Trabalho Prático 0 em Estruturas de Dados
Operações com Matrizes Alocadas Dinamicamente

Gustavo Freitas Cunha

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brazil

gustavocunha@dcc.ufmg.br

1. Introdução

Uma matriz é uma tabela de dados organizados em linhas e colunas. Muito utilizadas em diversas áreas como matemática, computação, física e estatística, tal estrutura é normalmente associada a outras iguais, de onde surge a necessidade de realizar operações dentre estas, com o fito de obter manipulações tabulares dos dados que contêm. Neste trabalho, no âmbito da disciplina Estruturas de Dados, serão implementadas as operações de adição, multiplicação e transposição de matrizes alocadas dinamicamente. O usuário do projeto terá controle sobre as matrizes a serem operadas, suas dimensões ou arquivos em que ocorrerão as operações e também poderá utilizar-se de registro de acesso para avaliação de desempenho. A construção do software se dará pela utilização de um ambiente de desenvolvimento de código, mirando a obtenção de um programa que disponibilize para o usuário tais operações de maneira não apenas funcional, mas eficiente, robusta e interativa.

2. Implementação

O programa foi desenvolvido e testado em sistema operacional Subsistema Windows para Linux (Windows Subsystem for Linux - WSL), na linguagem C, compilado pelo compilador GCC da GNU Compiler Collection, em processador Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.70 GHz, com 8GB de RAM instalada.

2.1 Organização do Código

Este projeto foi estruturado da seguinte forma:

```
- TP
  |- src
  |- bin
  |- obj
  |- include
  Makefile
```

O diretório `src` armazena os arquivos de código (extensão `.c`), o diretório `include` os arquivos de cabeçalho (extensão `.h`), o diretório `obj` armazena os arquivos objects (extensão `.o`) e o diretório `bin` armazena os executáveis (`.exe`).

Toda essa estruturação está configurada no Makefile disponível no diretório raiz.

Os diretórios `obj` e `bin` foram entregues vazios, conforme instrução, mas após compilação e linkagem pelo Makefile, os arquivos gerados são alocados em suas respectivas pastas.

2.2 Estrutura de Dados

Foi utilizada uma `struct` nomeada `mat_tipo` que armazena um vetor de vetores `m`, do tipo `double**`, as dimensões da matriz (`tamx` e `tamy`) e um identificador (`id`) para registro de acesso. Todas as chamadas e utilizações de funções e dessa `struct` serão abordadas nas duas próximas subseções.

2.3 Funcionamento do Programa

O programa principal é `matop.c`. É nele que está a função `main` e é feita a leitura dos argumentos passados na linha de comando, bem como a chamada das funções correspondentes. Para que o código pudesse ser mais reutilizável e modularizado, as funções operacionais sobre as matrizes, bem como as funções que alocam, desalocam, leem e escrevem matrizes em arquivos têm sua especificação no programa `mat.c`. Já o programa `mat.h` contém as assinaturas dessas funções, bem como a declaração da `struct mat_tipo`. Os programas `memlog.h` e `memlog.c` são auxiliares no quesito de análise de desempenho. Já o programa `mgassert.h` diz respeito às estratégias de robustez adotadas.

Você pode fazer a execução do programa utilizando flags para especificar as operações de interesse, abaixo estão listadas as flags disponíveis e suas respectivas atribuições:

```
-s    (soma matrizes)

-m    (multiplica matrizes)

-t    (transpõe matriz)

-1    (endereço e nome do arquivo em que está matriz 1, utilizada para
operações)

-2    (endereço e nome do arquivo em que está matriz 2, utilizada para
operações)

-o    (endereço e nome do arquivo em que será registrada a matriz
resultante das operações)

-p    (endereço e nome do arquivo de registro de desempenho)

-l    (padrão de acesso e localidade)

-x    (dimensão x das matrizes a serem geradas aleatoriamente para
operações)

-y    (dimensão y das matrizes a serem geradas aleatoriamente para
operações)
```

No APÊNDICE A, serão dadas mais instruções sobre a compilação e execução do programa.

O programa principal possui um mecanismo de identificação e leitura das flags e seus argumentos, de modo a executar apenas o solicitado, passando as informações lidas por um `switch`. Em cada `case` do `switch`, são chamadas funções implementadas em `mat.c`, como a `leMatrizArquivo` (se o usuário decidir por entrar com matrizes já prontas em um arquivo), que, como o próprio nome diz, faz a leitura das matrizes no arquivo indicado e, internamente, chama a função `criaMatriz`, que inicializa as variáveis da `struct mat_tipo` com os valores lidos e faz a alocação dinâmica do vetor de vetores `m`.

Caso o usuário opte por gerar as matrizes aleatoriamente, ele **NÃO** poderá entrar com matrizes em arquivos (e vice-versa), e serão chamadas as funções `criaMatriz` e `inicializaMatrizAleatoria`.

Após essa fase, as chamadas variam conforme a operação escolhida (soma, multiplicação ou transposição). Ao final das operações, a matriz resultante é impressa no terminal (função `imprimeMatriz`) e no arquivo solicitado (função `imprimeMatrizArquivo`), caso o usuário opte por trabalhar com matrizes em arquivos, e todas as variáveis `mat_tipo` têm seus parâmetros invalidados e sua memória desalocada dinamicamente pela função `destroiMatriz`.

