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

TP - 02

BCC202 - Estrutura de Dados

Jeanlucas Ferreira Santana
Professor: Pedro Henrique Lopes Silva

Ouro Preto
29 de agosto de 2024

Sumário

1 Introdução

Breve Introdução sobre o trabalho sendo feito

O problema em questão envolve a simulação de um autômato celular conhecido como Jogo da Vida de Conway. A tarefa é implementar um modelo que simula o comportamento de uma grade bidimensional de células, onde cada célula pode estar viva (`1`) ou morta (`0`). A grade evolui ao longo do tempo seguindo regras específicas que determinam o estado de cada célula na próxima geração com base no número de vizinhos vivos ao seu redor.

1.1 Especificações do problema

Descrição do problema a ser atacado (Ex: Precisamos encontrar a frequência das palavras num determinado conjunto de arquivos ou precisamos tornar transparente a programação paralela ou distribuída para o usuário)

1.2 Considerações iniciais

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 utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- *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: Intel Core i7-8750H.
- Memória RAM: 32Gb.
- Sistema Operacional: Windows 10, Ubuntu.

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

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

Usou-se para a compilação as seguintes opções:

- -*Wall*: para mostrar todos os possível *warnings* do código.
- -o: para linkar os arquivos objeto que compõem o programa e gerar o executável.

Para a execução do programa basta digitar:

```
./exe
```

Onde exe e o arquivo executável do projeto

[1] ????? está disponível em https://www.

[2] Disponível em https://www.overleaf.com/

2 Desenvolvimento

Descrição sobre a implementação do programa. Não faça ‘‘print screens’’ de telas. Ao contrário, procure resumir ao máximo a documentação, fazendo referência ao que julgar mais relevante. É importante, no entanto, que seja descrito o funcionamento das principais funções e procedimentos utilizados, bem como decisões tomadas relativas aos casos e detalhes de especificação que porventura estejam omissos no enunciado. Muito importante: os códigos utilizados na implementação devem ser inseridos na documentação. (Ex. Como foi feita a solução do trabalho, descrevendo os passos ou atividades envolvidas. Use figuras e diagramas para detalhar a solução escolhida)

2.1 Objetivo principal

O objetivo principal deste projeto é implementar o Jogo da Vida de Conway, um autômato celular, de forma que ele receba os valores de entrada do terminal. O programa deve:

1. **Receber Valores de Entrada:**
   - Tamanho da grade bidimensional usando listas encadeadas.
   - Número de gerações a serem simuladas.
   - Estado inicial dos elementos na grade.
2. **Processamento dos Dados:**
   - Ler os dados do terminal, garantindo que sejam valores inteiros válidos.
   - Ler a configuração inicial da grade bidimensional, composta por células que podem estar vivas (representadas por 1) ou mortas (representadas por 0).
3. **Evolução:**
   - Aplicar as regras do Jogo da Vida de Conway para evoluir a grade ao longo das gerações especificadas. As regras são:
     - Qualquer célula viva com menos de dois vizinhos vivos morre de solidão.
     - Qualquer célula viva com dois ou três vizinhos vivos continua viva para a próxima geração.
     - Qualquer célula viva com mais de três vizinhos vivos morre de superpopulação.
     - Qualquer célula morta com exatamente três vizinhos vivos se torna uma célula viva.
   - Uso de funcoes para progressao das celulas do reticulado(recursivas ou iterativas).
4. **Geração do Estágio Final:**
   - Imprimir a configuração inicial da grade.
   - Calcular e imprimir a configuração final da grade após todas as gerações terem sido processadas.

2.2 Implementação e Especificação

Utiliza-se quatro estruturas `listaLinhas listaColunas nCelula Celula` para representar a grade bidimensional:

```
typedef struct {
    nCelula* primeiraLinha;
} listaLinhas;
typedef struct{
    nCelula* primeiraColuna;
} listaColunas;
```

```c
typedef struct nCelula{
    Celula celula;
    struct nCelula* proximaLinha; //ponteiro para a proxima celula da mesma linha
    struct nCelula* proximaColuna; //ponteiro para a proxima celula da mesma coluna
} nCelula;
typedef struct{
    int linha; //coordenada que representa a linha em que a celula esta posicionada
    int coluna; //coordenada que representa a coluna em que a celula esta posicionada
} Celula;
```

