Trabalho Prático 0

Luiz Felipe de Sousa Faria – 2020027148

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

lutilipe@ufmg.br

1. Introdução

O problema proposto foi implementar um Tipo Abstrato de Dado (TAD) de uma matriz. O objetivo da documentação é analisar o desempenho, a robustez, a localidade de referência e distância de pilha dessa estrutura diante de operações comuns que elas realizam. No trabalho, foram implementadas e analisadas as operações de soma, multiplicação e transposição da matriz.

2. Implementação

O programa principal se encontra no diretório ***TP***. Esse diretório é dividido em:

- ***include***: contém o arquivo *header* das funções da matriz e do *memlog*. Contém também as macros de *asserts*;
- ***src***: contém a implementação das funções do include;
- ***obj***: contém os .o gerados pelo programa (vazio inicialmente)
- ***bin***: contém o binário do programa (vazio inicialmente)
- ***Makefile***: gera os arquivos binário e *.o* do programa e executa os testes de desempenho e análise de localidade. O *output* do programa se encontra na pasta *out*, localizada fora do diretório ***TP***.

Fora desse diretório, existem outros diretórios de utilidade para o projeto. Além do diretório out já citado, existe o diretório in, contendo as matrizes de entrada do programa, e o diretório analisamem, contendo o source e o binário do programa utilizado para fazer análises de localidade de referência

O programa foi escrito em *C* e compilado utilizando o *GCC*.

2.1. Estrutura de dados

A estrutura de dado principal do projeto é a matriz, uma forma de vetor em duas dimensões. Sua *struct* possui os atributos:

- ***m:*** correspondente a matriz em si. Esse atributo é um ponteiro para um ponteiro do tipo double;
- M e N: correspondem a quantidade de linhas e colunas, respectivamente;
- id: identificador da matriz para fins de *logs*.

O atributo *m* da estrutura é alocado dinamicamente. Isso permite a atribuição da dimensão da matriz, seja por um *input* externo ou alguma função interna, durante a execução do programa.

2.2. Funcionamento

O programa recebe como parâmetro o caminho para um arquivo contendo a primeira matriz (parâmetro *-1*) e outro caminho para outro arquivo contendo a segunda matriz (*-2*). Recebe também o caminho do arquivo de saída (*-o*). Esses arquivos possuem a primeira linha contendo as dimensões da matriz. O restante das linhas contém a matriz em si.

Pelos parâmetros, também são recebidos a operação a ser realizada (*-s* para soma; *-t* para transpor; e *-m* para multiplicação); o parâmetro *-l* para ativação do log de memória e o parâmetro *-p* indicando o caminho do output desse log.

As principais funções do programa são:

- ***initMatrixFromFile:*** essa função irá receber como parâmetro o nome do arquivo que contém a matriz que queremos usar; o ponteiro de uma *Matriz*; e o id da matriz. Ela irá ler a primeira linha do arquivo contendo a quantidade de linhas e colunas da matriz e criar uma matriz com essas dimensões. Após isso, inicializará a matriz com os valores do arquivo;
- ***writeMatrixToFile***: irá receber o nome do arquivo de saída e a *Matrix* contendo o resultado final de uma operação. A forma como a matriz é escrita no arquivo segue a mesma estrutura dos arquivos de entrada;
- ***sumMatrix***: realiza a soma de duas matrizes de dimensões iguais. Para cada posição das matrizes A e B, é obtido o valor resultante da soma de cada uma dessas posições. Esse valor é então armazenado em uma terceira matriz C. O ponteiro dessas matrizes são recebidos no parâmetro da função;
- ***multiplyMatrix***: realiza a multiplicação da matriz A pela matriz B, armazenando o resultado em C. Para cada linha da matriz A, obtém-se o valor da célula e é realizada a multiplicação escalar pela célula correspondente (mesmo *index*) na coluna B. O resultado é armazenado na posição (linha A)x(coluna B) na *Matrix* C.
- ***transposeMatrix***: realiza a transposição de uma matriz. Recebe como parâmetro a matriz a ser transposta e a matriz de destino na qual será armazenada o resultado da operação. A leitura é feita por linha na matriz A (origem) e armazenada por coluna na matriz B (destino). Essa função, ao contrário das outras duas relacionadas a operações (*sumMatrix* e

multiplyMatrix), necessita apenas de um arquivo de entrada contendo uma matriz. O segundo arquivo (caso seja passado por parâmetro), é ignorado.