2.3 Funções Operacionais

As principais funções operacionais e seus funcionamentos são descritos a seguir:

- `inicializaMatrizNula`: recebe um `mat_tipo` e atribui a todos os elementos de `m` o valor 0.
- `somaMatrizes`: recebe três `mat_tipo`'s `a`, `b` e `c` por referência, soma cada uma das posições correspondentes dos dois primeiros e armazena o resultado no terceiro, com o cuidado de alocar a matriz de `c` com dimensões compatíveis e verificar se `a` e `b` possuem dimensões que permitam a operação.
- `multiplicaMatrizes`: recebe três `mat_tipo`'s `a`, `b` e `c` por referência, multiplica as matrizes dos dois primeiros e armazena o resultado no terceiro, com o cuidado de alocar a matriz de `c` com dimensões compatíveis e verificar se `a` e `b` possuem dimensões que permitam a operação.
- `transpoeMatriz`: recebe **um, e somente um** `mat_tipo` `a` por referência, faz uma cópia de sua matriz em um `mat_tipo` internamente declarado, denominado `cpy`, desaloca a matriz de `a` (`a->m`), e aloca novamente com as dimensões transpostas. Após isso, o elemento `a->m[i][j]` recebe o elemento `cpy->m[j][i]`.
- `confereMatrizArquivo`: essa função possui um papel importantíssimo de robustez para o funcionamento do programa: ela verifica se as matrizes dos arquivos estão completas e correspondem às dimensões contidas na primeira linha do arquivo. Em caso negativo, o programa é **ABORTADO**, por meio de uma asserção, e um aviso é exibido.

2.4 Observações importantes

- Foi definido, em `mat.h`, uma macro chamada `MAXTAM` que define o **tamanho máximo permitido das matrizes**, sejam elas oriundas de um arquivo ou geradas aleatoriamente. Esse tamanho foi setado em **1000**, por segurança, mas pode ser facilmente alterado no arquivo indicado.

- Como já foi dito anteriormente, mas vale reforçar: as opções de **leitura de matrizes em arquivos** (`-1`, `-2` e `-o`) e **geração de matrizes aleatórias** (`-x` e `-y`) são **EXCLUDENTES**, não podendo serem executadas simultaneamente, há uma asserção que aborta o programa, se for o caso.

- A opção de transposição de matrizes (`-t`) só faz a transposição, no caso de leitura de arquivos, da matriz armazena no arquivo de endereço passado na flag `-1`, caso o usuário entre também com a flag -2 e algum argumento, **o programa será abortado** por uma asserção que indicará que deve-se entrar apenas com uma matriz.

- Ao entrar com a flag `-l` (padrão de acesso e localidade), o usuário deverá, **OBRIGATORIAMENTE**, entrar com a flag `-p`, para indicar o arquivo em que serão registrados os acessos à memória, caso não o faça, o programa será **abortado** por uma asserção que solicitará a entrada da flag e seu argumento. Note que o contrário não é obrigatório: ao entrar com a flag `-p`, o usuário não precisa entrar com a flag `-l`, mas ele só deverá o fazer caso queira que sejam registrados os acessos à memória. Caso contrário, apenas serão registrados o tempo inicial e final de execução do programa no arquivo do argumento da flag `-p`.

- Algumas funções, como a `drand48`, utilizada em `incializaMatrizAleatoria`, **não possuem implementações em Windows**.

3. Análise de Complexidade

Para as análises de complexidade em notação assintótica das funções dadas a seguir, é levado em consideração que as matrizes de entrada têm n linhas e m colunas.

3.1 Complexidade de Tempo

- `alocaMatriz`: considerando a função `malloc` com complexidade constante, I.E., $O(1)$, como temos nessa função apenas declarações de variáveis, uma execução da função `malloc` independente e, em seguida, uma execução da função `malloc` em um laço que percorre as linhas da matriz, temos: $O(1) + O(n) = O(n)$.

- `desalocaMatriz`: considerando a função `free` com complexidade constante, I.E., $O(1)$, como temos nessa função apenas declarações de variáveis, uma execução da função `free` independente ao final e, em antes, uma execução da função `free` em um laço que percorre as linhas da matriz, temos: $O(n) + O(1) = O(n)$.

- `criaMatriz`: nesta função temos apenas atribuições $(O(1))$ e uma chamada da função `alocaMatriz` $(O(n))$, temos: $O(n) + O(1) = O(n)$.

- `destroiMatriz`: nesta função temos uma chamada para `desalocaMatriz` $(O(n))$ e duas atribuições $(O(1))$. Logo, a complexidade é $O(n) + O(1) = O(n)$.

- `inicializaMatrizNula`: nesta função temos dois laços aninhados que, juntos, percorrem todos os elementos da matriz, com uma atribuição $(O(1))$ interna ao segundo laço $(O(nm))$. Logo, a complexidade é $O(nm)$.

- `inicializaMatrizAleatoria`: nesta função temos uma chamada a `inicializaMatrizNula` $(O(nm))$, e dois laços aninhados que, juntos, percorrem todos os elementos da matriz, com uma atribuição de valor aleatório dado pela função `drand48`, a qual consideramos $O(1)$, interna ao segundo laço $(O(nm))$. Logo, a complexidade é $O(nm) + O(nm) = O(nm)$.

- `acessaMatriz`: nesta função temos dois laços aninhados que, juntos, percorrem todos os elementos da matriz, com uma atribuição e soma $(O(1))$ internas ao segundo laço $(O(nm))$. Logo, a complexidade é $O(nm)$.

- `imprimeMatriz`: nesta função temos algumas impressões tabulares $(O(1))$, um laço que percorre apenas as colunas com a finalidade de formatação e numeração das colunas $(O(m))$ e dois laços aninhados que, juntos, percorrem todos os elementos da matriz, com impressões $(O(1))$ internas ao segundo laço $(O(nm))$. Logo, a complexidade é $O(nm)$.

- `copiaMatriz`: nesta função temos uma chamada a `criaMatriz` $(O(n))$ `inicializaMatrizNula` $(O(nm))$, e dois laços aninhados que, juntos, percorrem todos os elementos da matriz, com uma atribuição $(O(1))$ interna ao segundo laço $(O(nm))$. Logo, a complexidade é $O(n) + O(nm) + O(nm) = O(nm)$.

