva for linha in nova_geracao for celula in linha)
    estatico = nova_geracao == tabuleiro
    oscilador = nova_geracao == penultima_geracao
    return todas_mortas or estatico or oscilador
```

Figura 3. Função da verificação do tabuleiro

As listas ultima_geracao e penultima_geracao são atualizadas a cada ciclo utilizando cópias profundas[7], garantindo que a comparação entre matrizes reflita com precisão o estado do tabuleiro. Caso uma das condições seja satisfeita, o programa imprime uma mensagem de encerramento e a variável de controle do jogo é alterada, interrompendo o laço principal da execução.

Essa lógica garante robustez à simulação, evitando ciclos desnecessários e permitindo que o sistema reaja de maneira automática à estabilidade do ambiente, conforme recomendado por abordagens clássicas de detecção de padrões em mecanismos celulares.

2.3. Ordem de Codificação

Guiada pelas discussões em sala de aula, explorações feitas nas sessões tutoriais. Inicialmente, o foco esteve na compreensão do funcionamento do Jogo da Vida, incluindo suas regras e o comportamento esperado dos padrões clássicos. Esse entendimento foi essencial para definir os requisitos do sistema:

- Tabuleiro 10x10, células vivas e mortas, entrada via terminal;
- Construção e exibição do tabuleiro no terminal.
- Implementação do cursor interativo (comandos WASD e V).
- Função de contagem de vizinhos vivos.
- Aplicação das regras para gerar novas gerações.

- Verificação de fim de jogo.

2.4. Ferramentas

As ferramentas utilizadas para a elaboração do software foram a plataforma IDE (Integrated Development Environment ou Ambiente de Desenvolvimento Integrado) “Visual Studio Code”, o sistema operacional Windows – edição 11 –, e a versão 3.13.3 de 64 bits do Python.

3. Relatórios e Discussões

3.1. Manual de Uso

Requisitos

Recomendado utilizar um terminal que suporte a limpeza de tela por comandos cls (Windows) ou clear (Linux/macOS) para melhor visualização da simulação, além de ter:

- Python na versão 3.13.3 ou superior instalado;
- Ter as bibliotecas padrão: os [4] e time;
- Execução via terminal (cmd, PowerShell ou terminal integrado do VSCode).

Execução do Jogo

Conforme a Figura 3, o jogo inicia com o usuário escolhendo o símbolo para as células vivas e para as mortas:

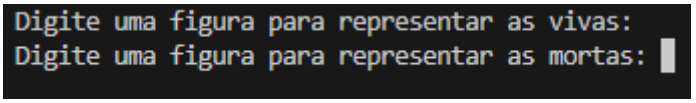

Figura 4. Escolha dos símbolos.

Personalizar Tabuleiro

Em seguida, o usuário poderá escolher pelo cursor (w [cima], a [esquerda], s [baixo] e d [direita],) onde as células vivas inicializaram, pressionando v para substituir a célula morta pela viva e ! para iniciar o jogo (Figura 4).

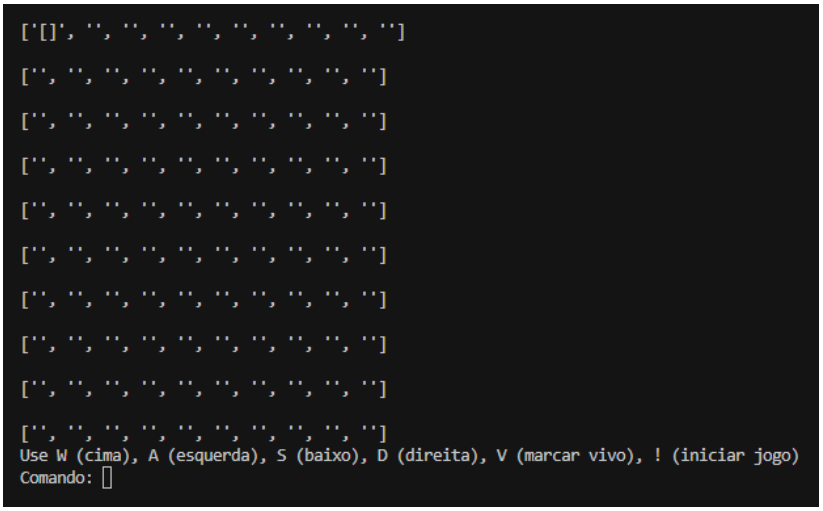

Figura 5. Menu de modificação das células

Encerrar Jogo

A saída do programa consiste na visualização do tabuleiro em cada geração. Ao detectar condições de fim de jogo (todas as células mortas, estado estático ou padrão oscilante), o sistema exibe uma mensagem informando que o jogo foi encerrado por uma dessas causas e interrompe automaticamente a execução, como na figura 5.