Para manipular o reticulado de maneira eficiente, é necessário o uso de listas encadeadas para alocar apenas celulas vivas, para isso, foram usadas funcoes para inicializar as listas como NULL. Funcoes de desalocacao para a passagem de novas celulas para os novos reticulados de maneira correta e posteriormente encerrar o programa sem leaks de memoria tambem foram implementadas.

```c
void alocarReticulado(listaLinhas* linhas, listaColunas* colunas){

    //inicializa as listas vazias(abordagem sem cabeca)

    linhas -> primeiraLinha = NULL;
    colunas -> primeiraColuna = NULL;
}
//funcao para desalocar cada celula de uma lista
void desalocarListaCelulas(nCelula* primeiraCelula){
    while(primeiraCelula != NULL){
        nCelula *temp = primeiraCelula; //usa a variavel temp pra receber sempre a proxima celula pra desalocar
        primeiraCelula = primeiraCelula -> proximaLinha; //coloca a proxima celula na primeira posicao pra ir a temp e ser desalocada
        free(temp);
    }
}

//funcao para desalocar as listas

void desalocarReticulado(listaLinhas* linhas){

    //desaloca todas as listas, usando apenas uma ja que os elementos que estao em uma estao na outra //caso tentar desalocar as duas vai abortar por desalocacao da mesma memoria duas vezes

    desalocarListaCelulas(linhas -> primeiraLinha);

}
```

A função de desalocar o reticulado é responsável por liberar a memória alocada dinamicamente para os reticulados, garantindo que não haja vazamento de memória. Esta função percorre cada linha do reticulado, libera a memória de cada linha e, em seguida, libera a memória do ponteiro que armazena todas as linhas e, por fim, libera as outras informações presentes em um reticulado.

```c
void desalocarReticulado(Reticulado* reticulado){
    for(int i = 0; i < reticulado->tamanho; i++){
        free(reticulado->reticulado[i]);
    }
    free(reticulado->reticulado);
    free(reticulado);
}
```

A função `LeituraReticulado` tem o objetivo de ler o tamanho do reticulado e o número de gerações a partir da entrada do terminal, além de preencher a grade inicial do reticulado. A função realiza os seguintes passos:

1. **Recepção de Dados**: A função recebe dois ponteiros (`tamanho` e `geracoes`) que representam as variáveis para o tamanho do reticulado e o número de gerações, respectivamente.

2. **Leitura e Validação**: O usuário é solicitado a inserir o tamanho do reticulado e o número de gerações. A função usa a função `lerInteiro` para garantir que as entradas sejam inteiros válidos, com o tamanho sendo um número positivo e o número de gerações não negativo. Se os valores não forem válidos, a função continua solicitando entradas até que valores corretos sejam fornecidos.

3. **Preenchimento do Reticulado**: Em seguida, a função solicita ao usuário que insira os valores iniciais para o reticulado, que deve ser uma matriz bidimensional. A função valida que os valores inseridos sejam apenas 0 ou 1, e preenche a grade do reticulado com esses valores.

4. **Alocação das celulas**: Após validar as entradas, a função utiliza as função `alocarReticulado` `adicionarCelulaLinha` `adicionarCelulaColuna`para criar as novas celulas, passar suas coordenadas e armazena-las nas respectivas listas.

5. **Retorno**: A função por ser do tipo void nao possui retorno.

```c
//funcao para leitura do reticulado

void leituraReticulado(listaLinhas* linhas, listaColunas* colunas, int* geracoes, int* tam){

    //verificacao para o tamanho do reticulado e do numero de geracoes

    printf("Digite o tamanho do reticulado e o numero de geracoes:\n");
    while(!lerInteiro(tam) || *tam <= 0 || !lerInteiro(geracoes) || *geracoes < 0){
        printf("Entrada invalida. Por favor, insira valores inteiros positivos para o tamanho do reticulado e nao negativos para o numero de geracoes:\n");
    }

    //inicializar as listas vazias

    alocarReticulado(linhas, colunas);

    //leitura dos valores do reticulado

    printf("Digite o reticulado inicial (%d linhas com %d colunas):\n",*tam, *tam);
    for(int i = 0; i < *tam; i++){
        for(int j = 0; j < *tam; j++){
            int valor; // variavel para conferir se o valor inserido pelo usuario esta correto
            while(!lerInteiro(&valor) || (valor != 0 && valor!= 1)){
                printf("Entrada invalida. Por favor, insira 0 ou 1 para a celula (%d, %d):\n", i, j);
            }
            if(valor == 1){

                //adicionar celula viva as listas
                nCelula* novaCelula = malloc(sizeof(nCelula));
                novaCelula -> celula.linha = i;
```

```
                novaCelula -> celula.coluna = j;
                novaCelula -> proximaLinha = NULL;
                novaCelula -> proximaColuna = NULL;

                adicionarCelulaLinha(linhas, novaCelula); // adiciona a nova
                    celula na lista de linhas
                adicionarCelulaColuna(colunas, novaCelula); // adiciona a nova
                     celula na lista de colunas
            }
        }
    }
}
}
```

