Trabalho Prático 0: Operações com matrizes alocadas dinamicamente

Pedro de Oliveira Guedes
2021040008

DCC/UFMG

Introdução

A documentação aqui presente se refere a um programa produzido na linguagem de programação C, com o objetivo de implementar um tipo abstrato de dados correspondente a uma matriz dinamicamente alocada, bem como operações matemáticas pertinentes a elas.

As matrizes estarão descritas em arquivos ".txt", que deverão ser lidos e armazenados dentro da estrutura de dados referente à matriz em questão. Além da leitura deste tipo de arquivo, o programa deve ser capaz também de realizar as seguintes operações matemáticas sobre as matrizes, armazenando o resultado das mesmas em outro arquivo de texto com a mesma formatação do mostrado anteriormente:

1. Soma
2. Multiplicação
3. Transposição

Mais detalhes sobre a implementação de cada um destes métodos serão dados no decorrer deste documento.

Implementação

Nesta seção, serão dadas informações sobre a organização do programa em pastas, bem como o que cada uma abriga, detalhes sobre a estrutura de dados implementada para a matriz, informações sobre as funções, métodos e procedimentos utilizados, etc.

Configurações do sistema de testes

O programa foi implementado na linguagem C, compilado pelo GCC, compilador da GNU Compiler Collection. Além disso, foi utilizado o sistema operacional Ubuntu 9.3.0-17 ubuntu1~20.04 em um Windows 10, através do Windows Subsystem for Linux (WSL2), com um processador AMD Ryzen 3 3200G (3.60GHz, 4 CPUs) e 13 GB RAM.

Formato de entrada e saída

A estrutura de dados de matriz implementada deve ser lida a partir de um arquivo de texto (extensão .txt), que possui a estrutura a seguir:

```
DimX DimY
valor[0][0] valor[0][1] valor[0][2] ... valor[0][DimY]
.
.
.
valor[DimX][0] valor[DimX][1] valor[DimX][2] ... valor[DimX][DimY]
```

Ou seja, a primeira linha deve conter apenas dois valores inteiros, representando as dimensões da matriz, e as linhas subsequentes devem ser os valores (em ponto flutuante) referentes a cada uma das posições da matriz.

Tanto a matriz de entrada quanto a de saída seguem este mesmo padrão de construção do arquivo .txt.

Organização de pastas

A pasta raiz do projeto se chama 00-matrizes_dinamicas e abriga, além das subpastas diretamente relacionadas ao programa, a subpasta “analisamem”, que não será discutida neste documento, já que foi criada pelos professores da disciplina.

Analises

Esta pasta contém todos os resultados computacionais gerados para as análises do programa (presentes no makefile), está dividido em subpastas, uma para cada operação que o programa realiza (soma, multiplicação e transposição), e, dentro das pastas de cada operação, estão outras subpastas para resultados específicos (gprof, logs, mem, perf). As pastas “mem” e “perf”, além de conterem os resultados na pasta “resultados”, também possuem as matrizes de entrada que sofreram as operações, juntamente com as matrizes produzidas pelas operações, dentro da pasta “matrizes”.

bin

Esta pasta contém todos os binários gerados pelo makefile, ou compilação manual. O makefile está configurado para gerar o executável com o nome “matop”.

include

Esta pasta contém todos os arquivos de cabeçalho utilizados para a implementação das funções nos arquivos de código fonte. Nela estão presentes:

mat.h

Definição da estrutura da matriz, que possui os seguintes atributos:

```
typedef struct mat{
    double **m; // Matriz dinâmica
    char *matrixPath; // Caminho onde o .txt da matriz está armazenado
    int tamX, tamY; // Dimensões X e Y da matriz
    int id; // Identificador da matriz para resultados computacionais
} mat_tipo; // Nome da estrutura de matriz
```

Além disso, são também declarados os métodos utilizados, que serão listados e melhor explicados no respectivo arquivo de implementação.

memlog.h

Define o TAD “memlog_tipo”, que é utilizado para fazer registros de acesso à memória. Define também métodos para registro de acesso à memória, que serão melhor explicados no respectivo arquivo de implementação.
Define também as macros “LEMEMLOG”, para facilitar o registro de acesso à memória para leitura, e “ESCREVEMEMLOG”, para facilitar o registro de acesso à memória para escrita.

msgassert.h

Define as macros:

- *avisoAssert:* recebe uma expressão booleana como primeiro parâmetro e, caso ela seja falsa, imprime na tela um aviso contendo a string enviada no segundo parâmetro. Caso a expressão seja verdadeira, a macro não faz nada.
- *erroAssert:* recebe uma expressão booleana como primeiro parâmetro e, caso ela seja falsa, imprime na tela um erro contendo a string enviada no segundo parâmetro e finaliza o programa. Caso a expressão seja verdadeira, a macro não faz nada.

obj

Esta pasta abriga os arquivos de objeto do programa principal gerados durante a compilação dos arquivos de código fonte (presentes na pasta "src"), de acordo com o Compilador utilizado e o Sistema Operacional instalado na máquina de testes.

src

Esta pasta contém todos os arquivos de código fonte produzidos para o funcionamento do programa. Eles serão listados a seguir, bem como as funções que cada um abriga:

mat.c

Implementação das funções declaradas no arquivo "mat.h". As funções são:

- *void dimensoesMatriz(mat_tipo *mat)*: Esta função recebe uma matriz do tipo "mat_tipo" por referência no parâmetro "mat", acessa o atributo "matrixPath" da mesma, realiza a abertura do arquivo informado neste atributo, lê as dimensões presentes na primeira linha, verifica estas dimensões quanto à validade e define os atributo "tamX" e "tamY" de acordo com eles.

- *void criaMatrizInput(mat_tipo *mat, char *matrixPath, int id)*: Esta função recebe uma matriz do tipo "mat_tipo" por referência no parâmetro "mat", a localização de um arquivo de texto contendo os valores e dimensões desta matriz no parâmetro "matrixPath" e um identificador para as análises de desempenho no parâmetro "id". Ela faz a validação dos parâmetros recebidos, define o caminho do .txt referente à matriz, realiza a chamada da função "dimensoesMatriz", realiza a alocação de

memória com base nas dimensões atribuídas na função anterior e define o identificador da matriz.

- *void criaMatrizOutput(mat_tipo *mat, char *matrixPath, int tamX, int tamY, int id)*: Esta função recebe uma matriz do tipo "mat_tipo" por referência no parâmetro "mat", a localização de um arquivo de texto contendo os valores e dimensões desta matriz no parâmetro "matrixPath", quantidade de linhas e colunas por "tamX" e "tamY" respectivamente e um identificador para as análises de desempenho no parâmetro "id". Ela faz a validação dos parâmetros recebidos, define o caminho do .txt referente à matriz, atribui as dimensões recebidas por "tamX" e "tamY" à matriz, realiza a alocação de memória com base nas dimensões atribuídas na função anterior e define o identificador da matriz.

  Obs: Foi criada uma função diferente para criação de matrizes de input e output porque os arquivos de texto são diferentes. Enquanto as matrizes de entrada possuem as informações de dimensão e todos os valores da matriz, as matrizes de saída não possuem nenhum dado e, em alguns casos, podem nem existir. Portanto, para a criação da matriz de output, é necessário que as dimensões sejam informadas por parâmetros

- *void inicializaMatrizNula(mat_tipo *mat)*: Esta função recebe uma matriz do tipo "mat_tipo" por referência no parâmetro "mat". Percorre por todos os elementos da matriz dinamicamente alocada e atribui a todas as posições, o valor 0.

- *void leMatrizDoTxt(mat_tipo *mat)*: Esta função recebe uma matriz do tipo "mat_tipo" por referência no parâmetro "mat". Abre o .txt, cujo caminho está armazenado no atributo "matrixPath", lê todos os elementos, fazendo verificação da consistência do arquivo (se é menor ou maior que as dimensões informadas) e atribui os valores lidos às suas respectivas posições na matriz.

- *double acessaMatriz(mat_tipo *mat)*: Esta função recebe uma matriz do tipo "mat_tipo" por referência no parâmetro "mat". Acessa todos os elementos da matriz, realizando a soma deles e registrando o acesso à memória para leitura. Ao fim, retorna o valor somado.

- *void copiaMatrizes(mat_tipo *src, mat_tipo *dst)*: Esta função recebe duas matrizes do tipo "mat_tipo" por referência através dos parâmetros "src" e "dst". Percorre por todos os elementos de "src" e os atribui nas mesmas posições de "dst", registrando todos os acessos à memória para leitura e escrita nas matrizes.

- *void matrizParaTxt(mat_tipo *src)*: Esta função recebe uma matriz do tipo "mat_tipo" por referência no parâmetro "src". Percorre por todos os elementos de "src", registrando os respectivos acessos para leitura, e os imprime no arquivo .txt presente no atributo "matrixPath" de "src".

- *void somaMatrizes(mat_tipo *a, mat_tipo *b, mat_tipo *c)*: Esta função recebe três matrizes do tipo "mat_tipo" por referência através dos parâmetros "a", "b" e "c". Percorre por todos os elementos de "a" e "b" simultaneamente, realizando o registro de acesso à memória para leitura deles, realiza a soma dos pares acessados, e registra a soma na posição correspondente da matriz "c", realizando, também, o registro de acesso para escrita.

- *void multiplicaMatrizes(mat_tipo *a, mat_tipo *b, mat_tipo *c)*: Esta função recebe três matrizes do tipo "mat_tipo" por referência através dos parâmetros "a", "b" e "c". Possui três laços indentados um sobre o outro, o primeiro percorrendo as linhas da matriz "a" (chamaremos de *i*), o segundo percorrendo as colunas da matriz "b" (chamaremos de *j*) e o terceiro percorrendo as colunas da matriz "a" e as linhas da matriz "b" (chamaremos de *k*) que, para que a multiplicação seja possível, devem possuir o mesmo tamanho. É registrado o acesso para leitura dos elementos de "a" e "b" nas posições *a[i][k]* e *b[k][j]*, os elementos são multiplicados e armazenados em *c[i][j]*, que também registra o acesso à memória para escrita.

