

# Jogo da Vida 2 - Implementação com Hash Duplo

João Vitor Coelho Oliveira  
Matrícula: 23.1.4133

13 de setembro de 2024

Universidade Federal de Ouro Preto  
Ciência da Computação  
Estrutura de Dados I

## Resumo

Este trabalho prático apresenta a implementação do Jogo da Vida de John Conway utilizando uma Tabela Hash de Endereçamento Linear com Hash Duplo. O relatório destaca as vantagens da nova abordagem em relação à versão anterior, que utilizava listas encadeadas para representar a matriz esparsa. As decisões de implementação, desafios e análise de desempenho são discutidos em detalhes.

## 1 Introdução

O Jogo da Vida é um autômato celular criado por John Horton Conway em 1970. Esta implementação simula a evolução de células vivas e mortas em um grid bidimensional ao longo de várias gerações, de acordo com regras simples baseadas no número de vizinhos vivos. Este trabalho explora uma abordagem com Tabelas Hash para otimizar a memória e o desempenho em grids grandes.

## 1.1 Especificações do Problema

As regras do Jogo da Vida são baseadas no estado dos vizinhos de cada célula:

1. Células vivas com menos de dois vizinhos vivos morrem por solidão.
2. Células vivas com mais de três vizinhos vivos morrem por superpopulação.
3. Células mortas com exatamente três vizinhos vivos tornam-se vivas.
4. Células vivas com dois ou três vizinhos vivos continuam vivas.

O objetivo é simular várias gerações do grid, imprimindo o estado final.

## 1.2 Especificações da Máquina

Processador: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz

Memória RAM: 16,0 GB

Sistema Operacional: Windows 10

## 2 Considerações Iniciais

- **Ambiente de desenvolvimento:** Visual Studio Code
- **Linguagem Utilizada:** C
- **Ferramentas:** GCC, Valgrind, WSL (Windows Subsystem for Linux)

## 3 Metodologia

Esta seção detalha as decisões de implementação, as vantagens da Tabela Hash com Hash Duplo em relação à versão anterior com listas encadeadas, e o processo de evolução do grid.

### 3.1 Implementação com Hash Duplo

Nesta implementação, utilizamos uma Tabela Hash de Endereçamento Linear com Hash Duplo para representar as células vivas no grid. A ideia é mapear as coordenadas das células vivas para uma tabela hash, permitindo um acesso rápido e eficiente ao estado de cada célula. O uso de hash duplo ajuda a reduzir colisões, garantindo que as células sejam distribuídas de maneira uniforme na tabela.

## 3.2 Vantagens do Hash Duplo

A utilização de uma Tabela Hash traz diversas vantagens em relação à abordagem anterior com listas encadeadas:

- **Acesso Rápido:** O uso de hash duplo permite acessar o estado de uma célula em tempo constante  $O(1)$ , comparado ao tempo linear  $O(n)$  em listas encadeadas.
- **Menor Uso de Memória:** Apenas as células vivas são armazenadas na tabela, evitando o uso excessivo de memória em grids grandes e esparsos.
- **Redução de Colisões:** O uso de duas funções hash (Hash Duplo) ajuda a minimizar colisões, garantindo uma distribuição eficiente das células vivas.

## 3.3 Dificuldades e Ajustes na Implementação

Durante o desenvolvimento, foi necessário lidar com alguns desafios:

- **\*\*Índices Negativos no Hash\*\*:** Inicialmente, as funções hash geravam índices negativos, causando erros de segmentação. Isso foi resolvido ajustando o cálculo do módulo para garantir que os valores fossem sempre positivos.
- **\*\*Manutenção do Grid\*\*:** Ao contrário da abordagem anterior, onde o grid era uma matriz de listas encadeadas, aqui precisamos gerenciar cuidadosamente a substituição da tabela hash a cada geração.
- **\*\*Gerenciamento de Memória\*\*:** O uso de Valgrind foi essencial para garantir que não houvesse vazamentos de memória ao desalocar a tabela antiga após cada geração.

