Relatório sobre a Implementação de um Autômato Celular

Introdução

Os autômatos celulares, introduzidos por John Von Neumann na década de 1950, são modelos matemáticos que simulam comportamentos biológicos e físicos por meio de células que interagem localmente. Um exemplo famoso é o "Jogo da Vida", criado por John Horton Conway em 1970. O jogo consiste em um reticulado bidimensional onde cada célula pode estar viva (1) ou morta (0). A evolução do reticulado é determinada por regras simples que consideram o estado das células vizinhas.

Neste trabalho, propomos a implementação de um programa em C que simula o Jogo da Vida, **utilizando Tabelas Hash com endereçamento aberto e a técnica de hashing duplo para armazenar o estado das células.**

Objetivo

O objetivo principal é desenvolver um Tipo Abstrato de Dados (TAD) que representa o autômato celular, capaz de ler reticulados, evoluí-los conforme as regras do jogo e imprimir o resultado em uma nova geração. As operações que devem ser implementadas no TAD incluem:

1. **alocarReticulado**: Aloca um TAD AutomatoCelular.
2. **desalocarReticulado**: Libera a memória ocupada por um TAD AutomatoCelular.
3. **leituraReticulado**: Inicializa o TAD com dados do terminal.
4. **evoluirReticulado**: Retorna o estado do reticulado após um número específico de gerações.
5. **imprimeReticulado**: Exibe o estado atual do TAD AutomatoCelular.

Detalhes da Implementação

O autômato celular é representado como uma Tabela Hash com endereçamento linear e hashing duplo. Utilizamos dois vetores de pesos para as funções hash: [1, 2] para a função primária e [3, 4] para a secundária. Isso garante uma distribuição eficiente das chaves, reduzindo colisões e melhorando o desempenho.

Estrutura da Tabela Hash:

- **Chaves**: Representam as coordenadas das células (linha, coluna).
- **Valores**: Armazenam o estado das células (0 ou 1).

Funções Implementadas e Análise de Complexidade

- **criarHashTable**: Aloca memória para a tabela hash e inicializa suas chaves como VAZIO (-1) e valores como 0.
    - **Complexidade**: O(n), onde n é o tamanho da tabela.
- **destruirHashTable**: Libera a memória alocada para a tabela hash.
    - **Complexidade**: O(n), onde n é o tamanho da tabela.
- **hash1 e hash2**: Funções de hash que calculam a posição inicial e o salto, respectivamente. Ambas têm complexidade O(1), pois realizam operações aritméticas simples.
- **encontrarPosicao**: Localiza a posição correta para uma nova chave, lidando com colisões através de hashing duplo.
    - **Complexidade**: O(n) no pior caso, onde n é o número de tentativas necessárias até encontrar uma posição vazia.
- **inserirHashTable**: Insere uma nova chave e seu valor correspondente na tabela.
    - **Complexidade**: O(n) no pior caso devido a possíveis colisões, mas O(1) em média.
- **buscarHashTable**: Retorna o valor associado a uma chave, ou indica se a célula está vazia.
    - **Complexidade**: O(n) no pior caso devido a colisões, mas O(1) em média.
- **alocarReticulado**: Aloca um TAD AutomatoCelular e inicializa suas propriedades.
    - **Complexidade**: O(1), pois apenas aloca espaço para as variáveis e inicializa.
- **desalocarReticulado**: Libera a memória ocupada por um TAD AutomatoCelular.
    - **Complexidade**: O(1), pois apenas libera a memória ocupada pelo reticulado.
- **leituraReticulado**: Inicializa o TAD com dados do terminal, preenchendo a tabela hash com as células vivas.
    - **Complexidade**: O(r), onde r é o número de células vivas a serem lidas e inseridas na tabela.
- **evoluirReticulado**: Função que atualiza o estado do reticulado com base nas regras do Jogo da Vida.
    - **Complexidade**: O(m), onde m é a dimensão do reticulado, pois cada célula é atualizada com base em suas vizinhas.
- **imprimeReticulado**: Exibe o estado atual do TAD AutomatoCelular no terminal.
    - **Complexidade**: O(m), onde m é a dimensão do reticulado, pois percorre todas as células para impressão.