- *void transpoeMatriz(mat_tipo *a, mat_tipo *b)*: Esta função recebe uma matriz do tipo "mat_tipo" por referência no parâmetro "a". Percorre todas as linhas e colunas desta matriz e insere os elementos de "a" em posições invertidas de "b", ou seja, os elementos que estavam na linha *i* e coluna *j* de "a", passam a estar na linha *j* e coluna *i* de "b". São registrados ambos acessos à memória para leitura e escrita.

- *void destroiMatriz(mat_tipo *a)*: Esta função recebe uma matriz do tipo “mat_tipo” por referência no parâmetro “a”. Percorre todos os elementos da matriz, desalocando-os e atribuindo seus respectivos endereços de memória a um ponteiro nulo (*NULL*).

matop.c

Implementação do fluxo de execução principal do programa. As funções são:

- *void uso()*: Não recebe nenhum parâmetro. Imprime na tela as instruções para utilizar o programa, as flags e os parâmetros que cada uma recebe.

- *void parse_args(int argc, char **argv)*: Recebe os parâmetros “argc”, relativo à quantidade de argumentos (flags) informados pelo usuário na execução do programa, e “argv”, relativo aos valores de cada um dos argumentos informados. Itera por cada um dos argumentos fornecidos e realiza operações pertinentes, como definir a operação a ser realizada, ou atribuir o arquivo de texto à sua respectiva matriz.

- *int main(int argc, char **argv)*: Recebe os parâmetros “argc”, relativo à quantidade de argumentos (flags) informados pelo usuário na execução do programa, e “argv”, relativo aos valores de cada um dos argumentos informados. Faz a chamada da função “parse_args” para definir a operação e outros parâmetros importantes para a execução do programa. Após isso, verifica qual foi a operação escolhida pelo usuário, e executa uma série de procedimentos relativos à ela.

memlog.c

Implementação das funções declaradas no arquivo “memlog.h”. Aqui serão listadas somente as funções mais importantes e relevantes para o funcionamento do programa:

- *int leMemLog(long int pos, long int tam, int id)*: Recebe os parâmetros “pos”, relativo à posição acessada na memória do sistema, “tam”, relativo ao tamanho do valor que o endereço acessado abriga, e “id”, relativo ao identificador da estrutura que abriga a posição da memória acessada. Através dos parâmetros recebidos, faz o registro do acesso à memória para leitura.

- *int escreveMemLog(long int pos, long int tam, int id)*: Recebe os parâmetros “pos”, relativo à posição acessada na memória do sistema, “tam”, relativo ao tamanho do

valor que o endereço acessado abriga, e “id”, relativo ao identificador da estrutura que abriga a posição da memória acessada. Através dos parâmetros recebidos, faz o registro do acesso à memória para escrita.

- *int defineFaseMemLog(int f)*: Recebe o parâmetro “f”, relativo à fase de registros de acesso à memória. Define a fase de registros de acesso à memória para que as funções “leMemLog” e “escreveMemLog” sejam executadas de forma organizada.

Análise de complexidade

A análise de complexidade contemplará as funções implementadas nos arquivos “mat.c” e “matop.c”. Para todas as funções, serão consideradas operações relevantes:

- Complexidade de tempo: leitura e atribuição de valores.
- Complexidade de espaço: alocação de memória dinâmica, recebimento de matrizes e chamada de funções.

A presente análise de complexidade leva em consideração, primeiramente a notação “grande θ", mas mostra também a complexidade grande O, considerando matrizes de dimensão n.

*dimensoesMatriz(mat_tipo *mat)*:

Esta função realiza a abertura de um arquivo de texto, lê as dimensões presentes na primeira linha, realizando duas atribuições, uma para cada valor lido referente às dimensões

Tempo

São realizadas duas atribuições, portanto: $\theta(2)$. Porém, como é uma análise assintótica (tendendo ao infinito), podemos ignorar constantes. Dessa forma, dizendo que a complexidade de tempo desta função é constante:

$$\theta(1)$$
$$O(1)$$

Espaço

Esta função não realiza alocação de memória, nem realiza o chamado de outras funções, portanto a complexidade de espaço da mesma é:

$$\theta(1)$$
$$O(1)$$

*criaMatrizInput(mat_tipo *mat, char *matrixPath, int id)*:

Esta função faz a atribuição de todos os parâmetros recebidos aos respectivos atributos da estrutura “mat”. Realiza a chamada da função “dimensoesMatriz” e a alocação de memória com base nas dimensões atribuídas na função anterior. Chamaremos as constantes de quantidade de linhas e colunas de, respectivamente, “M” e “N”.

Tempo

As atribuições dos parâmetros aos atributos é sempre constante e não varia de acordo com o tamanho da entrada, nos dando, inicialmente, complexidade $\theta(1)$. É feita também a chamada da função "dimensões Matriz", que já analisamos e descobrimos que também tem complexidade de tempo $\theta(1)$. Por fim, durante a alocação de memória, percebemos que inicialmente é feita **uma única** alocação para as "M" linhas da matriz, em seguida são alocados "N" endereços de memória para cada uma das "M" linhas, nos dando uma complexidade de $\theta(1) + \theta(M)$, que resulta em $\theta(M)$. Realizando a soma de todas complexidades de tempo anteriormente listadas, por propriedades de complexidade assintótica, temos que a complexidade de tempo total da função será:

$$\theta(M)$$

$$O(n)$$

Espaço

No começo desta função, é feita a chamada da função "dimensoesMatriz", que já estudamos e possui complexidade de espaço $\theta(1)$. Logo após, são alocados "M" endereços de memória para as linhas da matriz e, para cada um destes endereços alocados, dentro deles são alocados mais outros "N", relativos à quantidade de colunas. Dessa forma, percebemos que a alocação de memória, no total, terá complexidade de espaço $\theta(M * N)$. Por propriedades de complexidade assintótica, afirmamos então que a complexidade final da função é:

$$\theta(M * N)$$

$$O(n^2)$$

*criaMatrizOutput(mat_tipo *mat, char *matrixPath, int tamX, int tamY, int id)*:

Esta função faz a atribuição de todos os parâmetros recebidos aos respectivos atributos da estrutura "mat". Realiza a alocação de memória com base nas dimensões atribuídas por "tamX" e "tamY". Chamaremos as constantes de quantidade de linhas e colunas de, respectivamente, "M" e "N".

Tempo

As atribuições dos parâmetros recebidos na função para os atributos da estrutura são constantes com a variação da entrada, ou seja $\theta(1)$. É feita **uma única** alocação para as "M" linhas da matriz, em seguida são alocados "N" endereços de memória para cada uma das "M" linhas, nos dando uma complexidade de $\theta(1) + \theta(M)$, que resulta em $\theta(M)$. No total, teremos a complexidade de tempo desta função como:

$$\theta(M)$$

$$O(n)$$

Espaço

No começo desta função, são alocados "M" endereços de memória para as linhas da matriz e, para cada um destes endereços alocados, dentro deles são alocados mais outros "N", relativos à quantidade de colunas. Dessa forma, percebemos que a alocação de memória, no total, terá complexidade de espaço $\theta(M * N)$. Portanto, a complexidade de espaço será:

$$\theta(M * N)$$

$$O(n^2)$$

*inicializaMatrizNula(mat_tipo *mat)*:

Esta função percorre por todos os elementos da matriz dinamicamente alocada e atribui a todas as posições, o valor 0. Chamaremos as constantes de quantidade de linhas e colunas de, respectivamente, "M" e "N".

Tempo

A matriz é percorrida por todas as suas "M" linhas e "N" colunas, realizando uma única atribuição para cada um dos elementos acessados. Portanto:

$$\theta(M * N)$$

$$O(n^2)$$

Espaço

Esta função não realiza alocação dinâmica de memória e nem a chamada de outras funções, porém, recebe uma matriz por referência. Portanto:

$$\theta(M * N)$$

$$O(n^2)$$

*leMatrizDoTxt(mat_tipo *mat)*:

Esta função realiza a chamada da função "inicializaMatrizNula", percorre todas as linhas e colunas da matriz inicializada nula, e atribui a cada uma das posições, os valores lidos no arquivo de texto que contém as informações da matriz. Chamaremos as constantes de quantidade de linhas e colunas de, respectivamente, "M" e "N".

Tempo

É feita a chamada da função "inicializaMatrizNula", que tem complexidade de tempo conhecida como $\theta(M * N)$. Logo após, percorre pelas "M" linhas e "N" colunas, realizando uma atribuição para cada um destes acessos. Ou seja, mais uma complexidade de $\theta(M * N)$. Por regras da complexidade assintótica, tempos então que a complexidade de tempo total desta função é

$$\theta(M * N)$$

$$O(n^2)$$

Espaço

Não é feita a alocação dinâmica de memória nesta função. É feita porém a chamada da função "inicializaMatrizNula", que tem complexidade de espaço já conhecida e igual a $\theta(1)$, além disso, recebe uma matriz por referência. Portanto:

$$\theta(M * N)$$

$$O(n^2)$$

*acessaMatriz(mat_tipo *mat)*:

Esta função acessa todos os elementos da matriz, realizando a soma deles e registrando o acesso à memória para leitura. Chamaremos as constantes de quantidade de linhas e colunas de, respectivamente, “M” e “N”.

