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

Trabalho Prático I

BCC202 - Estrutura de Dados I

Caio Lucas Pereira da Silva, Vinicius Nunes dos Anjos, Lucca Sales de Souza Teodoro

Professor: Pedro Henrique Lopes Silva

Ouro Preto
11 de janeiro de 2023

Sumário

Lista de Tabelas

Lista de Códigos Fonte

1 Introdução

Para este trabalho é necessário entregar o código em C e um relatório referente ao que foi desenvolvido. O algoritmo a ser desenvolvido é o autômato celular Jogo da Vida (Conway's Game of Life), criado pelo matemático britânico John Horton Conway em 1970. Ele é baseado em regras simples que determinam como células vivas ou mortas interagem em um grid quadrado. Embora o jogo seja simples, as configurações resultantes podem ser surpreendentemente complexas e imprevisíveis.

O jogo da vida é muitas vezes usado como um exemplo de autômato celular, um sistema computacional que consiste em uma grade de células que podem estar em diferentes estados. Ele também é estudado como uma forma de simular sistemas biológicos e sociais, e foi usado como uma ferramenta para a ciência da computação e inteligência artificial.

A codificação deve ser feita em C, usando somente a biblioteca padrão da GNU, sem o uso de bibliotecas adicionais. Além disso, deve-se usar um dos padrões: ANSI C 89 ou ANSI C 99.

1.1 Especificações do problema

É necessário processar adequadamente um arquivo de entrada que contém a dimensão de uma matriz na sua primeira linha e os valores correspondentes nas linhas seguintes. Esses valores representam o estado inicial do jogo da vida. Com base nesses valores de entrada, é necessário gerar uma saída que apresente o próximo estado do jogo, levando em consideração as regras estabelecidas para a simulação.

1.2 Ferramentas utilizadas

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: Visual Studio Code. [1]
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Overleaf LaTeX. [2]

1.3 Ferramentas adicionais

Algumas ferramentas foram utilizadas para auxiliar no desenvolvimento, como:

- *Live Share*: ferramenta usada para *pair programming* à distância.
- *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: AMD Ryzen 5-5500U.
- Memória RAM: 8GB.
- Sistema Operacional: Linux Pop_OS.

[1] VScode está disponível em https://code.visualstudio.com/
[2] Disponível em https://www.overleaf.com/

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar o seguinte comando para rodar o arquivo *Makefile* disponível:

Compilando o projeto

```
make
```

Usou-se para a compilação as seguintes opções:

- *-o*: para definir o arquivo de saída.

- *-g*: para compilar com informação de depuração e ser usado pelo Valgrind.

- *-Wall*: para mostrar todos os possível *warnings* do código.

- *-c*: para compilação do código e geração dos arquivos objetos.

Para a execução do programa basta digitar :

```
./exe < arquivo.in
```

Onde o arquivo de entrada contém o tamanho da matriz e seus valores.

2 Desenvolvimento

O desenvolvimento foi realizado utilizando da técnica de pair programming, onde todos os integrantes do grupo programaram e participaram ativamente do código ao mesmo tempo. O uso das ferramentas Live Share para o compartilhamento de código e do Discord para a comunicação em equipe foi o que viabilizou o uso da técnica citada.

2.1 Codificação do autômato

Para a codificação do autômato foram criados dois arquivos: automato.h, responsável por guardar a prototipagem usada nas funções e automato.c, contendo a implementação das funções em si.

Começando pelo automato.h, temos o seguinte código:

```
1  #pragma once
2
3  typedef struct {
4    int dimension;
5    int **reticulated;
6  } CelularAutomaton;
7
8  void deallocateReticulated(CelularAutomaton *automaton);
9
10 void readReticulated(CelularAutomaton *automaton);
11
12 CelularAutomaton evolveReticulated(CelularAutomaton automaton);
13
14 void printReticulated(CelularAutomaton automaton);
```

Código 1: Headers do autômato.

