Universidade Federal de Ouro Preto - UFOP
Instituto de Ciências Exatas e Biológicas - ICEB
Departamento de Computação - DECOM
Ciência da Computação

Trabalho Prático I

BCC202 - Estruturas de Dados I

Vitor Oliveira Diniz
Maria Luiza Aragão
Jessica Machado
Professor: Pedro Silva

Ouro Preto
10 de janeiro de 2023

Sumário

Lista de Códigos Fonte

1 Introdução

Neste trabalho foi necessário entregar o código em C e um relatório referente ao que foi desenvolvido. O algoritmo a ser desenvolvido é um modelo de autômato celular, mais especificamente o jogo da vida (the game of life).

A codificação foi feita em C, usando somente a biblioteca padrão da GNU, sem o uso de bibliotecas adicionais.

1.1 Especificações do problema

Autômatos celulares são simples problemas de auto reprodução celular em que o sistema baseia-se em estados anteriores. Um autômato é definido por seu espaço celular e sua regra de transição. Nosso espaço celular é composto por um retículo de N células idênticas, que se encontram mortas ou vivas em um arranjo bi-dimensional. Assim, devemos implementar um programa em linguagem C que nos permita ler um reticulado do jogo da vida e retornar a malha da próxima geração, assim seguindo as regras definidas pelo jogo. Essas que consistem em:

Assim, devemos implementar um programa em linguagem C que nos permita ler um reticulado do jogo da vida e retornar a malha da próxima geração, assim seguindo as regras definidas pelo jogo. Essas que consistem em:

- Manter uma célula viva caso duas ou três de suas células vizinhas também estiverem vivas;
- Células vivas morrem caso haja superpopulação de três células vizinhas também vivas;
- Células vivas morrem caso haja menos de duas células vivas vizinhas;
- Células mortas tornam-se vivas caso haja três células vizinhas vivas.

1.2 Considerações Iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: Visual Studio Code.
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Visual Studio Code LaTeXWorkshop.

1.3 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- *CLANG*: ferramentas de análise estática do código.
- *Valgrind*: ferramentas de análise dinâmica do código.

1.4 Especificações da máquina

A máquina onde o desenvolvimento e os testes foram realizados possui a seguinte configuração:

- Processador: Ryzen 7-5800H.
- Memória RAM: 16 Gb.
- Sistema Operacional: Arch Linux x86_64.

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

```
gcc -c automato.c -Wall
gcc -c tp.c -Wall
gcc -g automato.o tp.o -o exe
```

Usou-se para a compilação as seguintes opções:

- *-g*: para compilar com informação de depuração e ser usado pelo Valgrind.
- *-Wall*: para mostrar todos os possível *warnings* do código.
- *-c*: Para compilar o arquivo sem linkar os arquivos para obtermos um arquivo do tipo objeto.

Para a execução do programa basta digitar:

```
./exe < caminho_até_o_arquivo_de_entrada
```

Onde "caminho-até-o-arquivo-de-entrada" pode ser: "1.in" para realizar o primeiro caso de teste e "2.in" para realizar o segundo.

2 Desenvolvimento

Seguindo as boas práticas de programação, implementamos um tipo abstrato de dados (TAD) para a representação do nosso problema. De acordo com o pedido, e para uma melhor organização, o nosso código foi modularizado em três arquivos, tp.c automato.h e automato.c em que o arquivo tp.c deve apenas invocar e tratar as respostas das funções e procedimentos definidos no arquivo automato.h.

2.1 TAD

Para começar a resolução do problema proposto, decidimos criar uma struct, que representaria o nosso autômato celular e seria nosso TAD, que no caso possui dois estados, vivo ou morto. Esses dois estados foram representados por um enum, ou enumeração, que é um tipo de dado definido pelo programador que irá assumir apenas um dos valores definidos.Seguindo essa lógica, determinamos o valor 0 para MORTO e 1 para VIVO.

```
1
2       typedef enum
3   {
4       MORTO = 0,
5       VIVO = 1
6   } VIDA;
7
8   struct automatoCelular
9   {
10      VIDA vida;
11  };
```

Código 1: TAD representando um automato celular

2.2 Funções

A seguir entraremos em detalhe sobre as principais funções utilizadas no programa.