A função `lerInteiro` é responsável por ler e validar a entrada de inteiros fornecidos pelo usuário. Ela recebe um ponteiro para uma variável inteira, lê o valor do terminal e armazena-o na variável apontada. A função verifica se a entrada é um inteiro válido e atende aos requisitos especificados; se não for, ela limpa o buffer de entrada e solicita uma nova entrada até que um valor aceitável seja fornecido.

```
void limparBuffer(){
    int c;
    while((c = getchar()) != '\n' && c != EOF);
}

bool lerInteiro(int* valor){
    int resultado = scanf("%d", valor);
    if(resultado != 1){
        limparBuffer();
    }
    return resultado == 1;
}
```

Para determinar o estado futuro de uma célula em um reticulado, é crucial saber quantos vizinhos vivos ela possui. Isso é feito por meio de uma função que calcula a quantidade de vizinhos vivos ao redor de uma célula específica. A função primeiro verifica os vizinhos vivos em torno das celulas vivas nas listas, percorrendo todas elas, e suas extremidades. Percorre-se novamente as celulas para verificar se alguma delas possui as coordenadas envolto da celula a ser analisada. Para as celulas mortas, usa-se um for para apenas percorrer coordenadas e suas extremidades e asssim compara-las com as das celulas vivas, para determinar qual posicao se tornara viva. As funcoes adicionarCelulaLinha e adicionarCelulaColuna, sao usadas para fazer a passagem das novas celulas para o novo reticulado. Que sera usado para as evolucoes seguintes.

```
void evoluirReticulado(listaLinhas* linhas, listaColunas* colunas,
    listaLinhas* novasLinhas, listaColunas* novasColunas, int tam) {
    // Inicializa as novas listas
    alocarReticulado(novasLinhas, novasColunas); // Setar as novas listas como
        nulas


    nCelula* celula = linhas->primeiraLinha;

    while (celula != NULL) {
        int linha = celula->celula.linha;
        int coluna = celula->celula.coluna;


        int vizinhosVivos = 0;

        for (int i = linha - 1; i <= linha + 1; i++) {
            for (int j = coluna - 1; j <= coluna + 1; j++) {
                if (i == linha && j == coluna) continue;

```

```
19
20                  nCelula* checarLinha = linhas->primeiraLinha;
21                  while (checarLinha != NULL) {
22                      if (checarLinha->celula.linha == i && checarLinha->celula.
                            coluna == j) {
23                          vizinhosVivos++;
24                          break;
25                      }
26                      checarLinha = checarLinha->proximaLinha;
27                  }
28              }
29          }
30
31          if (vizinhosVivos == 2 || vizinhosVivos == 3) {
32
33              nCelula* novaCelula = malloc(sizeof(nCelula));
34              novaCelula->celula.linha = linha;
35              novaCelula->celula.coluna = coluna;
36              novaCelula->proximaLinha = NULL;
37              novaCelula->proximaColuna = NULL;
38
39
40              adicionarCelulaLinha(novasLinhas, novaCelula);
41              adicionarCelulaColuna(novasColunas, novaCelula);
42          }
43
44
45          celula = celula->proximaLinha;
46      }
47
48
49      for (int i = 0; i < tam; i++) {
50          for (int j = 0; j < tam; j++) {
51              int vizinhosVivos = 0;
52
53              for (int x = i - 1; x <= i + 1; x++) {
54                  for (int y = j - 1; y <= j + 1; y++) {
55                      if (x == i && y == j) continue;
56
57                      nCelula* checarLinha = linhas->primeiraLinha;
58                      while (checarLinha != NULL) {
59                          if (checarLinha->celula.linha == x && checarLinha->
                                celula.coluna == y) {
60                              vizinhosVivos++;
61                              break;
62                          }
63                          checarLinha = checarLinha->proximaLinha;
64                      }
65                  }
66              }
67
68              if (vizinhosVivos == 3) {
69
70                  nCelula* novaCelula = malloc(sizeof(nCelula));
71                  novaCelula->celula.linha = i;
72                  novaCelula->celula.coluna = j;
73                  novaCelula->proximaLinha = NULL;
74                  novaCelula->proximaColuna = NULL;
75
76                  adicionarCelulaLinha(novasLinhas, novaCelula);
77                  adicionarCelulaColuna(novasColunas, novaCelula);
78              }
```

```
        }
    }


    desalocarReticulado(linhas);


    *linhas = *novasLinhas;
    *colunas = *novasColunas;


    novasLinhas->primeiraLinha = NULL;
    novasColunas->primeiraColuna = NULL;
}
}
```