- Linha **#pragma once** é um diretiva de pré-processador que garante que o arquivo só será incluído uma única vez no programa. Isso é útil para evitar problemas de inclusão múltipla.
- A estrutura **CelularAutomaton** armazena a dimensão da matriz reticulada e um ponteiro para a matriz. A matriz é representada como uma matriz de inteiros, onde cada célula pode ser 0 (morta) ou 1 (viva). Esta estrutura vai facilitar o manuseio dos dados.
- A função **deallocateReticulated(CelularAutomaton *automaton)**libera a memória alocada para a matriz reticulada. Esse passo é importante para evitar vazamentos de memória e problemas de performance.
- A função **readReticulated(CelularAutomaton *automaton)** lê os valores da matriz a partir de um arquivo de entrada e armazena-os na estrutura automaton. Dessa forma, é possível carregar estados previamente salvos do jogo.
- A função **evolveReticulated(CelularAutomaton automaton)** aplica as regras do jogo da vida na matriz fornecida e retorna uma estrutura CelularAutomaton com o estado evoluído. Essa é a parte importante da implementação onde ocorre o evoluir das celulas de acordo com as regras do jogo.
- A função **printReticulated(CelularAutomaton automaton)** imprime os valores da matriz reticulada em formato legível para o usuário. Isso permite que o usuário veja visualmente a evolução do jogo.

No arquivo automato.c temos a implementação das funções acima. Vale destacar as funções **countLiveNeighbors**, que não está na listagem por ser exclusiva do arquivo automato.c, e a **evolveReticulated**, que é a responsável pela principal finalidade do projeto. Outra função

não listada, mas de menor importância é a **allocateReticulated**, responsável pela alocação dinâmica das matrizes.

A seguir, o código da função **countLiveNeighbors**:

```
// Funcao para verificar as celulas vivas vizinhas
int countLiveNeighbors(CelularAutomaton automaton, int row, int col)
{
int count = 0;
// Checa as principais çõdirees
if (col > 0 && automaton.reticulated[row][col - 1] == 1)
{
count++;
}

if (col < automaton.dimension - 1 && automaton.reticulated[row][col + 1] == 1)
{
count++;
}

if (row > 0 && automaton.reticulated[row - 1][col] == 1)
{
count++;
}

if (row < automaton.dimension - 1 && automaton.reticulated[row + 1][col] == 1)
{
count++;
}

// Checa as diagonais
if (row > 0 && col > 0 && automaton.reticulated[row - 1][col - 1] == 1)
{
count++;
}

if (row > 0 && col < automaton.dimension - 1 && automaton.reticulated[row -
    1][col + 1] == 1)
{
count++;
}

if (row < automaton.dimension - 1 && col > 0 && automaton.reticulated[row +
    1][col - 1] == 1)
{
count++;
}

if (row < automaton.dimension - 1 && col < automaton.dimension - 1 &&
    automaton.reticulated[row + 1][col + 1] == 1)
{
count++;
}

return count;
}
```

Código 2: Função countLiveNeighbors.

A função acima é uma função para contar o número de células vizinhas vivas a uma dada célula em uma matriz reticulada, que é representada pela estrutura **CelularAutomaton**. A função recebe como parâmetros a estrutura automaton, uma linha (*row*) e uma coluna (*col*) da

matriz reticulada, que representam a posição da célula para a qual se deseja contar as células vizinhas vivas.

A função começa inicializando uma variável *count* com zero, que será usada para armazenar o número de células vizinhas vivas. Em seguida, a função verifica se as células nas posições imediatamente à esquerda, à direita, acima, abaixo, e nas diagonais da célula dada estão vivas. Isso é feito usando as condições de verificação nos *if statement*, que verificam se a posição está dentro dos limites da matriz e se o valor da célula na posição é igual a 1. Se a célula estiver viva, a variável *count* é incrementada.

No final da função, o número de células vizinhas vivas é retornado como o valor de retorno. Com essa função, é possível contar quantas células vivas estão no entorno de uma dada célula, o que é uma das regras para o jogo da vida.

A seguir está a função **evolveReticulated**:

```
// Funcao para evoluir as celulas
CelularAutomaton evolveReticulated(CelularAutomaton automaton)
{
  int nextGen[automaton.dimension][automaton.dimension];
  for (int i = 0; i < automaton.dimension; i++)
  {
    for (int j = 0; j < automaton.dimension; j++)
    {
      int liveNeighborsCount = countLiveNeighbors(automaton, i, j);

      // Validacao das regras de vivo ou morto de acordo com o numero de
         vizinhos
      if (automaton.reticulated[i][j] == 1)
      {
        if (liveNeighborsCount < 2 || liveNeighborsCount > 3)
        {
          nextGen[i][j] = 0;
        }
        else
        {
          nextGen[i][j] = 1;
        }
      }

      else
      {
        if (liveNeighborsCount == 3)
        {
          nextGen[i][j] = 1;
        }
        else
        {
          nextGen[i][j] = 0;
        }
      }
    }
  }

  CelularAutomaton nextGenAutomaton;
  nextGenAutomaton.dimension = automaton.dimension;
  nextGenAutomaton.reticulated = allocateReticulated(nextGenAutomaton.
     dimension);

  for (int i = 0; i < nextGenAutomaton.dimension; i++)
  {
    for (int j = 0; j < nextGenAutomaton.dimension; j++)
```

```
    {
      nextGenAutomaton.reticulated[i][j] = nextGen[i][j];
    }
  }

  return nextGenAutomaton;
}
```