## 3.4 Entrada e Saída

**Entrada:** A entrada consiste no tamanho do grid, o número de iterações e o estado inicial das células.

- A dimensão do grid.
- O número de iterações (gerações).
- O estado inicial do grid (1 para célula viva, 0 para célula morta).

```

#include <stdio.h>
#include <stdlib.h>
#include "automato.h"

int main() {
    int dimensao, geracoes;
    scanf("%d %d", &dimensao, &geracoes);

    AutomatoCelular *aut = alocarReticulado(dimensao * dimensao * 2);

    // leitura do estado inicial
    for (int i = 0; i < dimensao; i++) {
        for (int j = 0; j < dimensao; j++) {
            int estado;
            scanf("%d", &estado);
            if (estado == 1) {
                inserirCelula(aut, i, j);
            }
        }
    }
}

```

Figura 1: Leitura de dados

**Saída:** A saída é o estado final do grid após todas as iterações, impresso como uma matriz de '0's e '1's.

```

joaqu@DESKTOP-OIE98RL:~/EstruturadeDados1/JogoDaVidaTP3$ make
joaqu@DESKTOP-OIE98RL:~/EstruturadeDados1/JogoDaVidaTP3$ ./exe tests/1.in
3 1
0 0 0
0 1 1
0 1 0
0 0 0
0 1 1
0 1 1
joaqu@DESKTOP-OIE98RL:~/EstruturadeDados1/JogoDaVidaTP3$

```

Figura 2: Saída via terminal

## 4 Comparação com a Versão Anterior

A versão anterior utilizava uma matriz esparsa implementada com listas encadeadas para representar as células vivas. A nova versão, com Hash Duplo, apresenta diversas melhorias:

### 4.1 Memória

Enquanto a abordagem com listas encadeadas já otimizava o uso de memória em relação a uma matriz densa, o Hash Duplo vai além, oferecendo uma alocação ainda mais eficiente. Apenas as células vivas são armazenadas, e não há necessidade de manter múltiplas listas para linhas e colunas.

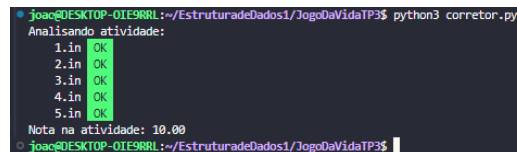
### 4.2 Desempenho

A implementação com Hash Duplo proporciona um desempenho significativamente melhor em grids grandes. O acesso constante  $O(1)$  às células vivas é uma vantagem crucial em relação ao tempo linear das listas encadeadas.

Isso foi especialmente notado em casos de teste com grids maiores, onde a versão com hash duplo executa mais rapidamente.

## 5 Resultados

Os resultados obtidos mostram que a implementação com hash duplo foi bem-sucedida. O uso de memória foi otimizado, e o desempenho foi aprimorado em relação à versão anterior. O Valgrind foi utilizado para garantir que não houvesse vazamentos de memória, e todos os casos de teste passaram com sucesso.

A terminal window with a dark background. The prompt is 'joao@DESKTOP-01E9RRL: ~/EstruturadeDados1/JogoDaVidaTP3\$'. The command 'python3 corretor.py' has been executed. The output shows 'Analisando atividade:' followed by a list of five items: '1.in OK', '2.in OK', '3.in OK', '4.in OK', and '5.in OK'. Below this, it says 'Nota na atividade: 10.00'. The prompt is now 'joao@DESKTOP-01E9RRL: ~/EstruturadeDados1/JogoDaVidaTP3\$' with a cursor.

```
joao@DESKTOP-01E9RRL: ~/EstruturadeDados1/JogoDaVidaTP3$ python3 corretor.py
Analisando atividade:
1.in OK
2.in OK
3.in OK
4.in OK
5.in OK
Nota na atividade: 10.00
joao@DESKTOP-01E9RRL: ~/EstruturadeDados1/JogoDaVidaTP3$
```

Figura 3: Teste com corretor.py

## 6 Lógica de Resolução

A lógica de resolução do problema foi baseada na representação do grid como uma Tabela Hash de Endereçamento Linear com Hash Duplo, onde apenas as células vivas são armazenadas. Essa abordagem otimiza o uso de memória, já que não é necessário armazenar o grid completo, e permite um processamento eficiente para grandes grids. A seguir, são detalhadas as principais funções implementadas para resolver o problema.