A função `imprimeReticulado` é responsável por imprimir a grade do reticulado na saída padrão.

```

    for (int i = 0; i < tam; i++){


        for (int j = 0; j < tam; j++){

            int encontrada = 0;   v


            nCelula* celulaLinha = linhas->primeiraLinha;
            while (celulaLinha != NULL) {
                //verifica se a celula atual da lista corresponde a celula (i,
                    j)
                if (celulaLinha->celula.linha == i && celulaLinha->celula.
                    coluna == j) {
                    encontrada = 1;  // a celula (i, j) foi encontrada
                    break;
                }
                celulaLinha = celulaLinha->proximaLinha;
            }


            printf("%d ", encontrada);
        }
        printf("\n");
    }
}
```

A função `main` gerencia o ciclo completo do programa, começando pela leitura dos dados e alocaçao das listas. Inicialmente, a função `LeituraReticulado` coleta o tamanho da grade e o número de gerações, alocando e preenchendo o reticulado inicial com os valores fornecidos pelo usuário. Em seguida, e feita uma verificacao para caso o numero de geracoes for 0(imprimir a matriz original).

A função `imprimeReticulado` é chamada para exibir a configuração inicial do reticulado antes de qualquer evolução. Posteriormente, Apos isso e feito o loop para evolucao progressiva do automato celular.

Após a evolução, o reticulado final é exibido novamente usando a função `imprimeReticulado`. Por fim, a memória alocada para evolucao do automato eliberada com a função `desalocarReticulado`, garantindo que todos os recursos sejam adequadamente liberados e que não haja vazamento de memória.

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

```c
#include "matriz.h"
#include "automato.h"

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

int main() {
    int tam;
    int geracoes;


    listaLinhas linhas;
    listaColunas colunas;
    alocarReticulado(&linhas, &colunas);

    listaLinhas novasLinhas;
    listaColunas novasColunas;
    alocarReticulado(&novasLinhas, &novasColunas);


    leituraReticulado(&linhas, &colunas, &geracoes, &tam);


    if (geracoes == 0) {
        printf("Reticulado Inicial:\n");
        imprimeReticulado(&linhas, tam);
        desalocarReticulado(&linhas);
        desalocarReticulado(&novasLinhas);
        return 1;
    }

    printf("Reticulado Inicial:\n");
    imprimeReticulado(&linhas, tam);


    for (int geracao = 0; geracao < geracoes; geracao++) {
        evoluirReticulado(&linhas, &colunas, &novasLinhas, &novasColunas, tam)
            ;

        alocarReticulado(&novasLinhas, &novasColunas);
    }

    printf("Reticulado Final:\n");
    imprimeReticulado(&linhas, tam);

    desalocarReticulado(&linhas);
    desalocarReticulado(&novasLinhas);


    return 0;
}
```

3 Experimentos

Descreva os experimento feitos, ou seja, o que você está medindo e em qual hardware você está fazendo tal medição. Normalmente, buscamos na computação medir o tempo de execução e o consumo de memória.

O código foi testado no Ubuntu para verificar sua funcionalidade, e o Valgrind foi utilizado para avaliar a eficiência da alocação de memória. Durante os testes, observou-se que o programa usou uma quantidade mínima de memória e não apresentou erros, o que é um indicativo de um desempenho eficiente e estável.

==11257== HEAP SUMMARY: ==11257== in use at exit: 0 bytes in 0 blocks ==11257== total heap usage: 50 allocs, 50 frees, 3,200 bytes allocated ==11257== ==11257== All heap blocks were freed – no leaks are possible ==11257== ==11257== For lists of detected and suppressed errors, rerun with: -s ==11257== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

4 Resultados

Descreva os resultados e comente os mesmos, critique ou elogie sua solução nos pontos fracos e fortes, respectivamente. Coloque gráficos que mostre, por exemplo, o tempo de execução das n soluções propostas. Ex. No caso da contagem das palavras - coloque um gráfico com o tempo de execução da solução com a primeira proposta, com a segunda proposta, etc..., mostrando assim a viabilidade da solução proposta).

Gostei bastante da minha implementação para a resolução do problema. Realizei verificações importantes e eficientes. Acredito que o uso de listas encadeadas e suas aplicacoes praticas, me deram um embasamento forte sobre eficiencia e processamento.

5 Considerações Finais

Comentários gerais sobre o trabalho e as principais dificuldades encontradas em sua implementação. Descreva o seu processo de implementação deste trabalho. Aponte coisas que gostou bem como aquelas que o desagradou. Avalie o que o motivou, conhecimentos que adquiriu, entre outros. Gostei bastante do trabalho, que me deu uma clareada significativa sobre listas encadeadas. Compreendi que posso usar esse recurso para melhorar o processamento de determinadas aplicacoes e ter uma eficiencia maior no uso da memoria, alem de claro, um tempo de processamento muito mais rapido.

Referências

/section