- `somaMatrizes`: nesta função temos uma chamada a `criaMatriz`

($O(n)$), `inicializaMatrizNula` ($O(nm)$), e dois laços aninhados que, juntos, percorrem todos os elementos da matriz, com uma atribuição e soma ($O(1)$) internas ao segundo laço ($O(nm)$). Logo, a complexidade é $O(n)$ + $O(nm)$ + $O(nm)$ = $O(nm)$.

- `multiplicaMatrizes`: nesta função temos uma chamada a `criaMatriz` ($O(n)$), `inicializaMatrizNula` ($O(nm)$), e três laços aninhados que fazem a multiplicação de duas matrizes, com uma atribuição e soma (O(1)) internas ao terceiro laço ($O(nm^2)$). Logo, a complexidade é $O(n)$ + $O(nm)$ + $O(nm^2)$ = $O(nm^2)$.

- `transpoeMatrizes`: nesta função temos uma chamada para `copiaMatriz` ($O(nm)$), uma chamada para `desalocaMatriz` ($O(n)$), uma chamada para `alocaMatriz` ($O(n)$), uma chamada para `inicializaMatrizNula` ($O(nm)$) e, finalmente, dois laços aninhados que, juntos, percorrem todos os elementos da matriz, com uma atribuição ($O(1)$) interna ao segundo laço ($O(nm)$). Logo, sua complexidade é dada por $O(nm)$ + $O(n)$ + $O(n)$ + $O(nm)$ + $O(nm)$ = $O(nm)$.

- `leMatrizArquivo`: nesta função temos uma chamada para `confereMatrizArquivo` ($O(nm)$), a declaração de algumas variáveis auxiliares e ponteiro para arquivo ($O(1)$), abertura do arquivo ($O(1)$), leitura de 2 dados do arquivo ($O(1)$), chamada da função `criaMatriz` ($O(n)$), dois laços aninhados que percorrem todos os elementos da matriz no arquivo, lendo-os ($O(nm)$) e, por fim, o fechamento do arquivo ($O(1)$). Logo, a complexidade desta função é $O(nm)$ + $O(1)$ + $O(1)$+ $O(1)$ + $O(n)$ + $O(nm)$ + $O(1)$ = $O(nm)$.

- `imprimeMatrizArquivo`: nesta função temos a declaração de algumas variáveis auxiliares e ponteiro para arquivo ($O(1)$), abertura do arquivo($O(1)$), escrita de 2 dados do arquivo ($O(1)$), dois laços aninhados que percorrem todos os elementos da matriz, escrevendo-os no arquivo ($O(nm)$) e, por fim, o fechamento do arquivo ($O(1)$). Logo, a complexidade desta função é $O(1)$ + $O(1)$+ $O(1)$ + $O(nm)$ + $O(1)$ = $O(nm)$.

- `confereMatrizArquivo`: nesta função temos a declaração de algumas variáveis auxiliares ($O(1)$), a abertura do arquivo passado como parâmetro ($O(1)$), leitura das dimensões na primeira linha do arquivo ($O(1)$), um laço que navega por todas as posições do arquivo com incrementos, condicionais e asserções ($O(mn)$) e, finalmente, o fechamento do arquivo ($O(1)$). Então a complexidade da função é dada por $O(mn)$.

3.2 Complexidade de Espaço

- `alocaMatriz`: ao receber as dimensões n e m, faz a alocação dinâmica demandada. Logo, $O(nm)$, já que as demais alocações de variáveis auxiliares são $O(1)$.

- `desalocaMatriz`: ao receber as dimensões n e m, faz a desalocação demandada, acessando as porções de memória. Logo, $O(nm)$, já que as demais alocações auxiliares são $O(1)$.

- `criaMatriz`: possui chamada para `alocaMatriz` ($O(nm)$), e algumas alocações de variáveis auxiliares ($O(1)$). Logo, $O(nm)$.

- `inicializaMatrizNula`: utiliza matriz já alocada anteriormente, demandando apenas alocações de variáveis auxiliares ($O(1)$). Logo, $O(1)$.

- `inicializaMatrizAleatoria`: utiliza matriz já alocada anteriormente, demandando apenas alocações de variáveis auxiliares ($O(1)$) e uma chamada para `inicializaMatrizNula` ($O(1)$). Logo, $O(1) + O(1) = O(1)$.

- `acessaMatriz`: utiliza matriz já alocada anteriormente, demandando apenas alocações de variáveis auxiliares ($O(1)$). Logo, $O(1)$.

- `imprimeMatriz`: utiliza matriz já alocada anteriormente, demandando apenas alocações de variáveis auxiliares ($O(1)$). Logo, $O(1)$.

- `copiaMatriz`: esta função tem uma chamada para `criaMatriz` ($O(nm)$), uma chamada para `inicializaMatrizNula` ($O(1)$), algumas alocações de variáveis auxiliares ($O(1)$) e utiliza algumas matrizes alocadas fora de seu escopo. Logo, tem complexidade $O(nm) + O(1) + O(1) = O(mn)$.

- `somaMatrizes`: esta função tem uma chamada para `criaMatriz` ($O(nm)$), uma chamada para `inicializaMatrizNula` ($O(1)$), algumas alocações de variáveis auxiliares ($O(1)$) e utiliza algumas matrizes alocadas fora de seu escopo. Logo, tem complexidade $O(nm) + O(1) + O(1) = O(mn)$.

- `multiplicaMatrizes`: esta função tem uma chamada para `criaMatriz` ($O(nm)$), uma chamada para `inicializaMatrizNula` ($O(1)$), algumas alocações de variáveis auxiliares ($O(1)$) e utiliza algumas matrizes alocadas fora de seu escopo. Logo, tem complexidade $O(nm) + O(1) + O(1) = O(mn)$.