3. Análise de complexidade

Para a análise da complexidade de tempo das funções e do programa em si, consideremos uma matriz de dimensões M x N (número de linhas e número de colunas, respectivamente).

Para as funções de soma, transposição, leitura e escrita em arquivos, a análise é bem similar, por isso será feita em conjunto. Nessas funções, todos os elementos da matriz vão ser acessados uma vez. No caso da função soma, duas matrizes são acessadas na mesma operação para que seja possível obter o valor delas e inseri-las na matriz resultante. No entanto, isso não interfere na análise. Para tais funções, a complexidade dá-se por:

$$O(M) \,.\, O(N) \;=\; O(N.M)$$

Para a função de multiplicação, a complexidade é diferente. Nessa função, cada elemento de uma linha da matriz A será multiplicado pelo seu correspondente na coluna da matriz B. O resultado de cada multiplicação será somado e adicionado em uma posição da matriz C. O processo se repete até que a matriz C tenha a quantidade de linhas iguais às linhas da matriz A, e colunas iguais às linhas da matriz B. Sendo assim, a complexidade desse processo se dá por:

$$O(M) \,.\, O(N) \,.\, O(N) \;=\; O(N^2.M)$$

Caso ambas as matrizes sejam quadradas, a complexidade será $O(N^3)$.

A complexidade geral do programa dependerá da operação escolhida.

4. Estratégias de robustez

As operações com matrizes, em sua maioria, exigem que certas condições sejam satisfeitas previamente. Caso contrário, essas operações não devem ocorrer. O programa apresenta estratégias para lidar com essas condições.

Na inicialização da matriz pelo arquivo, é validado o tipo e valor (maior que zero:) das dimensões a serem definidas para a estrutura, bem como os valores a serem inseridos nela. Nesse último caso, é checado se o valor a ser inserido é um

número. É checado também se o arquivo de entrada existe. Caso algum deles seja inválido, o programa para e gera uma mensagem de erro.

Para as funções que realizam operações sobre a estrutura, há uma checagem em relação às dimensões das matrizes. No caso da soma, é checado se a quantidade de linhas e colunas da matriz A é igual a da matriz B. Na multiplicação, verifica-se se a quantidade de colunas em A é igual a quantidade de linhas em B. Caso o contrário, também é gerado um erro. Na operação de transposição, é checado apenas a dimensão da matriz, para ver se ela é nula ou não.

Por fim, a função que escreve o resultado da operação no arquivo checa se o arquivo de destino existe. Caso não, cria no lugar. Se essa criação falhar, é gerado um erro com uma mensagem.

5. Testes

5.1. Plano experimental

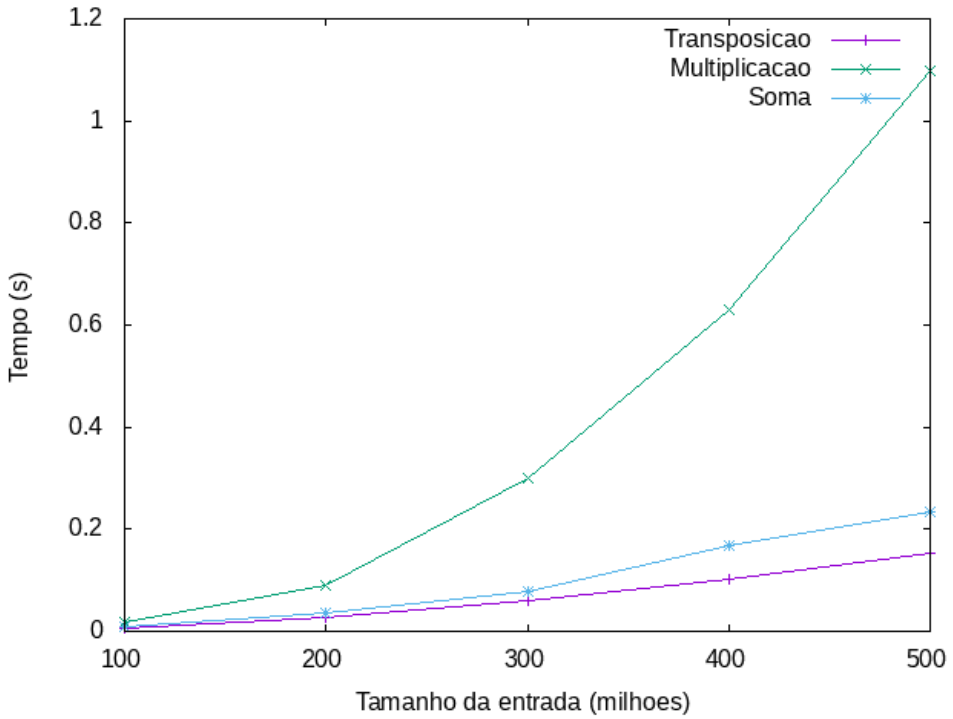
Para analisar a fundo o algoritmo e as operações (soma, multiplicação e transposição), foram realizados testes de desempenho computacional e análise de localidade de referência, juntamente com o padrão de acesso em memória. Os testes foram feitos com diferentes tamanhos de matrizes.