Tempo

Esta função percorre pelas “M” linhas e “N” colunas da matriz realizando duas atribuições para cada elemento acessado. Dessa forma, teríamos $\theta(2 * (M * N))$, mas, como em notações assintóticas constantes perdem o sentido, definimos a complexidade de tempo desta função como:

$$\theta(M * N)$$

$$O(n^2)$$

Espaço

Não é feita alocação dinâmica nesta função, nem chamada de outras funções, porém, é recebida uma matriz por referência. Portanto:

$$\theta(M * N)$$

$$O(n^2)$$

*copiaMatrizes(mat_tipo *src, mat_tipo *dst)*:

Esta função percorre, simultaneamente, todos os elementos de “src” e os atribui nas mesmas posições de “dst”, registrando todos os acessos à memória para leitura e escrita nas matrizes. Chamaremos as constantes de quantidade de linhas e colunas de, respectivamente, “M” e “N”.

Tempo

Percorre as “M” linhas e “N” colunas realizando uma única atribuição por posição acessada. Dessa forma, temos a complexidade de tempo como:

$$\theta(M * N)$$

$$O(n^2)$$

Espaço

Não é feita alocação dinâmica nesta função, nem chamada de outras funções, porém, é recebida uma matriz por referência. Portanto:

$$\theta(M * N)$$

$$O(n^2)$$

*matrizParaTxt(mat_tipo *src)*:

Esta função percorre por todos os elementos de "src", registrando os respectivos acessos para leitura, e os imprimindo no respectivo arquivo .txt. Chamaremos as constantes de quantidade de linhas e colunas de, respectivamente, "M" e "N".

Tempo

Percorre as "M" linhas e "N" colunas realizando uma única leitura por posição acessada. Dessa forma, temos a complexidade de tempo como:

$$\theta(M * N)$$

$$O(n^2)$$

Espaço

Não é feita alocação dinâmica nesta função, nem chamada de outras funções, porém, é recebida uma matriz por referência. Portanto:

$$\theta(M * N)$$

$$O(n^2)$$

*somaMatrizes(mat_tipo *a, mat_tipo *b, mat_tipo *c)*:

Esta função percorre por todos os elementos de "a" e "b" simultaneamente, realizando o registro de acesso à memória para leitura deles, realiza a soma dos pares acessados, e registra a soma na posição correspondente da matriz "c", realizando, também, o registro de acesso para escrita. Como para que a operação de soma de matrizes dê certo precisamos que todas as matrizes envolvidas, "a", "b" e "c" tenham exatamente as mesmas dimensões, chamaremos as constantes de quantidade de linhas e colunas de, respectivamente, "M" e "N".

Tempo

Percorre as “M” linhas e “N” colunas de “a”, “b” e “c” simultaneamente, realizando duas leituras e uma atribuição. Além disso, ela também realiza a chamada da função “criaMatrizOutput” (complexidade de tempo $\theta(M)$) e “inicializaMatrizNula” (complexidade de tempo $\theta(M * N)$). Dessa forma, por propriedades das notações assintóticas, temos a complexidade de tempo como:

$$\theta(M * N)$$

$$O(n^2)$$

Espaço

Não é feita a chamada de outras funções, porém, são recebidas três matrizes por referência, o que nos resultaria em $\theta(3 * (M * N))$. Além disso, é feita a chamada das funções “criaMatrizOutput” (complexidade de espaço $\theta(M * N)$) e “inicializaMatrizNula” (complexidade de espaço $\theta(M * N)$). Por propriedades de notação assintótica, definimos a complexidade de espaço desta função como:

$$\theta(M * N)$$

$$O(n^2)$$

*multiplicaMatrizes(mat_tipo *a, mat_tipo *b, mat_tipo *c)*:

Esta função possui três laços indentados um sobre o outro, o primeiro percorrendo as linhas da matriz “a”, o segundo percorrendo as colunas da matriz “b” e o terceiro percorrendo as colunas da matriz “a” e as linhas da matriz “b”, que, para que a multiplicação seja possível, devem possuir o mesmo tamanho. É registrado o acesso para leitura dos elementos de “a” e “b”, eles são multiplicados e armazenados em “c”, que também registra o acesso à memória para escrita. Como para que a operação de multiplicação de matrizes dê certo precisamos que a quantidade de colunas de “a” e a quantidade de linhas de “b” tenham exatamente as mesmas dimensões, chamaremos as constantes de quantidade de linhas de “a” por M, a quantidade de colunas de “a” e linhas de “b” por N, e a quantidade de colunas de B por O. Lembre-se que as dimensões de “c” são justamente MxO.

Tempo

Os três laços são feitos, em ordem, pelas linhas de “a”, as colunas de “b” e as colunas de “a” juntamente com linhas de “b”. Além disso, ela também realiza a chamada da função “criaMatrizOutput” (complexidade de tempo $\theta(M)$) e “inicializaMatrizNula” (complexidade de tempo $\theta(M * N)$). Dessa forma, por propriedades das notações assintóticas, a complexidade de tempo desta função é:

$$\theta(M * O * N)$$

$$O(n^3)$$

Espaço

Não é feita a chamada de outras funções, porém, são recebidas três matrizes por referência, o que nos resultaria em $\theta(3 * (M * N))$. Além disso, é feita a chamada das funções “criaMatrizOutput” (complexidade de espaço $\theta(M * N)$) e “inicializaMatrizNula” (complexidade de espaço $\theta(M * N)$). Por propriedades de notação assintótica, definimos a complexidade de espaço desta função como:

$$\theta(M * N)$$

$$O(n^2)$$

*transpoeMatriz(mat_tipo *a, mat_tipo *b)*:

Esta função percorre todas as linhas e colunas da matriz “a” e atribuindo os elementos desta matriz a posições invertidas de “b”. São registrados ambos acessos à memória para escrita. Como para que a operação de transposição de matrizes dê certo precisamos que as matrizes “a” e “b” tenham as mesmas dimensões, porém invertidas em linha e coluna, chamaremos as constantes de quantidade de linhas e colunas de “a”, respectivamente, de “M” e “N”.

Tempo

A função percorre todos os elementos de “a” e realiza uma atribuição para “b” por elemento acessado. Além disso, ela também realiza a chamada da função “criaMatrizOutput” (complexidade de tempo $\theta(M)$) e “inicializaMatrizNula” (complexidade de tempo

$\theta(M * N)$). Dessa forma, por propriedades das notações assintóticas, podemos dizer que a complexidade de tempo é:

$$\theta(M * N)$$

$$O(n^2)$$

Espaço

Esta função recebe duas outras matrizes "a" e "b", o que nos resulta em $\theta((M * N) + (N * M))$. Além disso, é feita a chamada das funções "criaMatrizOutput" (complexidade de espaço $\theta(M * N)$) e "inicializaMatrizNula" (complexidade de espaço $\theta(M * N)$). Por propriedades de notação assintótica, definimos a complexidade de espaço desta função como:

$$\theta(M * N)$$

$$O(n^2)$$

*destroiMatriz(mat_tipo *a)*:

Esta função percorre todos os elementos da matriz, desalocando-os e atribuindo seus respectivos endereços de memória a um ponteiro nulo (*NULL*). Chamaremos as constantes de quantidade de linhas e colunas de, respectivamente, "M" e "N".

Tempo

São percorridas todas as linhas da matriz "a" realizando a desalocação, as colunas não são percorridas. Portanto:

$$\theta(M)$$

$$O(n)$$

Espaço

Inicialmente, todas as posições da matriz estão alocadas, portanto:

$$\theta(M * N)$$

$$O(n^2)$$

main(int argc, char **argv)

Para o programa principal, não há uma complexidade de execução bem definida, já que a operação executada depende da escolha do usuário. Sendo assim, as análises serão feitas com base no pior caso, comparando as complexidades das funções já analisadas anteriormente e escolhendo a maior delas.

Tempo

Dentre as operações analisadas, a que tem maior complexidade de tempo é a de multiplicação. Dessa forma, a complexidade de tempo do programa como um todo será da ordem de:

$$O(n^3)$$

Espaço

A ordem da complexidade de espaço não varia muito entre as funções chamadas para as operações. Portanto, a ordem de complexidade de espaço do programa como um todo será de:

$$O(n^2)$$

Estratégias de Robustez

A robustez em um programa é de crucial importância para o bom funcionamento do mesmo. Nesse sentido, uma série de verificações de entrada do usuário se fazem necessárias, para que não haja execuções errôneas ou estouros grosseiros de erro no programa. Para isso, utilizei a biblioteca *msgassert.h*, também presente nos arquivos deste trabalho. As funções que compõem a biblioteca estão descritas no tópico "Implementação", subtópico "Organização de pastas", item "include".

As estratégias serão listadas e explicadas por arquivo e função, mostrando um exemplo de como foi inserida no código.

matop.c

Neste arquivo, duas funções utilizam estratégias de robustez para filtrar erros do usuário já nos estágios iniciais do processo. São elas:

void parse_args(int argc, char **argv)

Esta função utiliza o método *avisoAssert*, que informa ao usuário quando um parâmetro já havia sido informado e está sendo sobrescrito, e o método *erroAssert*, que realiza a terminação do programa caso algum parâmetro crucial (operação ou arquivo para logs de execução) não tenha sido informado.

Exemplos *avisoAssert*:

```
avisoAssert(opescolhida==-1, "Mais de uma operacao escolhida");

avisoAssert(matrixPathRes[0]==0, "Ja havia sido informado um
arquivo para a matriz de output");
```

Exemplos *erroAssert*:

```
erroAssert(opescolhida > 0, "matop - necessario escolher
operacao");
erroAssert( strlen(lognome) > 0, "matop - nome de arquivo de
registro de acesso tem que ser definido" );
```

int main(int argc, char **argv)

Esta função utiliza apenas o método *erroAssert*, que é invocado dentro de cada um dos casos possíveis de operação (Soma, Multiplicação e Transposição), verificando se os arquivos de matriz foram informados. Caso não tenham sido, uma mensagem é impressa na tela e o programa é terminado para não ocorrer mais erros.