Código 3: Função countLiveNeighbors.

A função toma como entrada um objeto **CelularAutomaton** chamado automaton e retorna um novo objeto **CelularAutomaton** chamado *nextGenAutomaton*.

A função inicia criando uma matriz chamada nextGen que tem o mesmo tamanho da matriz reticulated dentro do objeto automaton de entrada. Ela então usa duas estruturas de loop for aninhadas para iterar sobre cada célula na matriz reticulated.

Para cada célula, a função usa outra função, a **countLiveNeighbors** para contar quantas células vivas existem nas células vizinhas. Isso é armazenado na variável *liveNeighborsCount*.

A função então usa uma estrutura de condição para verificar se a célula atual é viva ou morta. Se a célula atual é viva, então a função verifica se o número de células vivas vizinhas é menor do que 2 ou maior do que 3. Se isso for verdade, a célula na próxima geração será morta; caso contrário, ela será viva. Se a célula atual é morta, a função verifica se o número de células vivas vizinhas é igual a 3. Se isso for verdade, a célula na próxima geração será viva; caso contrário, ela será morta.

Depois de iterar sobre todas as células, a função cria um novo objeto **CelularAutomaton** chamado *nextGenAutomaton* e atribui à sua matriz *reticulated* os valores da matriz *nextGen*. Finalmente, a função retorna *2* como o resultado.

2.2 Implementação do autômato

Seguido pelo tp.c, o arquivo main:

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

int main(int argc, char **argv)
{
  CelularAutomaton automaton;

  readReticulated(&automaton);
  CelularAutomaton nextGenAutomaton = evolveReticulated(automaton);

  printReticulated(nextGenAutomaton);

  deallocateReticulated(&automaton);
  deallocateReticulated(&nextGenAutomaton);

  return 0;
}
```

Código 4: Função main.

No código main é criado o TAD "automaton", ademais é chamado as funções de leitura e evolução das células, posteriormente se tem a saída dos dados dos resultados e a desalocação das memórias dinâmicas usadas no programa.

3 Experimentos

Para os experimentos foram usados dois arquivos de entrada seguindo o modelo já explicado anteriormente. Os testes foram realizados no hardware especificado anteriormente.

Além do mais, para saber o tempo de execução e o uso de memória foi utilizado o *valgrind* e o *time* do linux. Cada entrada foi executada três vezes.

3.1 Entrada 1

O primeiro teste recebeu como entrada os seguintes dados:

```
15
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 1 0 1 0 1 1 1 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 1 0 1 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 1 1 0 0 0
0 0 0 1 0 1 0 1 1 0 1 0 1 0 0
0 0 0 0 0 1 1 0 1 0 1 0 1 1 0
0 0 0 0 0 1 0 1 1 0 0 1 1 0 0
0 1 0 0 0 1 0 1 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Código 5: Entrada 1.

A saída obtida foi:

```
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 1 1 0 0 0 0 0 0 0 0 1 0 0 0
0 1 1 0 0 1 1 1 0 1 1 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 1 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 1 0 0 0
0 0 1 1 1 0 0 0 0 1 1 1 0 0 0
0 0 0 0 1 1 0 1 1 0 1 0 1 1 0
0 0 0 0 0 1 0 0 0 0 1 0 0 1 0
0 0 0 0 1 1 0 0 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 1 1 0
0 0 0 0 0 0 1 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Código 6: Saída 1.

- Teste 1: Tempo de execução: 0.002s; Consumo de memória: 7,160 bytes;
- Teste 2: Tempo de execução: 0.003s; Consumo de memória: 7,160 bytes;
- Teste 3: Tempo de execução: 0.003s; Consumo de memória: 7,160 bytes;

3.2 Entrada 2

O segundo teste recebeu como entrada os seguintes dados:

```
20
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0 1 0 0
0 0 0 0 1 1 0 0 0 1 0 0 0 0 1 1 0 0 0 0
0 0 1 1 1 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0
0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0 0 1 1 1 0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0
0 1 1 0 0 1 0 0 1 0 0 0 1 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0
0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0
0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Código 7: Entrada 2.