- `transpoeMatrizes`: esta função realiza uma alocação estática de mat_tipo ($O(1)$), faz uma chamada de `copiaMatriz` ($O(nm)$), uma chamada de `desalocaMatriz` ($O(nm)$), uma chamada de `alocaMatriz` ($O(nm)$), uma chamada de `inicializaMatrizNula` ($O(1)$) e algumas alocações de variáveis auxiliares ($O(1)$). Logo, sua complexidade é dada por $O(1) + O(mn) + O(mn) + O(mn) + O(1) + O(1) = O(mn)$.

- `destroiMatriz`: possui apenas chamada para `desalocaMatriz` ($O(nm)$). Logo, $O(nm)$.

- `leMatrizArquivo`: nesta função temos a declaração de algumas variáveis auxiliares e ponteiro para arquivo ($O(1)$) e uma chamada para `criaMatriz` ($O(nm)$). Sua complexidade é dada, então, por $O(1) + O(mn) = O(mn)$.

- `imprimeMatrizArquivo`: nesta função temos a declaração de algumas variáveis auxiliares e ponteiro para arquivo ($O(1)$), somente. Logo, sua complexidade é dada por $O(1)$.

- `confereMatrizArquivo`: nesta função temos, no que tange à espaço, apenas alocações de variáveis auxiliares ($O(1)$). Logo, $O(1)$.

4. Estratégias de Robustez

Como estratégias de robustez foram utilizadas as macros definidas em `mgassert.h`. Dentre as macro definidas, temos:

- `avisoAssert`: recebe uma expressão lógica e uma mensagem. Quando a expressão lógica falha, é chamado o método `__avisoAssert` e enviadas informações sobre a linha e o arquivo em que foi detectada a falha.
- `__avisoAssert`: recebe uma expressão lógica, uma mensagem e o arquivo e linha em que a falha foi detectada. Exibe a mensagem e aborta o programa. Essa macro foi utilizada em possíveis erros com os quais a execução do programa ficaria comprometida.
- `erroAssert`: recebe uma expressão lógica e uma mensagem. Quando a expressão lógica falha, é chamado o método `__erroAssert` e enviadas informações sobre a linha e o arquivo em que foi detectada a falha.
- `__erroAssert`: recebe uma expressão lógica, uma mensagem e o arquivo e linha em que a falha foi detectada. Exibe a mensagem, mas não aborta o programa. Essa macro foi utilizada em possíveis erros com os quais a execução do programa segue normalmente.

Tais mecanismos foram implementados com o fito de tornar o programa mais interativo e dinâmico, além de facilitar o reconhecimento de falhas das mais diversas naturezas:

alocação de memória, abertura de arquivo, verificação de entradas válidas, entre outros.

5. Análise Experimental

Utilizando a biblioteca `memlog`, foram distribuídas pelo código chamadas de suas funções `leMemLog` e `escreveMemLog`, as quais possibilitam registro do endereço, identificação e tamanho dos dados lidos ou escritos, respectivamente, em uma dada porção de memória. No programa principal, há as chamadas para inicialização e finalização desse registro (se solicitado pelo usuário, conforme flags de entrada). É importante mencionar, ainda, a função `acessaMatriz`, implementada em `mat.c`, que possibilita um "aquecimento" da memória, momentos antes de sua utilização em porções específicas, visando otimizar a análise de localidade de referência. Essas funcionalidades permitiram as análises e experimentos de desempenho computacional e eficiência de acesso à memória, cujos resultados serão apresentados nas duas próximas subseções.

5.1 Desempenho Computacional

Foram realizados alguns experimentos com a implementação deste projeto, com o objetivo de avaliar seu desempenho e funcionalidade mediante variação considerável de parâmetros das dimensões das matrizes. As matrizes utilizadas são quadradas e foram geradas aleatoriamente, por meio da função `inicializaMatrizAleatoria`. Os resultados são representados abaixo de forma tabular e gráfica.

Operações	Multiplicação	Soma	Transposição
Dimensões	**Tempo de execução em segundos**		
100	0,030326800	0,010965500	0,010254200
200	0,367354000	0,248162200	0,217754300
300	0,841829100	0,628607400	0,548120600
400	1,939622000	1,220419900	1,051874100
500	3,442717300	1,931133600	1,851826100