Para o teste de memória, foram utilizadas matrizes de dimensões 20x20. Já o teste de análise de desempenho foi dividido em duas etapas: a primeira analisando a evolução do tempo de execução de cada operação, na qual foram utilizadas matrizes quadradas de dimensões entre [100-500]; já a segunda etapa, analisou-se o tempo de execução de cada função de cada operação. Para tal, foi utilizado uma matriz de dimensão 1000x1000.

Esses experimentos foram realizados em um ambiente sem interferência de programas rodando em *background* que utilizam CPU ou memória RAM do computador. Os resultados podem ser vistos nas seções a seguir.

5.2. Resultados

5.2.1. Desempenho computacional



5.2.1.1. Soma

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
50.07	0.02	0.02	4	05.01	05.01	accessMatrix
25.04	0.03	0.01	2	05.01	05.01	initMatrixFromFile
25.04	0.04	0.01	1	10.01	10.01	sumMatrix
0.00	0.04	0.00	4	0.00	0.00	createMatrix
0.00	0.04	0.00	3	0.00	0.00	defineFaseMemLog
0.00	0.04	0.00	3	0.00	0.00	destroyMatrix
0.00	0.04	0.00	3	0.00	0.00	initNullMatrix
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	parse_args
0.00	0.04	0.00	1	0.00	0.00	writeMatrixToFile

5.2.1.2. Multiplicação

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
99.97	11.46	11.46	1	11.46	11.46	multiplyMatrix
0.09	11.47	0.01	4	0.00	0.00	accessMatrix
0.09	11.48	0.01	1	0.01	0.01	writeMatrixToFile
0.00	11.48	0.00	4	0.00	0.00	createMatrix
0.00	11.48	0.00	3	0.00	0.00	defineFaseMemLog
0.00	11.48	0.00	3	0.00	0.00	destroyMatrix
0.00	11.48	0.00	3	0.00	0.00	initNullMatrix
0.00	11.48	0.00	2	0.00	0.00	initMatrixFromFile
0.00	11.48	0.00	1	0.00	0.00	clkDifMemLog
0.00	11.48	0.00	1	0.00	0.00	desativaMemLog
0.00	11.48	0.00	1	0.00	0.00	finalizaMemLog
0.00	11.48	0.00	1	0.00	0.00	iniciaMemLog
0.00	11.48	0.00	1	0.00	0.00	parse_args

5.2.1.3. Transposição

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
50.07	0.02	0.02	3	6.68	6.68	accessMatrix
25.04	0.03	0.01	1	10.01	10.01	transposeMatrix
25.04	0.04	0.01	1	10.01	10.01	writeMatrixToFile
0.00	0.04	0.00	3	0.00	0.00	createMatrix
0.00	0.04	0.00	3	0.00	0.00	defineFaseMemLog
0.00	0.04	0.00	2	0.00	0.00	destroyMatrix
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	initMatrixFromFile
0.00	0.04	0.00	1	0.00	0.00	initNullMatrix
0.00	0.04	0.00	1	0.00	0.00	parse_args