Exemplos *erroAssert*:

```
erroAssert(matrixPath1[0] != 0, "Deve ser informado um arquivo de texto contendo a matriz 1.");

erroAssert(matrixPath2[0] != 0, "Deve ser informado um arquivo de texto contendo a matriz 2.");

erroAssert(matrixPathRes[0] != 0, "Deve ser informado um arquivo de texto para armazenar o resultado da operação.");
```

mat.c

Neste arquivo, as estratégias de robustez são voltadas, principalmente, para evitar erros de acesso à memória desalocada. As estratégias para cada função serão melhor explicadas a seguir:

void dimensoesMatriz(mat_tipo *mat)

Esta função verifica se as linhas informadas no arquivo da matriz são válidas, ou seja, se são maiores que 0. Caso não sejam, um erro é retornado ao usuário e o programa é finalizado.

Exemplos *erroAssert*:

```
erroAssert(rows > 0, "As dimensões da matriz não podem ser nulas");

erroAssert(columns > 0, "As dimensões da matriz não podem ser nulas");
```

void criaMatrizInput(mat_tipo *mat, char *matrixPath, int id)

Além das mesmas verificações de *dimensoesMatriz*, já que ela é chamada por esta função, verifica também se o caminho informado para o txt da matriz é válido (não nulo) e se a alocação da matriz ocorreu corretamente.

Exemplos *erroAssert*:

```
erroAssert(matrixPath != NULL, "O caminho para a matriz precisa
ser especificado");

erroAssert(mat->m[i] != NULL, "Ocorreu um erro ao alocar a
matriz");
```

void criaMatrizOutput(mat_tipo *mat, char *matrixPath, int tamX, int tamY, int id)

Realiza as mesmas verificações que *dimensoesMatriz*, quanto aos parâmetros "tamX" e "tamY" informados. As outras verificações são idênticas às de *criaMatrizInput* e, por isso, não serão exemplificadas aqui.

void inicializaMatrizNula(mat_tipo *mat)

Verifica se a matriz informada como parâmetro já foi alocada, caso não tenha sido termina o programa com erro.

Exemplos *erroAssert*:

```
erroAssert(mat->m != NULL, "Matriz informada ainda não foi
alocada, portanto, não pode ser inicializada");
```

void leMatrizDoTxt(mat_tipo *mat)

Verifica se a matriz informada como parâmetro já foi alocada, caso não tenha sido termina o programa com erro. Como esta verificação já foi exemplificada em funções anteriores, não será exemplificada aqui. Verifica também se o arquivo de texto da matriz abriu corretamente, se não tiver sido, também termina o programa com erro.

A cada iteração pelas colunas, verifica também se havia algum elemento para ser lido, caso não haja, termina o programa com erro, já que a matriz claramente é menor que a especificada. Ao final da leitura do das dimensões informadas, tenta novamente ler outro

elemento, caso consiga, isso quer dizer que a matriz é maior do que a informada e um erro é retornado.

Exemplos *erroAssert*:

```
erroAssert(fscanf(arq, "%lf ", &num) == 1, "Matriz informada não
bate com as especificações ditas (menor)");

erroAssert(fscanf(arq, "%lf ", &num) == -1, "Matriz informada não
bate com as especificações ditas (maior)");

erroAssert(arq != NULL, "Ocorreu um erro ao abrir o arquivo da
matriz");
```

double acessaMatriz(mat_tipo *mat)

Esta função realiza uma única verificação, que é se a matriz informada já foi alocada. Se ela ainda não tiver sido, a função não é executada e um erro terminal é retornado.

void copiaMatrizes(mat_tipo *src, mat_tipo *dst)

Esta função verifica se as matrizes informadas como parâmetros já foram alocadas. Caso ainda não tenham sido, o programa finaliza com erro e a função não é executada.

void matrizParaTxt(mat_tipo *src)

A função verifica se a matriz informada já foi alocada. Caso não tenha sido, o programa termina com erro. Além disso, após abrir o arquivo de texto informado como atributo da matriz para a escrita, verifica também se a abertura ocorreu corretamente. Caso não tenha, o programa também termina com erro.

void somaMatrizes(mat_tipo *a, mat_tipo *b, mat_tipo *c)

A função verifica se as matrizes “a” e “b” informadas já foramalocadas. Essa verificação não se estende para “c”, já que esta matriz será recriada e inicializada como nula dentro da função. Caso as matrizes “a” e “b” ainda não tenham sido alocadas, o programa finaliza com erro.

É verificado também o pré-requisito da operação de soma de matrizes, que é ambas as matrizes somadas terem as exatas mesmas dimensões. Caso isso não ocorra, o programa também é finalizado com erro.

Exemplos *erroAssert*:

```
erroAssert(a->tamX == b->tamX, "Dimensoes incompativeis");

erroAssert(a->tamY == b->tamY, "Dimensoes incompativeis");
```

void multiplicaMatrizes(mat_tipo *a, mat_tipo *b, mat_tipo *c)

Da mesma forma que *somaMatrizes*, esta função verifica se as matrizes "a" e "b" informadas já foram alocadas. Caso não tenham sido, o programa finaliza com erro.

Além disso, o pré-requisito para multiplicação de matrizes também é verificado. Ou seja, caso as colunas de "a" não tenham as mesmas dimensões que as linhas de "b", o programa finaliza com erro.

Exemplos *erroAssert*:

```
erroAssert( a->tamY == b->tamX, "Dimensões incompativeis" );
```

void transpoeMatriz(mat_tipo *a, mat_tipo *b)

Esta função verifica se a matriz "a" já foi alocada. A verificação não se estende a "b", porque esta será alocada com a execução da função. Caso a matriz "a" não tenha sido alocada, o programa finaliza com erro.

void destroiMatriz(mat_tipo *a)

Esta função utiliza apenas a função *avisoAssert* para verificar se a matriz "a" já foi desalocada. Se já tiver sido, é lançado um aviso para o usuário sobre isso, mas a função não é interrompida.

Exemplos *avisoAssert*:

```
avisoAssert(a->m != NULL , "Matriz já foi desalocada");
```

Testes

Os testes para as operações de matrizes implementadas serão divididos entre testes de acesso à memória e testes de desempenho computacional para as diferentes operações. Os resultados das operações serão armazenados em arquivos .txt, os tempos de execução em arquivos .out, mapas de acesso à memória em arquivos .png e o teste gprof também em um arquivo .txt. Para maiores informações sobre onde encontrar estes arquivos, volte à seção de organização das pastas.

O teste de memória verifica a performance do programa quanto o acesso à memória alocada pelo mesmo. Será utilizada para metrificar, a ferramenta da “distância de pilha”. Basicamente, quando um endereço de memória é acessado, ele é inserido na pilha, quando ele for acessado novamente, ele é retirado da posição da pilha em que volta para a base da mesma. A distância da posição em que o elemento estava ao ser chamado para a primeira posição da fila será a distância de pilha daquela chamada em questão.

A distância de pilha total será calculada pela média ponderada da frequência em que um elemento é desempilhado com aquela distância, sendo que a distância em si é o peso da média. Quanto menor for a distância de pilha, melhor projetado é o código, indicando menor custo de memória.

Os testes de desempenho verificarão o tempo que leva para que uma função seja executada para um determinado tamanho de matriz de entrada. Para isso, será marcado o tempo de início da execução do programa e o tempo de finalização do mesmo. A diferença entre estes tempos é então calculada e será o tempo de execução do programa.

Será analisado não somente o tempo individual de execução para diferentes entradas, mas a evolução deste tempo e se ele comprova ou não a análise de complexidade feita em um dos tópicos anteriores.

Análise Experimental

A análise experimental foi dividida entre testes de acesso à memória e testes de desempenho computacional, que serão abordados em tópicos abaixo, com subdivisões para cada operação implementada. Para mais informações sobre os métodos de teste utilizados para esta análise experimental, consulte o tópico “Testes” deste documento.

Desempenho Computacional

Os testes de desempenho computacional foram realizados 5 vezes com crescimentos lineares nas matrizes quadradas de entrada. Elas começam com dimensão 500x500 e terminam com dimensão 900x900, crescendo de 100 em 100 dimensões simultaneamente para as linhas e as colunas.

A seguir, serão apresentados os desempenhos computacionais para cada uma das operações implementadas.