### 6.1 Uso de Tabelas Hash com Hash Duplo

Para evitar colisões e garantir uma distribuição eficiente das células vivas na tabela, utilizamos duas funções de hash. A primeira função (hash1) calcula a posição base de uma célula, enquanto a segunda função (hash2) é usada para tratar colisões, gerando um deslocamento alternativo. Esse esquema de endereçamento reduz significativamente o tempo de busca, inserção e remoção de células.

A principal vantagem dessa abordagem é o tempo constante  $O(1)$  nas operações de acesso, em contraste com o tempo linear  $O(n)$  das listas encadeadas. Isso é especialmente importante em grids grandes e esparsos, onde muitas células estão mortas e não precisam ser representadas.

```

#include "double_hash.h"

#define PESOS1 1, 2
#define PESOS2 3, 4

int hash1(int linha, int coluna, int tamanho) {
    int pesos[] = PESOS1;
    int hash = (pesos[0] * linha + pesos[1] * coluna) % tamanho;
    return (hash < 0) ? hash + tamanho : hash; // Garante que o resultado seja positivo
}

int hash2(int linha, int coluna, int tamanho) {
    int pesos[] = PESOS2;
    int hash = (pesos[0] * linha + pesos[1] * coluna) % (tamanho - 1);
    if (hash < 0) {
        hash += (tamanho - 1); // Garante que o resultado seja positivo
    }
    return (hash == 0) ? 1 : hash; // Garante que o hash2 nunca seja 0
}

```

Figura 4: Funções Hash1 e Hash2

## 6.2 Função inserirCelula

A função `inserirCelula` é responsável por inserir uma célula viva na Tabela Hash. Para isso, utilizamos as duas funções hash mencionadas. Caso ocorra uma colisão (isto é, a posição calculada pela primeira função já esteja ocupada), a segunda função hash gera um deslocamento, e a célula é inserida na nova posição disponível.

Esta abordagem evita colisões de maneira eficiente, e o uso de memória é otimizado, pois apenas as células vivas são armazenadas na tabela.

```

void inserirCelula(AutometroCelular *aut, int linha, int coluna) {
    int tamanho = aut->tamanho;
    int h1 = hash1(linha, coluna, tamanho);
    int h2 = hash2(linha, coluna, tamanho);

    for (int i = 0; i < tamanho; i++) {
        int pos = (h1 + i * h2) % tamanho;

        // Verificamos se a posição está dentro dos limites
        if (pos < 0 || pos >= tamanho) {
            printf("Erro: índice fora dos limites (%d).\n", pos);
            return;
        }

        if (aut->tabela[pos] == NULL) { // Se a posição está livre, insira a célula
            Célula *nova = (Célula *)malloc(sizeof(Célula));
            if (!nova) {
                printf("Erro: falha ao alocar célula.\n");
                return;
            }
            nova->linha = linha;
            nova->coluna = coluna;
            nova->estado = 1; // Viva
            aut->tabela[pos] = nova;
            aut->numElementos++;
            return;
        }
    }

    // Caso a tabela esteja cheia (não deveria acontecer, mas é uma verificação extra)
    printf("Erro: tabela hash cheia! Não foi possível inserir a célula (%d, %d).\n", linha, coluna);
}

```

Figura 5: Função inserirCelular

## 6.3 Função evoluirReticulado

A função `evoluirReticulado` implementa a lógica de evolução do Jogo da Vida, onde o grid é atualizado geração após geração com base nas regras do jogo. Para cada célula viva, a função conta seus vizinhos utilizando a Tabela Hash e aplica as regras para determinar se a célula continuará viva ou morrerá.