A saída obtida foi:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0
0 0 0 1 1 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0
0 0 1 0 0 1 0 0 1 1 1 0 0 0 1 1 0 0 0 0
0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 0 0 1 1 1 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 1 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Código 8: Saída 2.

- Teste 1: Tempo de execução: 0.003s; Consumo de memória: 8,640 bytes;
- Teste 2: Tempo de execução: 0.002s; Consumo de memória: 8,640 bytes;
- Teste 3: Tempo de execução: 0.002s; Consumo de memória: 8,640 bytes;

4 Resultados

A seguir está um resumo dos resultados obtidos a partir de cada entrada, bem como os pontos fortes e fracos do código-fonte do trabalho.

4.1 Resumo dos experimentos

Entrada 1:

Teste	Tempo de execução (em segundos)	Consumo de memória (em bytes)
Teste1	0.002	7,160
Teste 2	0.003	7,160
Teste 3	0.003	7,160

Tabela 1: Resumo dos testes da entrada 1.

Entrada 2:

Teste	Tempo de execução (em segundos)	Consumo de memória (em bytes)
Teste1	0.003	8,640
Teste 2	0.002	8,640
Teste 3	0.002	8,640

Tabela 2: Resumo dos testes da entrada 2.

4.2 Pontos fortes

]

- O código é bem estruturado e fácil de ler, com comentários úteis explicando o que cada função faz.
- A alocação dinâmica de memória para armazenar a matriz do autômato é uma boa prática, pois permite que o tamanho do autômato seja definido dinamicamente e evita problemas de buffer overflow.
- A função countLiveNeighbors() conta corretamente as células vivas vizinhas a cada célula na matriz, garantindo que a evolução do autômato seja precisa.

4.3 Pontos fracos

- Não há nenhuma verificação de erro para a função fscanf na função readReticulated() o que pode causar erro caso o arquivo esteja incompleto ou tenha formato inválido.
- A função evolveReticulated() cria uma cópia da matriz do autômato antes de evoluir as células. Isso pode causar problemas de desempenho se a matriz for grande e essa cópia deve ser desalocada manualmente.

5 Considerações Finais

Inicialmente, nós analisamos o que foi pedido dentro do pdf do trabalho prático, buscamos compreender como funcionava o The Game of Life e como deveria ser implementado, após isso listamos as funções que seriam necessárias dentro do arquivo 'automato.h', ainda não havíamos entendido muito bem como implementar o TAD dentro do código então seguimos com o projeto.

Em seguida, partimos para o desenvolvimento das funções no arquivo 'automato.c', onde começamos pelas funções mais simples de alocação, desalocação, leitura do arquivo e de impressão da matriz com a próxima geração, essa parte consideramos mais simples, a função de evoluir a geração passamos mais tempo desenvolvendo e após vários testes conseguimos desenvolver com auxílio de uma outra função chamada countLiveNeighbors, que basicamente checa através de várias condições e retorna através de um contador quantas células vivas são vizinhas a posição da matriz passada, assim depois de um tempo checando conseguimos êxito.

Por fim, depois de um tempo pensando surgimos com a ideia de usar o TAD para armazenar o arquivo de entrada, contendo a dimensão da matriz e a própria matriz, sendo assim, gastamos um bom tempo implementando em cada função e foi mais cansativo para alocar de forma correta por ser um struct, mas enfim conseguimos e fomos para o debug, nessa fase gastamos mais tempo que as outras consertando erros de sintaxe e alocação, utilizamos o valgrind que foi muito útil para localizar os vazamentos de memória, assim que retiramos todos os bugs, finalizamos e enviamos no run.codes.

Em resumo, nós seguimos os passos de compreender e resolver o problema proposto de implementar um automato celular de acordo com as regras do The Game of Live. Começando pelas funções mais simples e evoluindo para as mais complexas que englobaram os ensinamentos passados durante as aulas e colocando todos conhecimentos já aprendidos à prova e passando pela fase de debug que foi importante para encontrar e corrigir erros, logrando êxito ao final na execução do projeto.