Soma

O desempenho computacional tem a ver com o tempo de execução, portanto, verificaremos a análise de complexidade de tempo para a operação de soma, que foi $O(n^2)$. Dividiremos esta análise em partes:

Saída gprof:

Os resultados gerados pelo *gprof* se referem aos custos individuais de cada função na operação como um todo. Para a operação de soma obtivemos:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 50.06      0.02     0.02        2    10.01    10.01
leMatrizDoTxt
 25.03      0.03     0.01        4     2.50     2.50  acessaMatriz
 25.03      0.04     0.01        1    10.01    10.01  somaMatrizes
  0.00      0.04     0.00        4     0.00     0.00
inicializaMatrizNula
```

```
  0.00      0.04      0.00        3     0.00     0.00
defineFaseMemLog
  0.00      0.04      0.00        3     0.00     0.00
destroiMatriz
  0.00      0.04      0.00        2     0.00     0.00
criaMatrizInput
  0.00      0.04      0.00        2     0.00     0.00
criaMatrizOutput
  0.00      0.04      0.00        2     0.00     0.00
dimensoesMatriz
  0.00      0.04      0.00        1     0.00     0.00  clkDifMemLog
  0.00      0.04      0.00        1     0.00     0.00
desativaMemLog
  0.00      0.04      0.00        1     0.00     0.00
finalizaMemLog
  0.00      0.04      0.00        1     0.00     0.00  iniciaMemLog
  0.00      0.04      0.00        1     0.00     0.00
matrizParaTxt
  0.00      0.04      0.00        1     0.00     0.00  parse_args
```

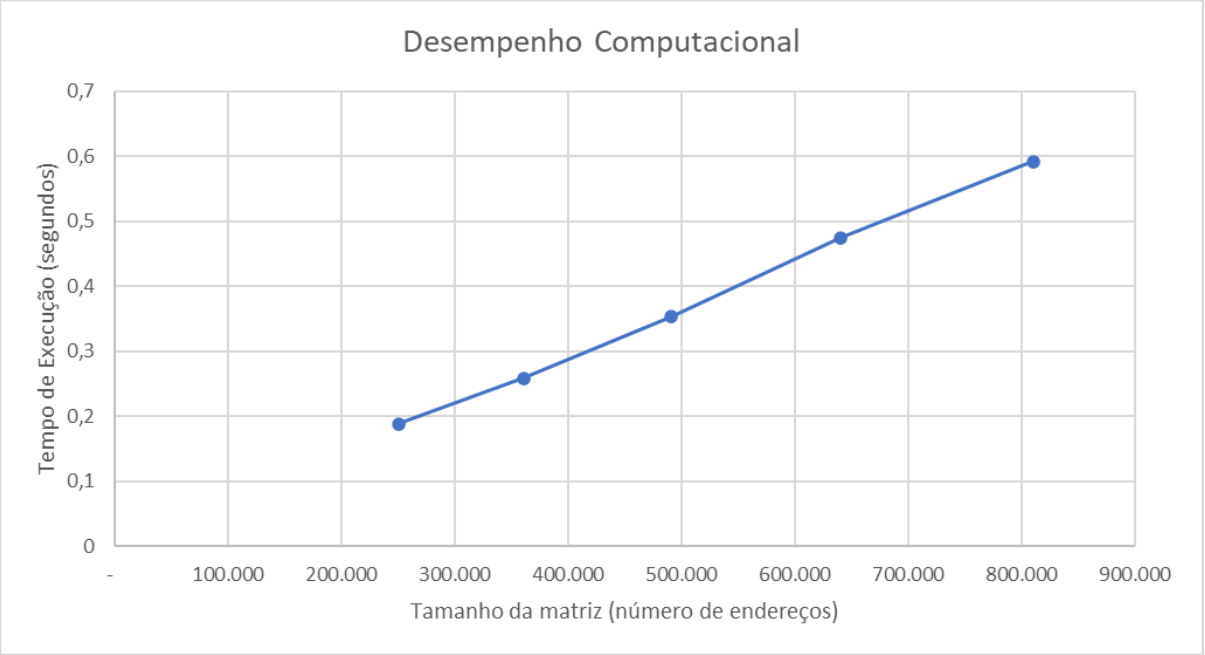
Percebemos que a função que mais gasta tempo é *leMatrizDoTxt*, infelizmente, não há como realizar muitas melhorias nela. Podemos, porém, suprimir a chamada da função *acessaMatriz*, que não faz diferença para a operação de soma, está presente apenas para fins de teste de memória.

Evolução do Desempenho Computacional

Veremos uma tabela que mostra a evolução do desempenho com o crescimento das matrizes de entrada para a operação de soma. Será exibido também um gráfico que mostra esta evolução de forma mais visual.

Número de linhas e colunas e nome do arquivo	**Tempo inicial**	**Tempo final**	**Duração da execução**
500 (soma500.out)	6737.519649286	6737.708333823	0.188684537
600 (soma600.out)	6738.820583860	6739.079204481	0.258620621

700 (soma700.out)	6741.073634387	6741.426423360	0.352788973
800 (soma800.out)	6744.816100179	6745.290678006	0.474577827
900 (soma900.out)	6750.287910863	6750.879309540	0.591398677



Observamos que o crescimento do tempo de execução, apesar de ter sido estimado como $O(n^2)$, apresenta crescimento linear. Isso, de forma alguma é prova para refutar a análise de complexidade do algoritmo. Na verdade, está muito mais associado ao tamanho das matrizes e o crescimento das mesmas. Infelizmente o sistema de testes não é tão potente para experimentos muito bruscos de desempenho computacional, então não será possível comprovar visualmente o crescimento quadrático da função.

Multiplicação

O desempenho computacional tem a ver com o tempo de execução, portanto, verificaremos a análise de complexidade de tempo para a operação de multiplicação, que foi $O(n^3)$. Dividiremos esta análise em partes:

Saída gprof:

Os resultados gerados pelo *gprof* se referem aos custos individuais de cada função na operação como um todo. Para a operação de multiplicação, obtivemos:

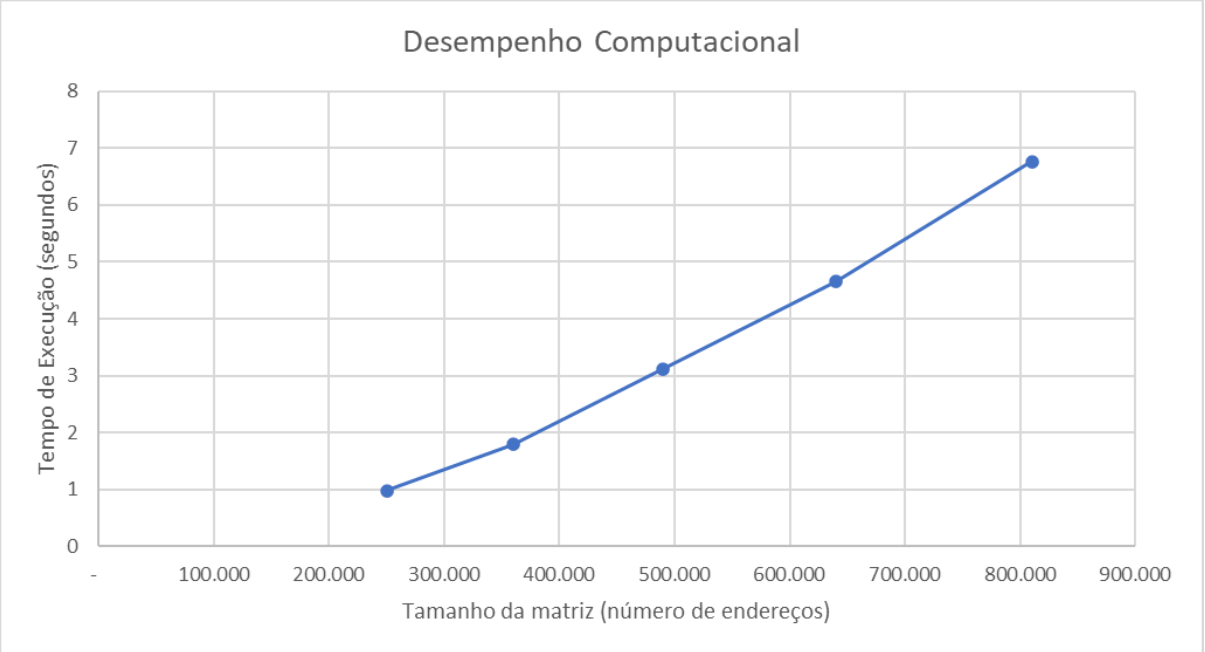
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 99.95      6.05     6.05        1     6.05     6.05
multiplicaMatrizes
  0.17      6.06     0.01        4     0.00     0.00
inicializaMatrizNula
  0.00      6.06     0.00        4     0.00     0.00  acessaMatriz
  0.00      6.06     0.00        3     0.00     0.00
defineFaseMemLog
  0.00      6.06     0.00        3     0.00     0.00
destroiMatriz
  0.00      6.06     0.00        2     0.00     0.00
criaMatrizInput
  0.00      6.06     0.00        2     0.00     0.00
criaMatrizOutput
  0.00      6.06     0.00        2     0.00     0.00
dimensoesMatriz
  0.00      6.06     0.00        2     0.00     0.00
leMatrizDoTxt
  0.00      6.06     0.00        1     0.00     0.00  clkDifMemLog
  0.00      6.06     0.00        1     0.00     0.00
desativaMemLog
  0.00      6.06     0.00        1     0.00     0.00
finalizaMemLog
  0.00      6.06     0.00        1     0.00     0.00  iniciaMemLog
  0.00      6.06     0.00        1     0.00     0.00
matrizParaTxt
  0.00      6.06     0.00        1     0.00     0.00  parse_args
```

Percebemos que a função que mais gasta tempo é *multiplicaMatrizes*. Infelizmente, para matrizes não quadráticas, não há nada que possamos fazer. Mas, para matrizes quadradas, existe o chamado algoritmo de *Strassen*, que é de implementação mais complexa e é executado ligeiramente mais rápido.

Evolução do Desempenho Computacional

Veremos uma tabela que mostra a evolução do desempenho com o crescimento das matrizes de entrada para a operação de multiplicação. Será exibido também um gráfico que mostra esta evolução de forma mais visual.

Número de linhas e colunas e nome do arquivo	**Tempo inicial**	**Tempo final**	**Duração da execução**
500 (mult500.out)	6737.709367530	6738.686514055	0.977146525
600 (mult600.out)	6739.080244267	6740.877630120	1.797385853
700 (mult700.out)	6741.427610588	6744.552217993	3.124607405
800 (mult800.out)	6745.292157540	6749.950594034	4.658436494
900 (mult900.out)	6750.880765664	6757.640726316	6.759960652



Observamos que o crescimento do tempo de execução, apesar de ter sido estimado como $O(n^3)$, assim como a operação de soma, apresenta crescimento levemente linear. Isso, de forma alguma é prova para refutar a análise de complexidade do algoritmo. Na verdade, está muito mais associado ao tamanho das matrizes e ao crescimento das mesmas. Infelizmente o sistema de testes não é tão potente para experimentos muito bruscos de desempenho computacional, então não será possível comprovar visualmente o crescimento cúbico da função.