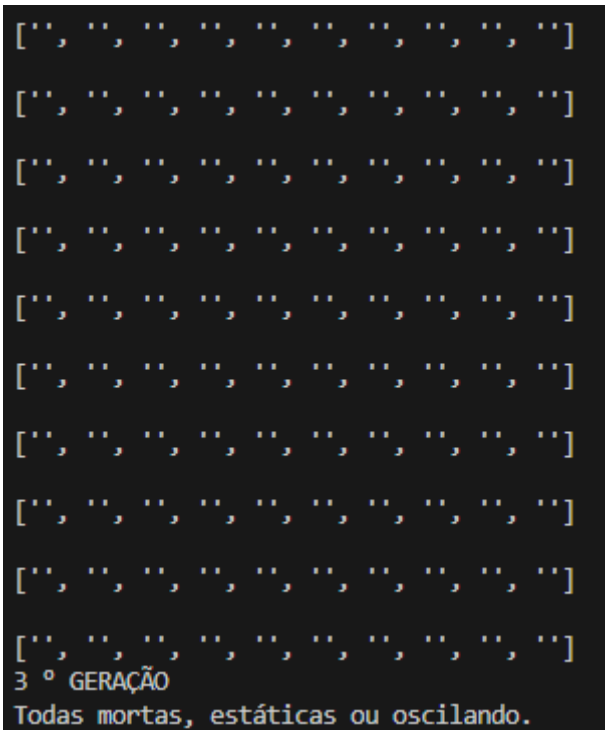

Figura 6. Tela de fim da simulação

3.2. Testes e Erros

Diversos testes foram realizados para validar o funcionamento do programa. O padrão "bloco" foi usado para verificar a detecção de estado estático e duas novas listas (ultima_geracao e penultima_geracao) para testar a detecção de oscilações. Em ambos os casos, o sistema finalizou corretamente a simulação após reconhecer os padrões. Padrões aleatórios também foram testados, revelando comportamentos como extinção rápida, estabilização e superpopulação. Durante os testes iniciais, foi identificado um erro na detecção de osciladores devido à comparação inadequada das gerações. Esse problema foi corrigido com a inclusão da comparação com a penúltima geração. Após os ajustes, o programa passou a se comportar de forma estável e precisa.

4. Conclusão

A simulação do Jogo da Vida foi implementada com sucesso, funcionando de forma interativa e autônoma. O sistema se mostrou capaz de lidar com diferentes configurações de entrada, além de identificar corretamente os padrões que levam ao fim da execução, como extinções, padrões estáticos e oscilações. Durante o processo, não apenas conseguimos cumprir os requisitos iniciais, como também adicionamos um recurso extra importante: a detecção de padrões oscilantes, que exigiu um controle cuidadoso das gerações anteriores para garantir que o jogo se encerrasse adequadamente nesses casos.

4.1. Melhorias

Seria interessante implementar recursos visuais mais elaborados, como uma interface no terminal mais detalhada, o jogo poderia rodar em looping, a cada fim de jogo, o usuário poderia decidir se ele gostaria de jogar novamente. Outra melhoria possível seria dar poder de opção para o usuário escolher a amplitude do tabuleiro, ou até mesmo, permitir a criação e salvamento de padrões personalizados.

5. Referências

1. WIKIPÉDIA. *John Conway*. Disponível em: https://pt.wikipedia.org/wiki/John_Conway. Acesso em: 28 abr. 2025.
2. PYTHON SOFTWARE FOUNDATION. *time — Manipulação de tempo. Documentação Python 3.13.3*. Disponível em: time — Time access and conversions — Python 3.13.3 documentation. Acesso em: 25 abr. 2025.
3. PYTHON SOFTWARE FOUNDATION. *Função all() – Documentação oficial*. Disponível em: https://docs.python.org/3/library/functions.html#all. Acesso em: 2 mai. 2025.
4. PYTHON SOFTWARE FOUNDATION. *os — Interfaces diversas do sistema operativo*. Disponível em: https://docs.python.org/3/library/os.html. Acesso em: 2 mai. 2025.
5. OLIVEIRA, M. de. *O Algoritmo do Jogo da Vida de Conway e sua Importância Histórica.* Disponível em: https://pt.linkedin.com/pulse/o-algoritmo-do-jogo-da-vida-de-conway-e-sua-hist%C3%B3rica-de-oliveira-mad7f. Acesso em: 2 mai. 2025.
6. PYTHON SOFTWARE FOUNDATION. *5.1.4. Compreensões de lista aninhadas* Disponível em: 5. Estruturas de dados — Documentação Python 3.13.3. Acesso em: 2 mai. 2025.
7. PYTHON SOFTWARE FOUNDATION. *copy — Operações de cópia profunda e cópia rasa.* Disponível em: copy — Operações de cópia profunda e cópia rasa — Documentação Python 3.13.3. Acesso em: 2 mai. 2025.