Comparação com Implementações Anteriores

- **TP1**: Utilizou uma matriz normal para armazenar células vivas e mortas, ocupando um espaço total de O(m × m) para uma matriz de dimensão m. Esta abordagem se tornava ineficiente em situações onde o número de células vivas (r) era pequeno, resultando em um uso excessivo de memória. Por exemplo, uma matriz 100x100 ocupava 10.000 posições de memória, mesmo que apenas 10% das células estivessem vivas.

- **TP2**: A matriz esparsa em formato de listas encadeadas permitiu uma representação mais econômica. O espaço utilizado passou a ser O(m + m + r), onde o primeiro m corresponde às listas de células, o segundo m refere-se aos endereços e r às células vivas. Essa abordagem se mostrou mais eficiente em termos de memória, pois permitiu armazenar apenas as células vivas, resultando em economia significativa quando r era muito menor que m². Por exemplo, para uma matriz 100x100 com apenas 10 células vivas, o uso de memória seria reduzido para aproximadamente 120 posições, em vez das 10.000 necessárias na matriz tradicional.

- **TP3**: Neste trabalho, adotamos uma Tabela Hash para armazenar as células vivas, proporcionando um uso ainda mais econômico da memória. A estrutura de tabela hash ocupa O(n), onde n é o número de células vivas armazenadas. Isso se traduz em uma alocação de memória que é linear em relação ao número de células vivas, resultando em um uso eficiente de recursos. Para uma situação semelhante à do TP2, onde apenas 10 células vivas são armazenadas, a tabela hash ocuparia em média apenas 20 a 30 posições, dependendo da taxa de colisão e do fator de carga da tabela. Isso torna o TP3 a implementação mais eficiente em termos de uso de memória, especialmente quando r é pequeno em comparação a m².

Conclusão

A implementação do autômato celular com Tabelas Hash e hashing duplo permite uma simulação eficiente do Jogo da Vida. O uso de um TAD modularizado facilita a manutenção e a expansão do código, enquanto a análise de complexidade assegura que o algoritmo opere dentro dos limites exigidos. Com isso, conseguimos criar um modelo que não só respeita as regras do jogo, mas também otimiza a gestão da memória e o tempo de execução, conforme as diretrizes propostas no enunciado do trabalho.

TP.C

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include "automato.h"

int main(int argc, char* argv[]) {
   int linhas, geracoes;

   // Le a primeira linha do arquivo que contem a dimensao da matriz e
o numero de geracoes
   fscanf(stdin, "%d %d", &linhas, &geracoes);

   AutomatoCelular* automato = alocarReticulado(linhas, linhas);
   leituraReticulado(automato, stdin); // Le o reticulado a partir do
stdin

   evoluirReticulado(automato, geracoes);

   imprimeReticulado(automato);

   desalocarReticulado(&automato);

   return 0;
}
```

AUTOMATO.H

```c
#ifndef AUTOMATO_H
#define AUTOMATO_H

#include "double_hash.h"

typedef struct AutomatoCelular {
   HashTable* tabela; // Usa a tabela hash para armazenar apenas
células vivas
   int linhas, colunas;
} AutomatoCelular;

// Funções principais
AutomatoCelular* alocarReticulado(int linhas, int colunas);
int leituraReticulado(AutomatoCelular* automato, FILE* arquivo); //
Alterado para retornar int
void desalocarReticulado(AutomatoCelular** automato);
void evoluirReticulado(AutomatoCelular* automato, int geracoes);
```

```
void imprimeReticulado(const AutomatoCelular* automato);