Tabela 1 - Relação de tempo de execução por operação e dimensão em matrizes

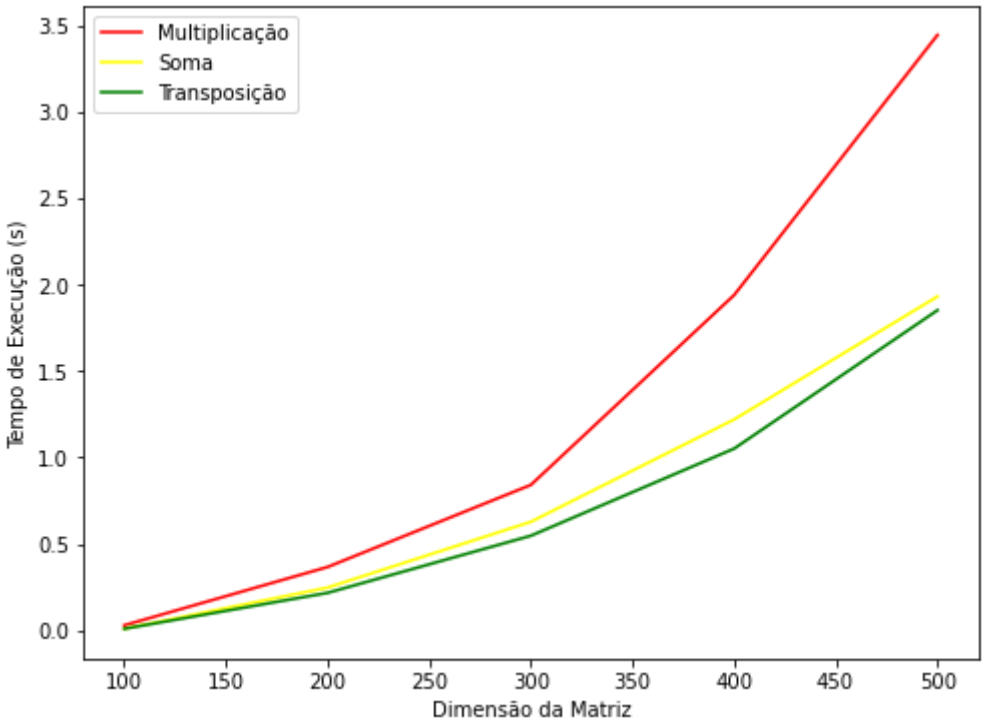

Figura 1 - Gráfico: Dimensão Matriz x Tempo de Execução

Os dados gerados estão sujeitos à margem de erro devido à execução inevitável de outros processos do computador e suas características de processamento, mas dão uma boa noção de como ocorre a variação do tempo de execução conforme se aumentam as dimensões das matrizes.

Nota-se que a multiplicação de matrizes demanda mais tempo para execução, fato que pode ser explicado não somente, mas também pela complexidade de tempo da função `multiplicaMatrizes`, dada por $O(nm^2)$, enquanto `somaMatrizes` e `transpoeMatriz` são $O(nm)$, conforme analisado na subseção 3.1.

5.2 Eficiência de Acesso à Memória

Nesta subseção será analisado como a execução do programa se comporta ao acessar as porções de memória demandadas e o quão eficiente é esse acesso.

5.2.1 Padrão de Acesso

Foram analisados, por meio dos arquivos gerados pela biblioteca `memlog`, como se dá o acesso aos endereços das três matrizes (id 0, 1 e 2) envolvidas na operação de **multiplicação**. As matrizes utilizadas são quadradas e de dimensão 5, geradas pela função `inicializaMatrizAleatoria`. Os resultados são apresentados de forma gráfica a seguir.

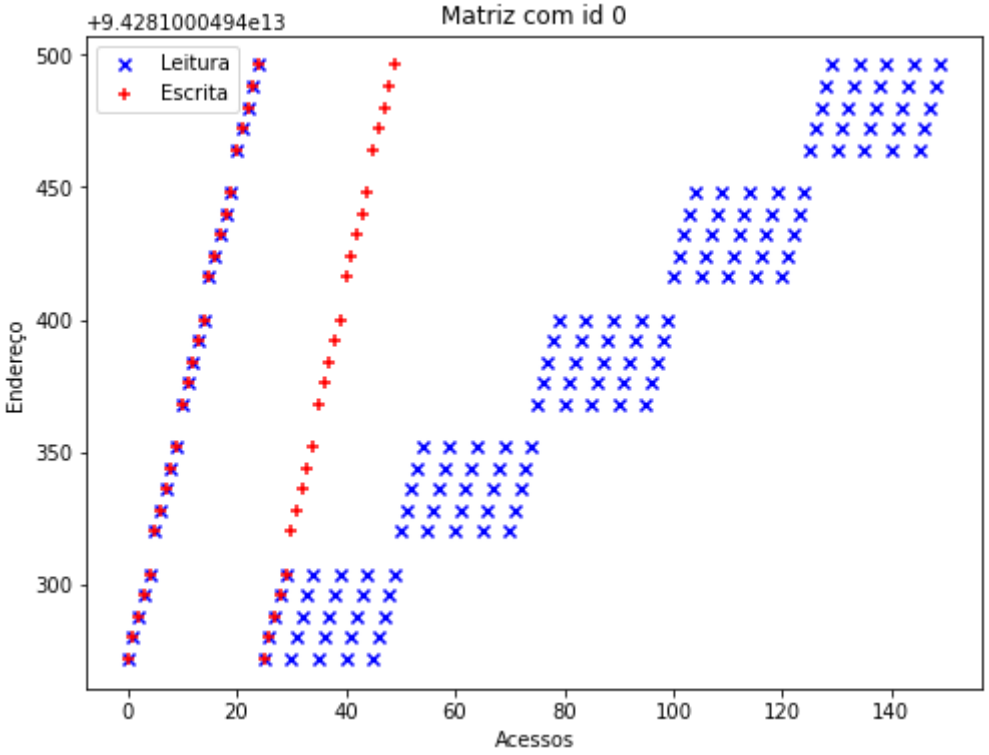

Figura 2 - Gráfico: Acessos x Endereço da Matriz com id 0

Note que as duas primeiras fileiras verticais maiores de acessos de leitura e escrita estão relacionadas à alocação, inicialização e à função `acessaMatriz`. Cada um dos agrupamentos 5X5 de leitura posteriores de leitura dizem respeito a uma linha da matriz, que é acessada 5 vezes para a multiplicação.