Transposição

O desempenho computacional tem a ver com o tempo de execução, portanto, verificaremos a análise de complexidade de tempo para a operação de transposição, que foi $O(n^2)$. Dividiremos esta análise em partes:

Saída gprof:

Os resultados gerados pelo *gprof* se referem aos custos individuais de cada função na operação como um todo. Para a operação de multiplicação, obtivemos:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.37      0.01     0.01        3     3.34     3.34
inicializaMatrizNula
 33.37      0.02     0.01        2     5.01     5.01  acessaMatriz
 33.37      0.03     0.01        1    10.01    13.35
leMatrizDoTxt
  0.00      0.03     0.00        3     0.00     0.00
defineFaseMemLog
  0.00      0.03     0.00        2     0.00     0.00
criaMatrizOutput
  0.00      0.03     0.00        2     0.00     0.00
destroiMatriz
  0.00      0.03     0.00        1     0.00     0.00  clkDifMemLog
  0.00      0.03     0.00        1     0.00     0.00
criaMatrizInput
  0.00      0.03     0.00        1     0.00     0.00
```

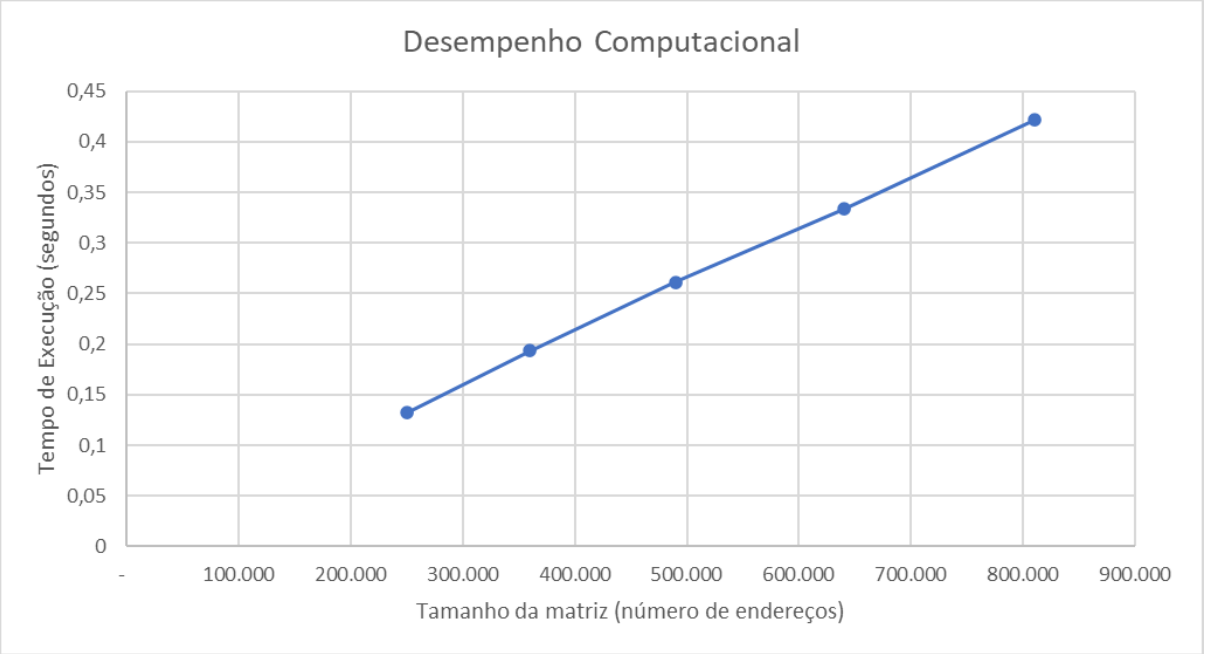
```
desativaMemLog
  0.00      0.03     0.00        1     0.00     0.00
dimensoesMatriz
  0.00      0.03     0.00        1     0.00     0.00
finalizaMemLog
  0.00      0.03     0.00        1     0.00     0.00  iniciaMemLog
  0.00      0.03     0.00        1     0.00     0.00
matrizParaTxt
  0.00      0.03     0.00        1     0.00     0.00  parse_args
  0.00      0.03     0.00        1     0.00     3.34
transpoeMatriz
```

Percebemos que a função que mais gasta tempo é *inicializaMatrizNula*. Infelizmente, não há muito que se possa fazer em termos de melhoria, já que esta função é crucial para todas as operações.

Evolução do Desempenho Computacional

Veremos uma tabela que mostra a evolução do desempenho com o crescimento das matrizes de entrada para a operação de transposição. Será exibido também um gráfico que mostra esta evolução de forma mais visual.

Número de linhas e colunas e nome do arquivo	**Tempo inicial**	**Tempo final**	**Duração da execução**
500 (transp500.out)	6738.687446386	6738.819687723	0.132241337
600 (transp600.out)	6740.879232769	6741.072717902	0.193485133
700 (transp700.out)	6744.553464804	6744.814609249	0.261144445
800 (transp800.out)	6749.951937201	6750.285688028	0.333750827
900 (transp900.out)	6757.642252701	6758.063654590	0.421401889