#endif // AUTOMATO_H
```

AUTOMATO.C

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

// Aloca e inicializa um reticulado
AutomatoCelular* alocarReticulado(int linhas, int colunas) {
   AutomatoCelular* automato =
(AutomatoCelular*)malloc(sizeof(AutomatoCelular));
   if (automato == NULL) {
       return NULL;
   }

   automato->linhas = linhas;
   automato->colunas = colunas;
   automato->tabela = criarHashTable(linhas * colunas); // Tabela hash
do tamanho da area do reticulado

   if (automato->tabela == NULL) {
       free(automato);
       return NULL;
   }

   return automato; // Retorna o ponteiro para o automato alocado
}

// Libera a memoria associada ao reticulado
void desalocarReticulado(AutomatoCelular** automato) {
   if (*automato != NULL) {
       destruirHashTable((*automato)->tabela); // Destroi a tabela hash
       free(*automato); // Libera a memoria do automato
       *automato = NULL; // Zera o ponteiro para evitar dangling
pointers
   }
}
```

```c
// Le os valores iniciais do reticulado a partir de um arquivo
int leituraReticulado(AutomatoCelular* automato, FILE* arquivo) {
    int valor;
    for (int i = 0; i < automato->linhas; i++) {
        for (int j = 0; j < automato->colunas; j++) {
            if (fscanf(arquivo, "%d", &valor) != 1) {
                return 0; // Falha na leitura
            }
            if (valor == 1) { // Inserir apenas celulas vivas
                if (!inserirHashTable(automato->tabela, i *
automato->colunas + j, 1)) {
                    return 0; // Falha na insercao
                }
            }
            // Celulas com valor 0 (mortas) nao sao inseridas na tabela
hash
        }
    }
    return 1; // Sucesso
}

// Conta o numero de vizinhas vivas de uma celula
int contarVizinhasVivas(AutomatoCelular* automato, int linha, int
coluna) {
    int vizinhasVivas = 0; // Contador de celulas vivas vizinhas
    int dx[] = {-1, -1, -1, 0, 0, 1, 1, 1}; // Deslocamentos em linha
    int dy[] = {-1, 0, 1, -1, 1, -1, 0, 1}; // Deslocamentos em coluna

    // Verifica todas as 8 vizinhas
    for (int i = 0; i < 8; i++) {
        int novaLinha = linha + dx[i];
        int novaColuna = coluna + dy[i];

        // Verifica se a nova posicao esta dentro dos limites do
reticulado
        if (novaLinha >= 0 && novaLinha < automato->linhas && novaColuna
>= 0 && novaColuna < automato->colunas) {
            int valor = buscarHashTable(automato->tabela, novaLinha *
automato->colunas + novaColuna);
            vizinhasVivas += valor; // Adiciona a contagem se a vizinha
esta viva (1) ou morta (0)
        }
    }
```

```
    return vizinhasVivas; // Retorna o numero de vizinhas vivas
}

// Evolui o reticulado por um numero especificado de geracoes
void evoluirReticulado(AutomatoCelular* automato, int geracoes) {
    for (int g = 0; g < geracoes; g++) {
        HashTable* novaGeracao = criarHashTable(automato->linhas *
automato->colunas); // Cria uma nova tabela hash para a proxima geracao

        if (novaGeracao == NULL) {
            fprintf(stderr, "Erro ao criar nova geração.\n");
            return;
        }

        // Itera sobre todas as celulas do reticulado
        for (int i = 0; i < automato->linhas; i++) {
            for (int j = 0; j < automato->colunas; j++) {
                int vivas = contarVizinhasVivas(automato, i, j);
                int estadoAtual = buscarHashTable(automato->tabela, i *
automato->colunas + j); // Obtem o estado atual da celula

                // Aplica as regras do Jogo da Vida
                int novoEstado = 0;
                if (estadoAtual == 1 && (vivas == 2 || vivas == 3)) {
                    novoEstado = 1; // A celula continua viva
                } else if (estadoAtual == 0 && vivas == 3) {
                    novoEstado = 1; // A celula se torna viva
                }
                // Caso contrario, a celula permanece morta (nao e
inserida na tabela)

                if (novoEstado == 1) { // Insere apenas celulas vivas
                    if (!inserirHashTable(novaGeracao, i *
automato->colunas + j, 1)) {
                        fprintf(stderr, "Erro ao inserir nova
célula.\n");
                        destruirHashTable(novaGeracao);
                        return;
                    }
                }
                // Celulas mortas nao sao inseridas na tabela hash
            }
```

```
        }

        destruirHashTable(automato->tabela); // Destroi a tabela
anterior
        automato->tabela = novaGeracao;      // Atualiza para a nova
geracao
    }
}

// Imprime o reticulado no estado atual
void imprimeReticulado(const AutomatoCelular* automato) {
    for (int i = 0; i < automato->linhas; i++) {
        for (int j = 0; j < automato->colunas; j++) {
            int valor = buscarHashTable(automato->tabela, i *
automato->colunas + j);
            printf("%d ", valor);
        }
        printf("\n");
    }
}
```