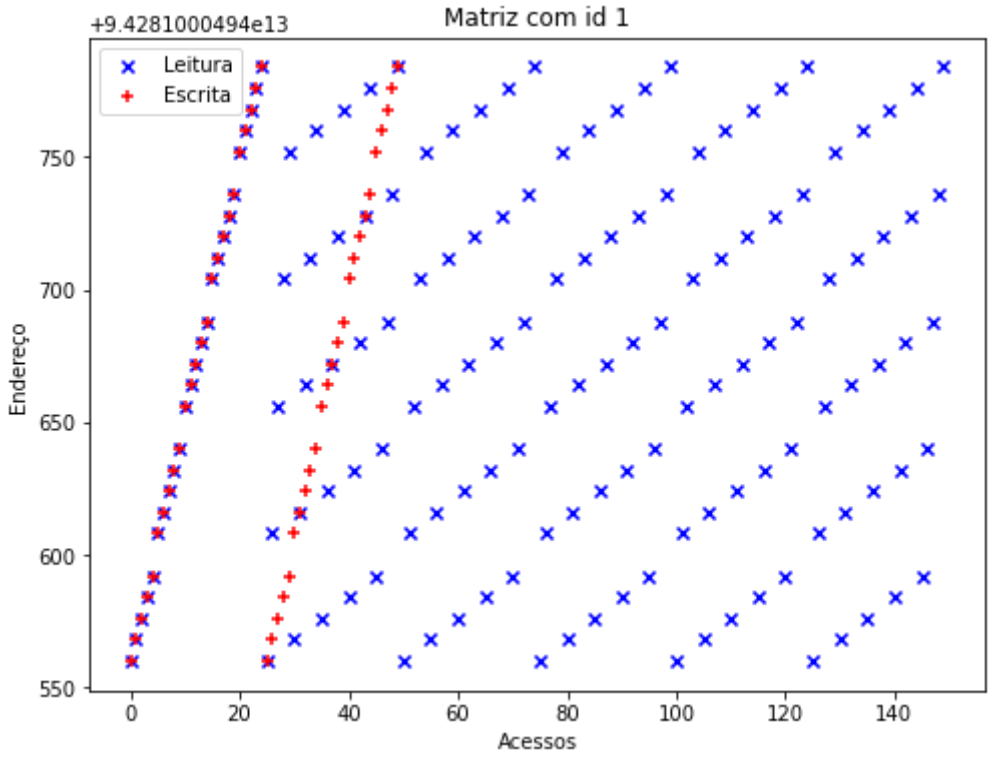

Figura 3 - Gráfico: Acessos x Endereço da Matriz com id 1

Novamente, as duas primeiras fileiras verticais maiores de acessos de leitura e escrita estão relacionadas à alocação, inicialização e à função `acessaMatriz`. Já as fileiras diagonais de leitura que se seguem representam a forma descontínua com que a matriz é acessada, já que ela é alocada por linha: os acessos se dão por colunas e, ao fim, voltam no início, repetindo esse processo por 5 vezes. Esse tipo de acesso representa grande distância de pilha, já que os endereços do começo e início da fileira estão distantes. Esse quesito será analisado na próxima subseção.

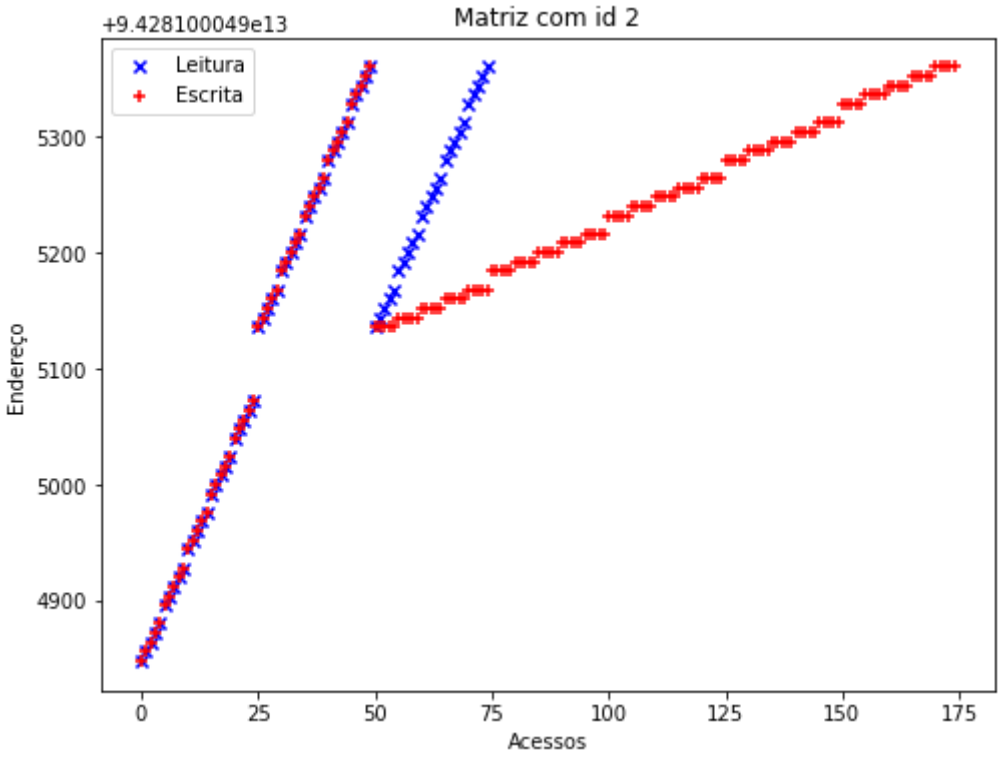

Figura 4 - Gráfico: Acessos x Endereço da Matriz com id 2

Agora, nota-se que são feitas duas inicializações, já que a matriz 2 é inicializada novamente para que suas dimensões se adequem ao resultado da operação de multiplicação e, em seguida, temos os acessos de leitura da função `acessaMatriz`. Posteriormente, são representadas as somas graduais que ocorrem nos elementos da matriz, em que se nota uma excelente distância de pilha, já que os endereços subsequentes estão próximos.

5.2.2 Localidade de Referência e Conjunto de Trabalho

A seguir, será feita a análise de Localidade de Referência e Conjunto de Trabalho utilizando a **Distância de Pilha**. As medições foram feitas sobre a operação de **multiplicação** e no decorrer da fase principal: a **fase 1**, em que é efetivamente chamada a função `multiplicaMatrizes`. As matrizes utilizadas são quadradas, de dimensão 5 e são geradas aleatoriamente pela função `inicializaMatrizAleatoria`. Os resultados serão apresentados com histogramas.