Observamos que o crescimento do tempo de execução, apesar de ter sido estimado como $O(n^2)$, assim como a operação de soma e multiplicação, apresenta crescimento levemente linear. Isso, de forma alguma é prova para refutar a análise de complexidade do algoritmo. Na verdade, está muito mais associado ao tamanho das matrizes e ao crescimento das mesmas. Infelizmente o sistema de testes não é tão potente para experimentos muito bruscos de desempenho computacional, então não será possível comprovar visualmente o crescimento cúbico da função.

Análise de Padrão de Acesso à Memória e Localidade de Referência

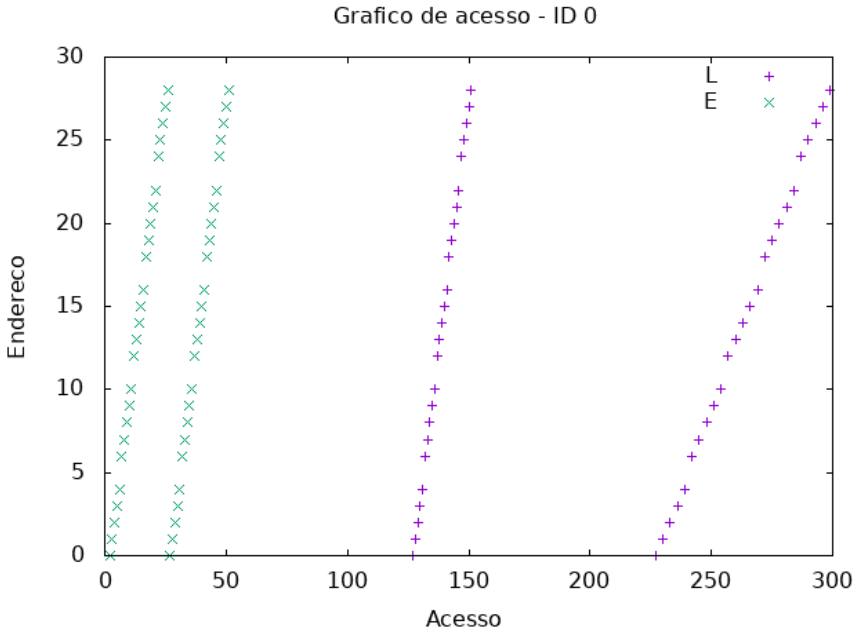
Os testes de memória serão realizados com matrizes 5x5 para melhor visualização dos acessos. Não serão repetidos estes testes, já que eles são apenas para geração dos gráficos.

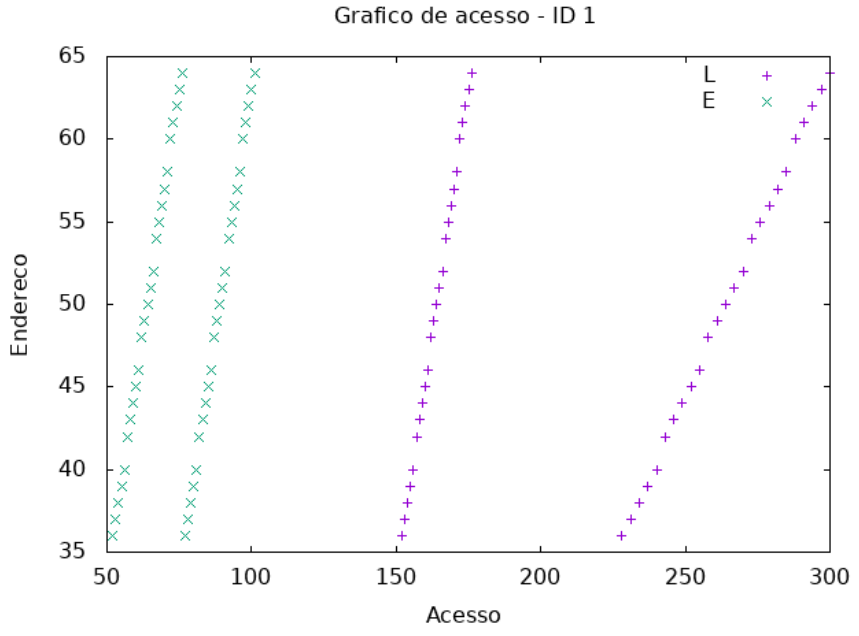
Soma

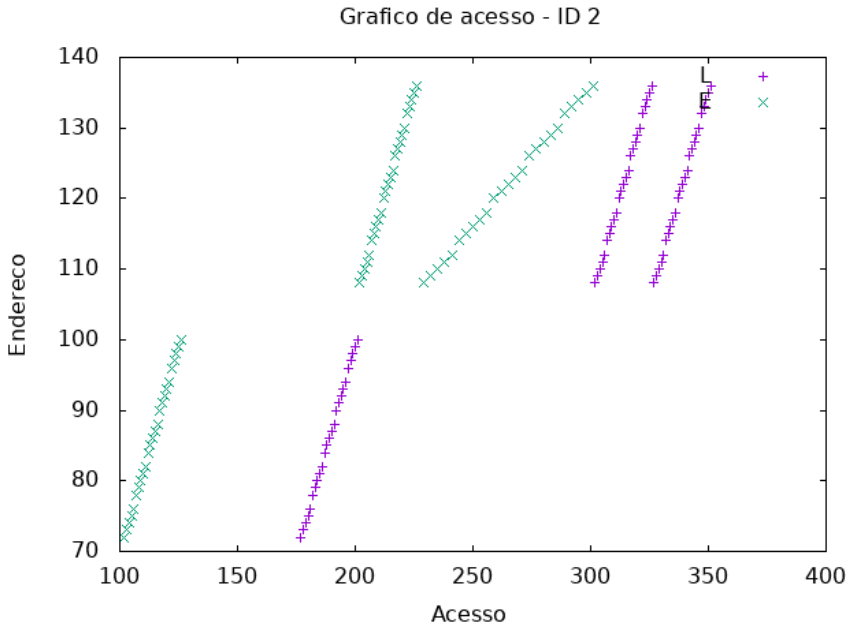
Como os experimentos não são relativos ao crescimento da entrada, a análise de complexidade de espaço desta operação não será levada em consideração. A análise será dividida em partes:

Mapa de acesso das matrizes

Este gráfico mostra como os endereços da matriz são acessados a cada chamada de função, assim como a legenda explicita, pontos em formato de “X” na cor verde representam acesso à memória para escrita, já os em formato de “+” na cor rosa, representam acesso para leitura.





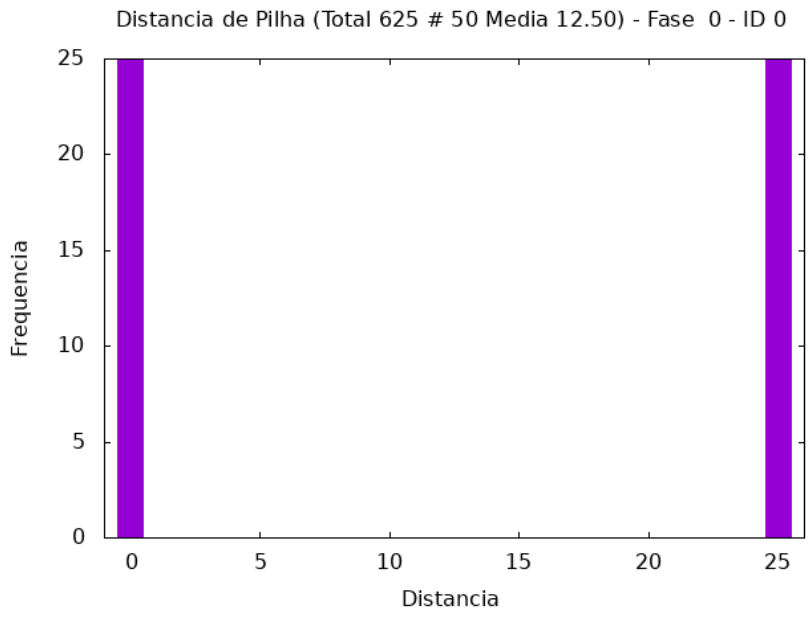


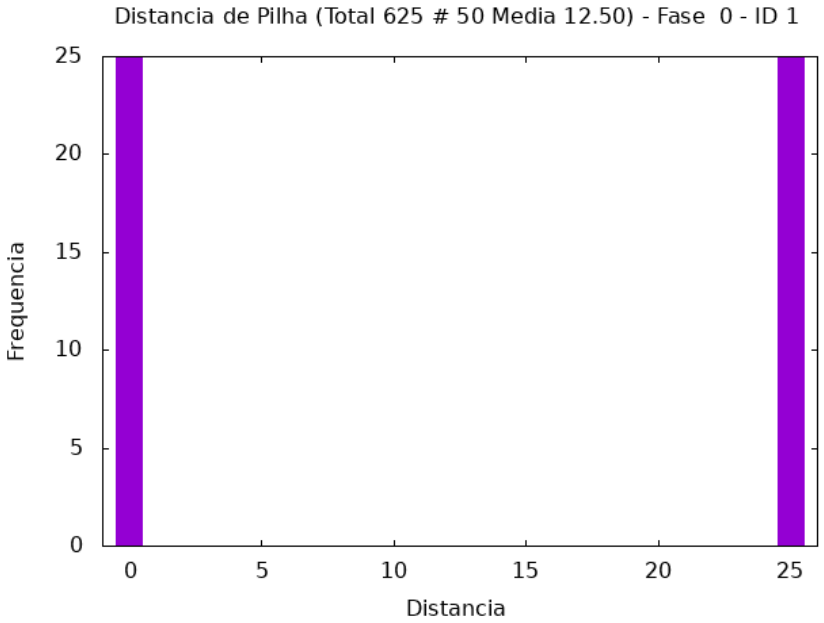
É possível observar que os gráficos de ID 0 e 1 são bem semelhantes, por se tratarem de matrizes de input. Elas recebem as mesmas chamadas de função e sofrem operações parecidas. Já o último gráfico de acesso mostra muito mais endereços acessados, já que, para todas as operações, é feita uma realocação de memória para garantir que a matriz terá as dimensões adequadas.

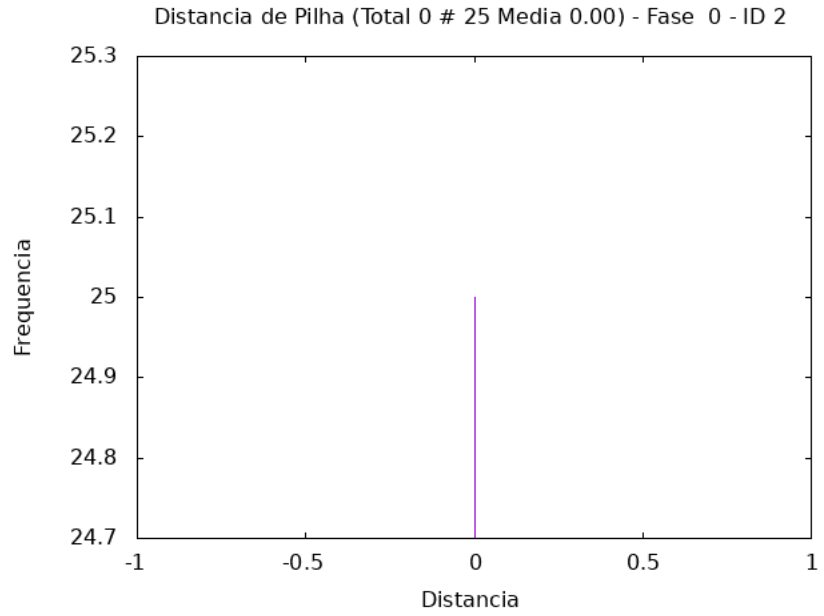
Os endereços de memória são acessados linearmente.

Distância de pilha

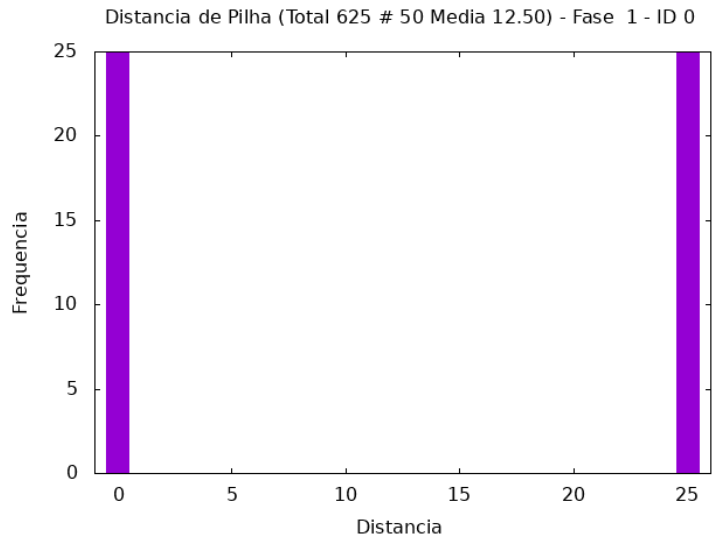
Estes gráficos são chamados de histogramas e mostram a distância de pilha dos acessos aos endereços de memória alocados para as matrizes.

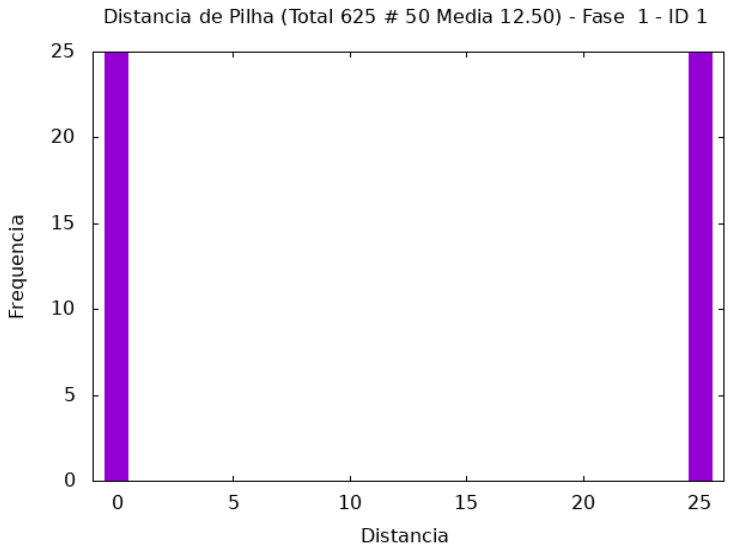


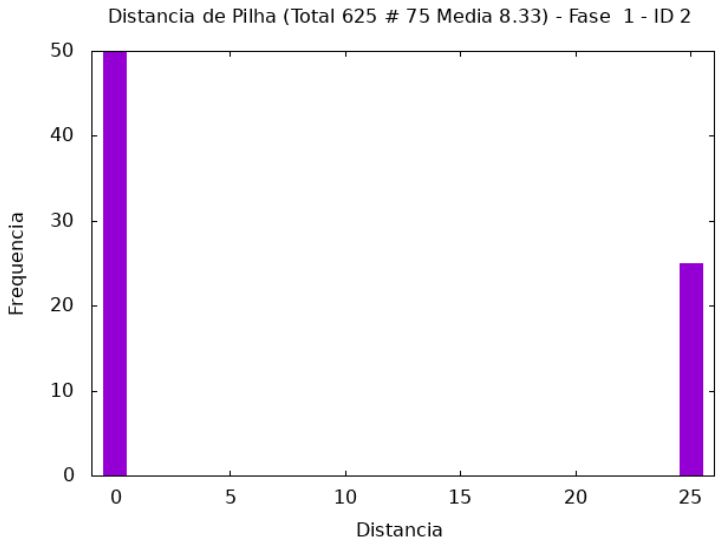




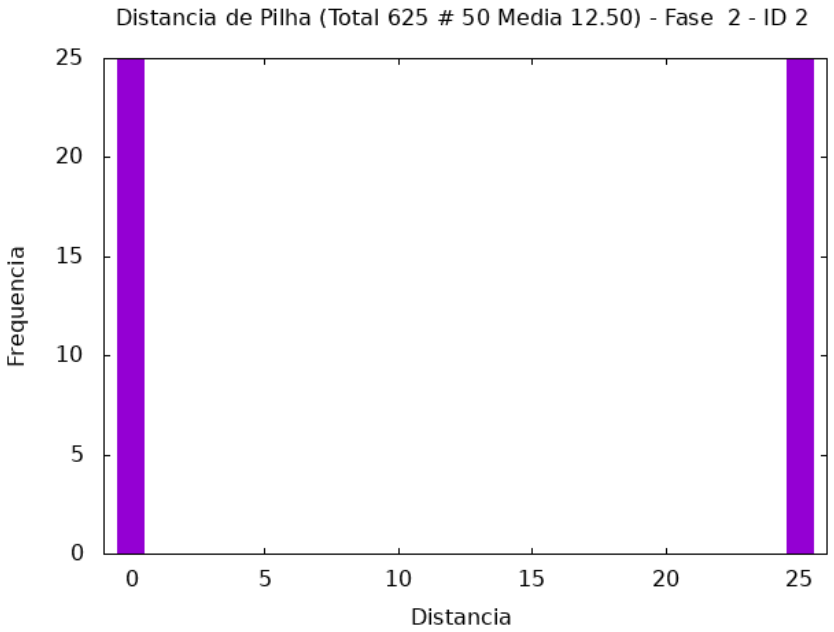