DOUBLE_HASH.H

```
#ifndef DOUBLE_HASH_H
#define DOUBLE_HASH_H

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

// Estrutura da Tabela Hash
typedef struct HashTable {
    int* chaves;       // Vetor de chaves
    int* valores;      // Vetor de valores
    int tamanho;       // Tamanho total da tabela
    int ocupados;      // Numero de posicoes ocupadas
} HashTable;

// Funces principais
HashTable* criarHashTable(int tamanho);
void destruirHashTable(HashTable* tabela);
```

```
int inserirHashTable(HashTable* tabela, int chave, int valor);
int buscarHashTable(HashTable* tabela, int chave);

// Funcoes auxiliares para hashing duplo
int hash1(int chave, int tamanho);
int hash2(int chave, int tamanho);
int encontrarPosicao(HashTable* tabela, int chave);

#endif // DOUBLE_HASH_H
```

DOUBLE_HASH.C

```
#include "double_hash.h"
#include <stdlib.h>
#include <stdio.h>

#define VAZIO -1

// Cria uma tabela hash
HashTable* criarHashTable(int tamanho) {
    HashTable* tabela = (HashTable*)malloc(sizeof(HashTable));
    if (tabela == NULL) return NULL;

    tabela->chaves = (int*)malloc(tamanho * sizeof(int));
    tabela->valores = (int*)malloc(tamanho * sizeof(int));

    if (tabela->chaves == NULL || tabela->valores == NULL) {
        free(tabela);
        return NULL;
    }

    for (int i = 0; i < tamanho; i++) {
        tabela->chaves[i] = VAZIO; // Inicializa todas as chaves como
vazias
        tabela->valores[i] = 0;    // Inicializa os valores como 0
    }

    tabela->tamanho = tamanho;
    tabela->ocupados = 0;

    return tabela;
}
```

```c
// Destroi a tabela hash
void destruirHashTable(HashTable* tabela) {
    if (tabela) {
        free(tabela->chaves);
        free(tabela->valores);
        free(tabela);
    }
}

// Funcao de hash primaria
int hash1(int chave, int tamanho) {
    return ((chave * 1 + 2) % tamanho);
}

// Funcao de hash secundaria
int hash2(int chave, int tamanho) {
    return ((chave * 3 + 4) % tamanho);
}

// Funcao para encontrar a posicao correta de uma chave
int encontrarPosicao(HashTable* tabela, int chave) {
    int posicao = hash1(chave, tabela->tamanho);
    int salto = hash2(chave, tabela->tamanho);
    int i = 0;

    while (tabela->chaves[posicao] != VAZIO) {
        if (tabela->chaves[posicao] == chave) {
            return posicao; // Retorna a posicao se encontrar a chave
        }
        posicao = (posicao + salto) % tabela->tamanho;
        i++;

        if (i >= tabela->tamanho) return -1; // Se percorreu toda a
tabela e nao encontrou
    }

    return posicao; // Retorna a posicao onde pode inserir
}

// Funcao para inserir na tabela hash
int inserirHashTable(HashTable* tabela, int chave, int valor) {
    if (tabela->ocupados >= tabela->tamanho) return 0; // Tabela cheia
```

```
    int posicao = encontrarPosicao(tabela, chave);
    if (posicao == -1) return 0; // Nao encontrou espaco

    if (tabela->chaves[posicao] == VAZIO) {
        tabela->ocupados++;
    }

    tabela->chaves[posicao] = chave;
    tabela->valores[posicao] = valor;
    return 1;
}

// Funcao para buscar na tabela hash
int buscarHashTable(HashTable* tabela, int chave) {
    int posicao = encontrarPosicao(tabela, chave);
    if (posicao == -1 || tabela->chaves[posicao] == VAZIO) {
        return 0;
    }

    return tabela->valores[posicao]; // Retorna 1 se a celula esta viva
}
```