A função também verifica células mortas adjacentes para determinar se elas devem se tornar vivas na próxima geração. Ao final de cada iteração, a tabela hash antiga é substituída pela nova, que contém o estado atualizado do grid.

```
void evoluirReticulado(AutomatoCelular *aut, int dimensao) {
    AutomatoCelular *novaAut = alocarReticulado(aut->tamanho); // Tamanho é o mesmo do aut atual

    // Verificações de limites e inicialização
    if (!novaAut || !aut) {
        printf("Erro: falha ao alocar reticulado.\n");
        return;
    }

    for (int i = 0; i < dimensao; i++) {
        for (int j = 0; j < dimensao; j++) {
            int vizinhos = contarVizinhos(aut, i, j);
            int viva = buscarCelula(aut, i, j);

            // Aplicando as regras do jogo da vida
            if (vizinhos == 3 || (vizinhos == 2 && viva)) {
                inserirCelula(novaAut, i, j);
            }
        }
    }
}
```

Figura 6: Função evoluirReticulado

## 6.4 Função buscarCelula

A função `buscarCelula` realiza a busca do estado de uma célula específica no grid. Utilizando as funções de hash, ela verifica se a célula está presente na Tabela Hash (indicando que a célula está viva). Se a célula não estiver presente, isso significa que ela está morta.

Essa função é essencial para verificar o estado dos vizinhos de uma célula viva e determinar sua evolução de acordo com as regras do jogo.

```
int buscarCelula(AutomatoCelular *aut, int linha, int coluna) {
    int tamanho = aut->tamanho;
    int h1 = hash1(linha, coluna, tamanho);
    int h2 = hash2(linha, coluna, tamanho);

    for (int i = 0; i < tamanho; i++) {
        int pos = (h1 + i * h2) % tamanho;

        if (pos < 0 || pos >= tamanho) {
            printf("Erro: índice fora dos limites ao buscar célula (%d).\n", pos);
            return 0; // Retornamos 0 (morta) se o índice estiver fora dos limites
        }

        if (aut->tabela[pos] != NULL && aut->tabela[pos]->linha == linha && aut->tabela[pos]->coluna == coluna) {
            return 1; // Célula viva
        }
    }
    return 0; // Célula morta
}
```

Figura 7: Função buscarCelula

## 6.5 Função contarVizinhos

A função `contarVizinhos` verifica as oito posições ao redor de uma célula para contar quantos de seus vizinhos estão vivos. Ela utiliza a função `buscarCelula` para acessar cada uma dessas posições na Tabela Hash.

Essa contagem é crucial para decidir o futuro de cada célula: se ela continuará viva, morrerá ou se tornará viva, dependendo do número de vizinhos.

```

int contarVizinhos(AutomatoCelular *aut, int linha, int coluna) {
    int vizinhos = 0;
    for (int i = -1; i <= 1; i++) {
        for (int j = -1; j <= 1; j++) {
            if (i == 0 && j == 0) continue;
            vizinhos += buscarCelula(aux, linha + i, coluna + j);
        }
    }
    return vizinhos;
}

```

Figura 8: Função contarVizinhos

## 6.6 Vantagens da Lógica com Hash Duplo

A utilização de Tabelas Hash com Hash Duplo traz diversas vantagens, incluindo:

- **Otimização de Memória:** Apenas as células vivas são armazenadas, o que reduz significativamente o uso de memória em grids esparsos.
- **Desempenho Melhorado:** O tempo de acesso constante  $O(1)$  nas operações de busca e inserção melhora o desempenho em comparação com listas encadeadas.
- **Escalabilidade:** A abordagem com hash duplo é escalável para grids grandes, permitindo a representação de configurações complexas sem consumir grandes quantidades de memória.

## 7 Conclusão

A implementação do Jogo da Vida com Tabela Hash de Endereçamento Linear e Hash Duplo trouxe melhorias significativas em termos de desempenho e uso de memória. O projeto permitiu o aprofundamento em técnicas de otimização e estrutura de dados, além de demonstrar a importância de uma boa gestão de memória em programas complexos. A abordagem com hash duplo se mostrou eficiente e escalável, sendo uma solução robusta para o problema.

## 8 Referências

- Gardner, M. (1970). Mathematical games: The fantastic combinations of John Conway's new solitaire game "life". Scientific American, 223(4):120-123.
- Valgrind. Disponível em: <http://valgrind.org/>



- Visual Studio Code. Disponível em: <https://code.visualstudio.com/>
- Overleaf. Disponível em: <https://www.overleaf.com/>