É notável que na fase 0, todas as matrizes são inicializadas, portanto, possuem histogramas semelhantes. O último gráfico porém é diferente, já que, após ser alocada, a matriz de resultados só é inicializada como nula, já que os valores ainda não estão escritos no txt.


Distancia de Pilha (Total 625 # 50 Media 12.50) - Fase 1 - ID 0
Frequencia
Distancia
0
5
10
15
20
25


Distancia de Pilha (Total 625 # 50 Media 12.50) - Fase 1 - ID 1
Frequencia
Distancia
0
5
10
15
20
25


Distancia de Pilha (Total 625 # 75 Media 8.33) - Fase 1 - ID 2
Frequencia
Distancia
0
10
20
30
40
50
0
5
10
15
20
25

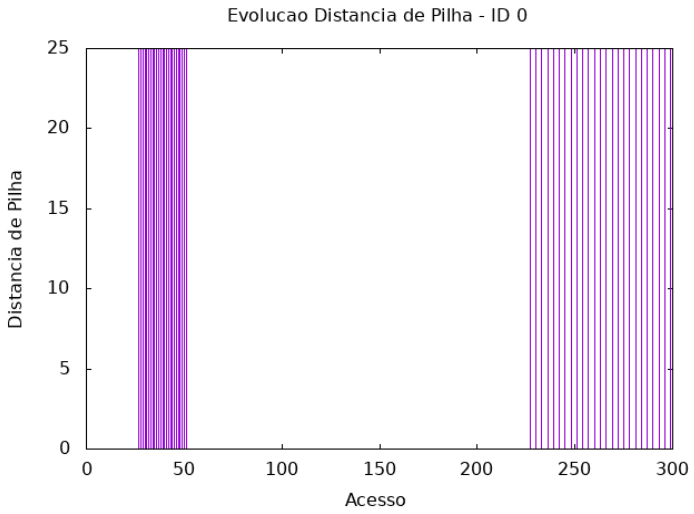
Na fase 1, a operação é efetivamente realizada. O útlimo gráfico destoa dos demais por ter uma nova alocação dinâmica nesta fase. Desta forma, há mais distâncias de pilha 0 na chamada.

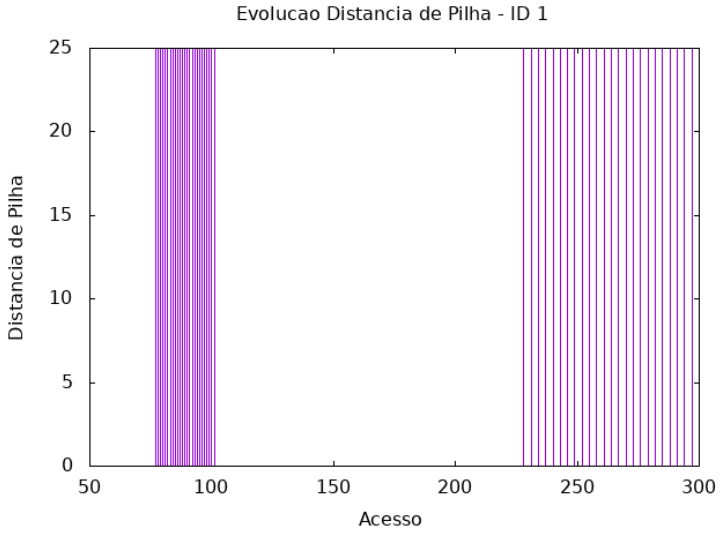


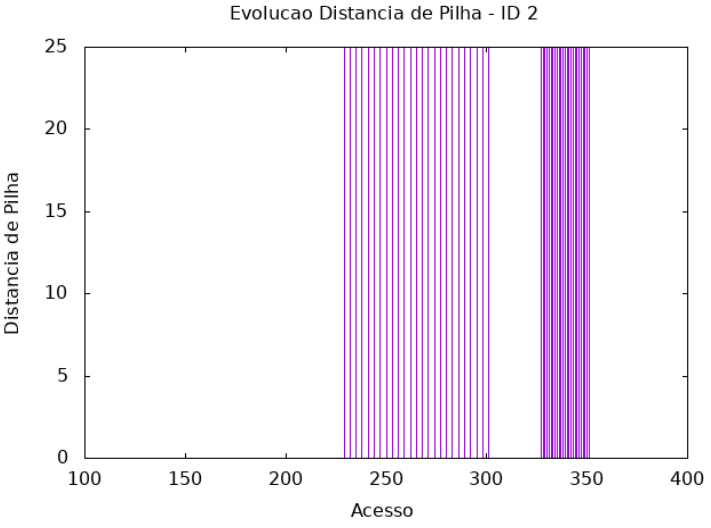
Na última fase, somente a matriz de resultado tem seus elementos acessados. Eles são acessados para leitura e para escrita no arquivo .txt.

Evolução da Distância de Pilha

O acesso deve ser intenso no começo e mais leve ao final.


Evolucao Distancia de Pilha - ID 0
Distancia de Pilha
Acesso
0
5
10
15
20
25
0
50
100
150
200
250
300


Evolucao Distancia de Pilha - ID 1
Distancia de Pilha
Acesso
0
5
10
15
20
25
50
100
150
200
250
300


Evolucao Distancia de Pilha - ID 2
Distancia de Pilha
Acesso
0
5
10
15
20
25
100
150
200
250
300
350
400

Conclusão

Neste trabalho, foi implementado um Tipo Abstrato de Dados para matriz com alocação dinâmica. Para ele, foram criadas funções que permitiram operações de Soma, Multiplicação e Transposição. Toda a implementação foi feita na linguagem C, utilizando o compilador GCC, sendo também desenvolvido em sistema Linux.

Foi possível relembrar conceitos de programação dinâmica, sintaxe de programação em C, bem como o aprendizado de ferramentas de análise de desempenho de execução e de acesso à memória, assim como foi explicitado no tópico anterior.

Bibliografia

Meira, W. and Pappa, G. (2021). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Instruções para compilação e execução

1. Extraia o arquivo .zip presente;
2. Abra um terminal na pasta raiz do mesmo;
3. Digite “make”. Isso compilará o programa;
4. Digite “./bin/matop”. Isso invocará a função de uso do programa, com detalhes mais específicos sobre sua execução.