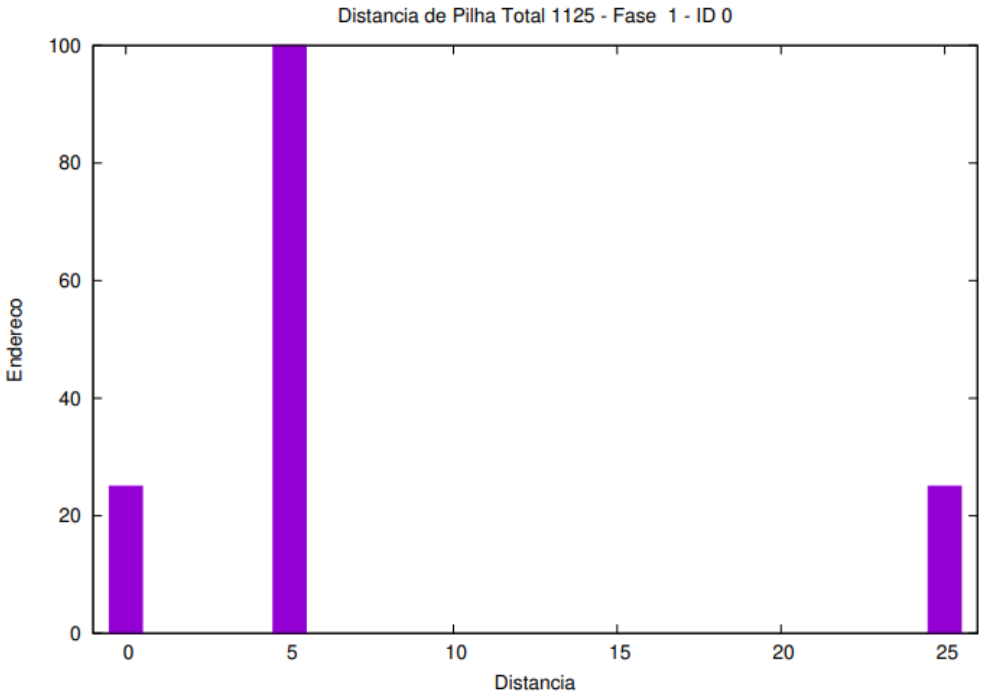

Figura 5 - Histograma: Distância de Pilha - Fase 1 - Matriz id 0

Note que os pouco mais de 20 acessos com distância de pilha 0 dizem respeito aos acessos da função `acessaMatriz`. Já os acessos com dp = 5, se dão ao fazer a multiplicação da matriz, ao repetir o acesso a uma mesma linha. Já os acessos com dp = 25 são do fim da matriz para o início.

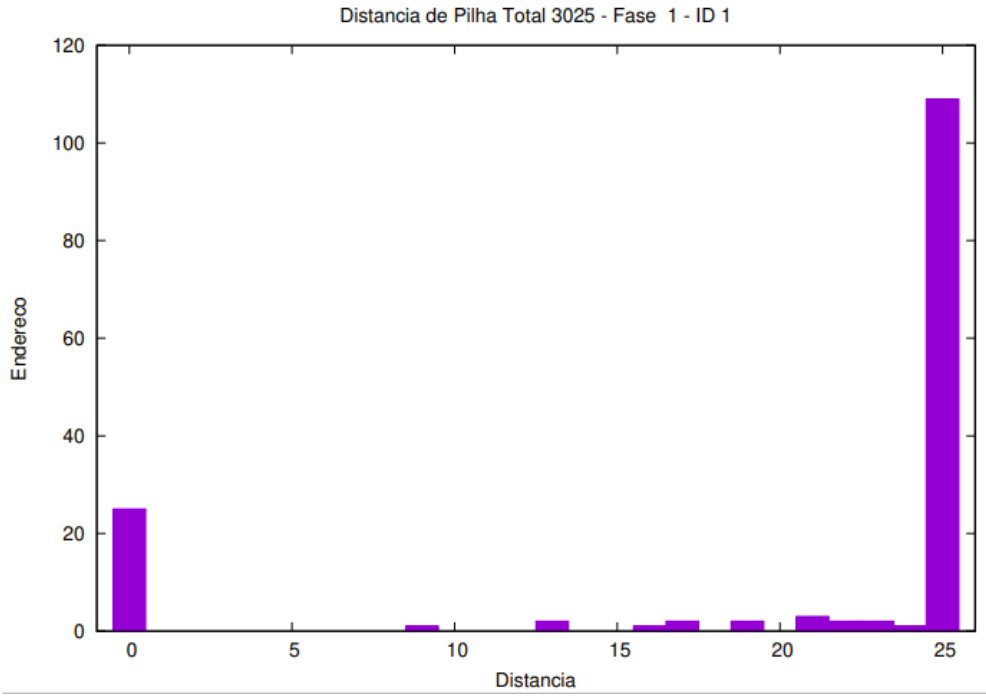

Figura 5 - Histograma: Distância de Pilha - Fase 1 - Matriz id 1

Temos, novamente, os acessos com dp=0 que são da função `acessaMatriz`. Conforme análise na *Figura 3*, notamos que a matriz com id 1 possui grande número de acessos descontínuos, o que caracteriza o grande número de acessos com dp=25. Já as demais ocorrências têm relação com trocas de colunas durante o processo da multiplicação.