2.2.1 lerDimensao

Criamos essa função para armazenar a dimensão da tabela em um ponteiro, passado através de um parâmetro.

```
1
2   void lerDimensao(int *dimensao)
3   {
4       scanf("%d", dimensao);
5   }
```

Código 2: Função lerDimensao

2.2.2 alocarReticulado

Posteriormente criamos esta função para alocar dinamicamente a tabela com as dimensões inseridas, primeiramente alocamos um ponteiro de tamanho n do nosso autômato, que será correspondente a cada linha. E usamos uma repetição para alocar n elementos em cada linha. Foi utilizada a função malloc da biblioteca padrão que realiza essa alocação dinamicamente.

```
1
2   AutomatoCelular **alocarReticulado(int dimensao)
3   {
4       AutomatoCelular **automatoCelular = malloc(dimensao * sizeof(
            AutomatoCelular *));
5
6       for (int i = 0; i < dimensao; i++)
7       {
```

```
        automatoCelular[i] = malloc(dimensao * sizeof(AutomatoCelular));
    }

    return automatoCelular;

    }
```

Código 3: Função alocarReticulado

2.2.3 desalocarReticulado

Esta função foi implementada com o objetivo de liberar o espaço de memória que alocamos para a tabela. Como alocamos uma matriz, foi necessário o uso de uma estrutura de repetição.

```
void desalocarReticulado(AutomatoCelular ***automatoCelular, int dimensao)
{
    for (int i = 0; i < dimensao; i++)
    {
        free((*automatoCelular)[i]);
    }
    free(*automatoCelular);
}
```

Código 4: Função desalocarReticulado

2.2.4 leituraReticulado

Aqui usamos como parâmetro a nossa matriz e sua dimensão, para assim poder utilizá-las nas repetições que possibilitariam o preenchimento do usuário em relação aos valores da tabela. Criamos uma variável do tipo inteiro para receber esses valores e na repetição atribuímos ele a matriz na posição atual, avançando para a próxima posição a cada vez que o "for" se repetir.

```
void leituraReticulado(AutomatoCelular **automatoCelular, int dimensao)
{
    int vidaReticulado;

    for (int i = 0; i < dimensao; i++)
    {
        for (int j = 0; j < dimensao; j++)
        {
            scanf("%d", &vidaReticulado);
            automatoCelular[i][j].vida = vidaReticulado;
        }
    }
}
```

Código 5: Função leituraReticulado

2.2.5 copiaAutomato

Esta função de cópia foi implementada com o intuito de ser uma cópia da matriz original já que, no nosso código, contamos as células ao redor e atualizamos sua situação (viva ou morta) imediatamente, o que poderia afetar no resultado final.

```

AutomatoCelular **copiaAutomato(AutomatoCelular **automatoCelular, int
    dimensao){


    AutomatoCelular **copia;
    copia = alocarReticulado(dimensao);

```

```
    for(int i = 0; i < dimensao; i++){
        for(int j = 0; j < dimensao; j++){
            copia[i][j].vida = automatoCelular[i][j].vida;
        }
    }

    return copia;

}
```

Código 6: Função copiaAutomato

2.2.6 evoluirReticulado

Para a função de evoluir o reticulado, iremos dividir a abordagem em 3 partes.

Primeiramente, devemos percorrer cada célula do reticulado para assim aplicarmos as regras definidas pelo jogo da vida. Atenção para a cópia da nossa matriz de autómatos que foi feita.

```
AutomatoCelular **evoluirReticulado(AutomatoCelular **automatoCelular, int
    dimensao)
{

    AutomatoCelular **CopiaAutomatoCelular;
    CopiaAutomatoCelular = copiaAutomato(automatoCelular, dimensao);

    for (int i = 0; i < dimensao; i++)
    {
        for (int j = 0; j < dimensao; j++)
        {
```

Código 7: Função evoluirReticulado parte 1

Em todas as regras devemos percorrer as células adjacentes a que está sendo analisada, assim devemos definir o valor inicial e final do for, caso algum elemento esteja na borda, não queremos acessar uma posição não alocada da matriz, então faremos um simples tratamento de exceção em que a posição inicial e final sempre estarão alocadas.

```

    int contadorCelular = 0;
    int x0 = i - 1;
    int y0 = j - 1;
    int xf = i + 1;
    int yf = j + 1;


    if (x0 < 0)
        x0 = 0;

    if (y0 < 0)
        y0 = 0;

    if(xf >= dimensao)
        xf = dimensao - 1;

    if(yf >= dimensao)
        yf = dimensao - 1;
```

Código 8: Função evoluirReticulado parte 2

Agora contaremos todas as células vivas adjacentes à célula analisada para enfim decidirmos qual regra será aplicada, já que todas as regras se baseiam na quantidade de células vivas adjacentes.

```

    for (int k = x0; k <= xf; k++)
```

```
    {
        for (int l = y0; l <= yf; l++)
        {
            if (!(k == i && l == j))
            {
                // printf("checando a celula[%d][%d] = %d\n",k, j,
                    automatoCelular[k][l].vida);
                if (CopiaAutomatoCelular[k][l].vida == VIVO)
                {
                    contadorCelular++;
                }
            }
        }
    }
```