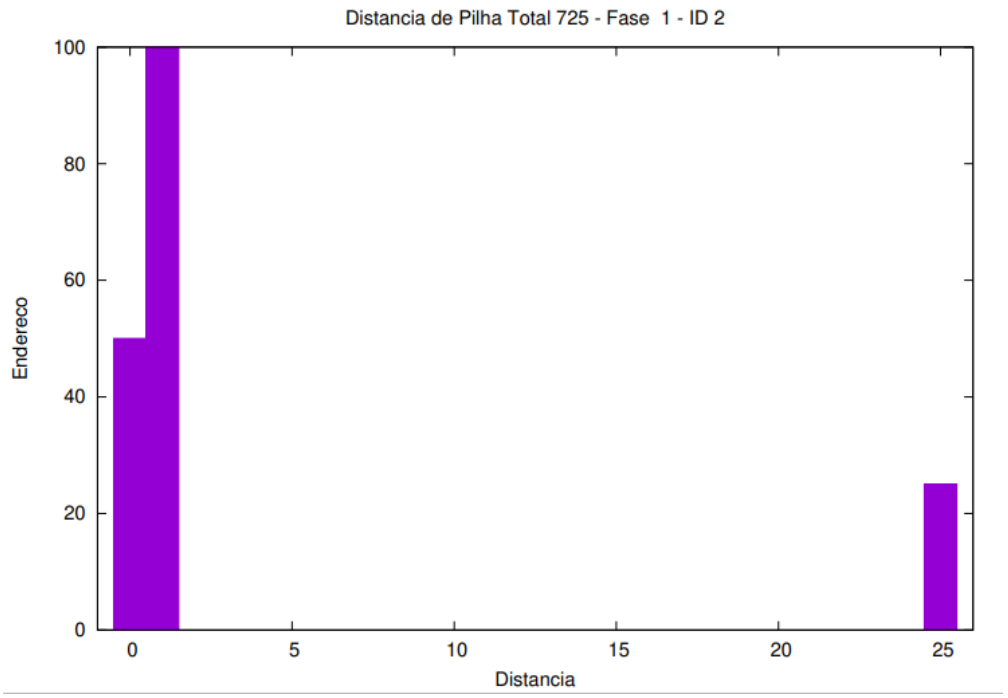

Figura 5 - Histograma: Distância de Pilha - Fase 1 - Matriz id 2

Conforme se observa na *Figura 4*, a maioria dos acessos na matriz 2 se dão em endereços subsequentes, com pequena distância de pilha e esse comportamento pode ser observado no histograma. Já os acessos com dp=25 são da transição do fim para o início da matriz.

6. Conclusão

Este trabalho teve como objetivo a implementação de um programa que possibilitasse operações com matrizes alocadas dinamicamente, de modo interativo e eficiente. Após a implementação do projeto como um todo, a minha percepção sobre o processo de aprendizagem foi ampliada. Neste trabalho, não apenas a funcionalidade do código é levada em consideração, mas também aspectos que seriam, possivelmente, cobrados na "vida real", digamos: robustez, complexidade de funções, modularização e demanda por memória. O grande desafio foi, além da parte lógica e de ter de relembrar e pesquisar aspectos da alocação dinâmica em linguagem C, aprender e executar processos paralelos e inerentes ao desenvolvimento de código, como os citados. Nunca havia tido contato com a biblioteca `memlog`, que foi utilizada no

registro de acesso, ou com `gnuplot`, que utilizei para plotar os histogramas de distância de pilha, mas pude vislumbrar sua utilidade e como, na prática, os recursos viabilizados, mesmo de forma tabular ou gráfica, enriquecem o conhecimento do desenvolvedor acerca de seu próprio código e possibilitam sua otimização e refatoração, processos que nunca têm fim.

7. Bibliografia

CORMEN , T., LEISERSON, C, RIVEST R., STEIN, C. **Introduction to Algorithms**, Third Edition, MIT Press, 2009.

CHAIMOWICZ, L. and PRATES, R. **Slides virtuais da disciplina de estruturas de dados**, 2020.

Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Alocação Dinâmica. **Unicamp**, 2005.

Disponível em: <https://www.ic.unicamp.br/~norton/disciplinas/mc1022s2005/03_11.html>. Acesso em: 05 de novembro de 2021.

FEOFILOFF, Paulo. Alocação Dinâmica de Memória. **IME-USP**, 2018.

Disponível em: <https://www.ime.usp.br/~pf/algoritmos/aulas/aloca.html>. Acesso em: 05 de novembro de 2021.

Curso de C. **PUC-SP**, 1999.

Disponível em: <https://www.pucsp.br/~so-comp/cursoc/aulas/c520.html#c522.html>.

Acesso em: 05 de novembro de 2021.

APÊNDICE A
Instruções de Compilação e Execução

Com a utilização do Makefile, a compilação e linkagem do programa se dão de maneira muito simples:

- Colocando o seu terminal no diretório raiz do projeto, basta que você digite `make` e o programa será compilado e linkado.
- Para executar o programa utilizando matrizes aleatórias, digite no terminal **(o terminal deve estar no diretório raiz do projeto)**, após compilação:

```
bin/matop -<operacao>[1]
          -<demais flags desejadas>[2]
          -x <dimensao x>
          -y <dimensao y>
```

- **Alternativamente**, é possível utilizar a leitura de matrizes em arquivos, ao colocar, na linha de comando:

```
bin/matop -<operacao>[3]
          -<demais flags desejadas>
          -1 <endereço e nome do arquivo da primeira matriz>
          -2 <endereço e nome do arquivo da segunda matriz (apenas se operação não
for                transposicao)>
          -o <endereço e nome do arquivo onde será armazenado o resultado>
```

Os arquivos que armazenam as matrizes devem ser extensão `.txt` e conter as **dimensões da matriz na primeira linha**, e, claro, depois, a matriz.

Por fim, é possível excluir, caso se queira, os arquivos objects (`.o`) gerados com a compilação e, ainda, o arquivo executável, entrando com o comando `make clean` no terminal, quando ele estiver no diretório raiz do projeto.

---

[1] A quebra de linha não é necessária no terminal. Foi feita, nesse caso, apenas para melhor visualização.
[2] Por exemplo: -l, para realizar registro de acesso.
[3] A quebra de linha não é necessária no terminal. Foi feita, nesse caso, apenas para melhor visualização.