Código 9: Função evoluirReticulado parte 3

em seguida iremos aplicar as regras do jogo da vida através de simples condições que dependem do estado da célula e do numero de células vivas adjacentes

```

    //celula renasce
    if (automatoCelular[i][j].vida == MORTO && contadorCelular == 3)
    {
        automatoCelular[i][j].vida = VIVO;
    } // solidao
    else if (automatoCelular[i][j].vida == VIVO && contadorCelular < 2)
    {
        automatoCelular[i][j].vida = MORTO;
    } // continua viva
    else if (automatoCelular[i][j].vida == VIVO && (contadorCelular == 2  ||
        contadorCelular == 3))
    {
        automatoCelular[i][j].vida = VIVO;
    } //sufocamento
    else if (automatoCelular[i][j].vida == VIVO && contadorCelular > 3)
    {
        automatoCelular[i][j].vida = MORTO;
    }
```

Código 10: Função evoluirReticulado parte 4

Ao final da função, iremos desalocar nossa matriz de cópia e retornar a nova geração do automato.

```

    desalocarReticulado(&CopiaAutomatoCelular, dimensao);
    return automatoCelular;
}
```

Código 11: Função evoluirReticulado parte 5

2.3 main

Na função main invocamos as funções necessárias para a realização dos procedimentos, sendo eles: a leitura dos dados da matriz, a sua alocação, seu tratamento, sua impressão e por último, desalocação.

```
1
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "automato.h"
5
6  int main(){
7      int dimensao;
8      AutomatoCelular **automatoCelular;
9
10     lerDimensao(&dimensao);
11     automatoCelular = alocarReticulado(dimensao);
12     leituraReticulado(automatoCelular, dimensao);
13     automatoCelular = evoluirReticulado(automatoCelular, dimensao);
14     imprimeReticulado(automatoCelular, dimensao);
15     desalocarReticulado(&automatoCelular, dimensao);
16
17     return 0;
18 }
```

Código 12: função main

3 Impressões Gerais

Com as funções já pré definidas foi muito mais fácil construir a lógica para o desenvolvimento modular do código. O nosso grupo então se reuniu e pensou coletivamente sobre a ordem de execução das funções e suas utilidades. O uso do Latex para a documentação foi um ponto positivo para o trabalho, com um membro do grupo já tendo certos conhecimentos e o restante aprendendo com o decorrer do uso. Outro conhecimento adquirido e posto em prática foi o uso do TAD, tipo abstrato de dados, para a solução do nosso problema. Houve também o desenvolvimento de um código bem modularizado, com uma excelente ajuda das instruções contidas no documento que nos foi disponibilizado como exemplo.

4 Análise

Após o desenvolvimento do programa, a primeira análise feita foi atráves dos casos de teste disponibilizados na página do trabalho prático do run.codes, com simples exemplos de entrada e saída, executamos o programa com um dos exemplos de entrada e assim, foi possível fazer uma simples análise se o programa se comportava corretamente. Primeiramente obtivemos algumas dificuldades que serão abordadas na conclusão. Após a correção do erro encontrado, as próximas realizações de testes apresentaram resultados iguais ao exemplo de saída disponibilizado. Depois dos testes iniciais para verificar um funcionamento inicial do programa, utilizamos o valgrind, uma ferramenta de análise dinâmica de código para conferir se há algum memory leak ou warning referente a manipulação de memória.

Após esses dois testes, fizemos o envio para o run codes que apresenta uma bateria de teste ainda mais extensa e obtivemos o resultado esperado.

5 conclusão

Com este trabalho ampliamos os nossos conhecimentos referente a criação de um TAD (tipo abstrato de dados) e como aplicá-lo para encontrar a solução de um problema. Aprendemos como utilizar e fazer um documento em LaTeX, que será extremamente útil e essencial ao longo do curso, tanto para apresentação de trabalhos acadêmicos e para o desenvolvimento de artigos científicos. Como uma dificuldade inicial, tivemos o erro de interpretação em relação aos dados da matriz e suas modificações.

Estávamos realizando a leitura das células adjacentes e aplicando as regras de transição imediatamente, considerando os novos dados. Após um tempo pensando sobre o que poderia estar afetando os nossos resultados, chegamos à conclusão de que os dados que deveriam ser levados em consideração seriam os da matriz original e não da matriz modificada gradativamente ao decorrer do programa.