5.2.2. Localidade de referência e acesso à memória

5.2.2.1. Soma

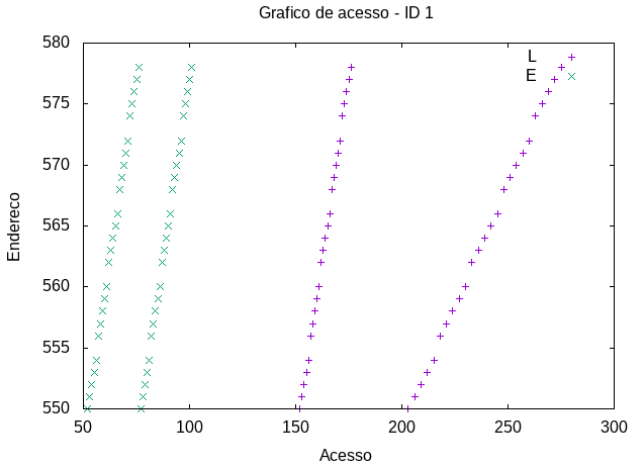


Grafico de acesso - ID 1
Endereco
Acesso
L
E

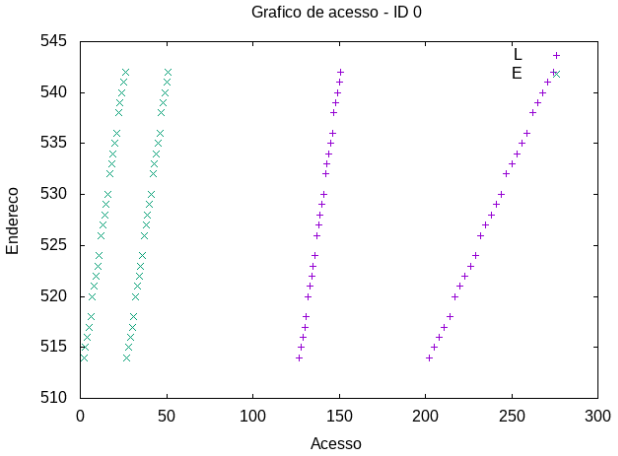


Grafico de acesso - ID 0
Endereco
Acesso
L
E

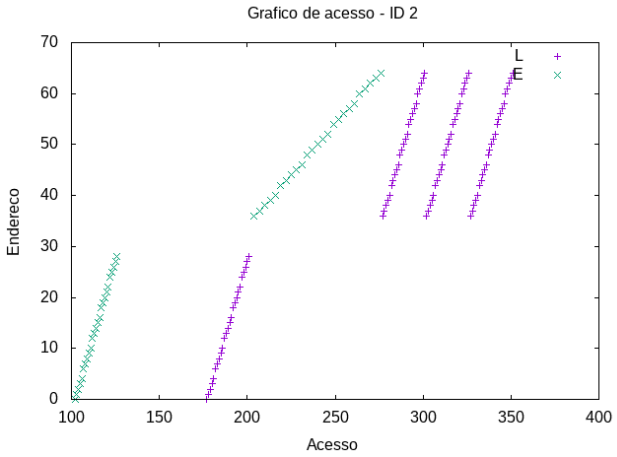
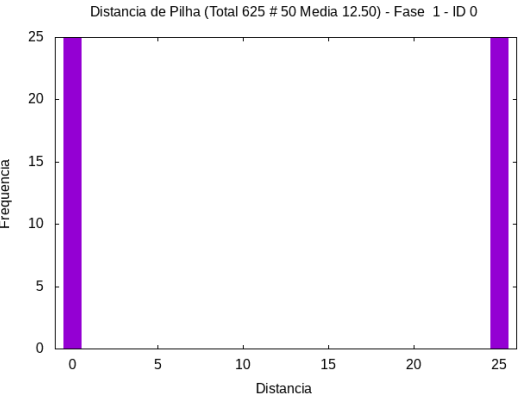
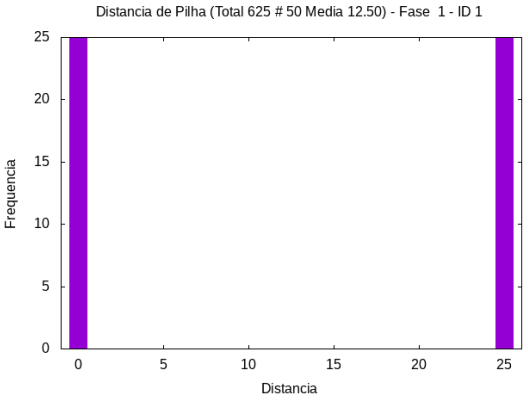
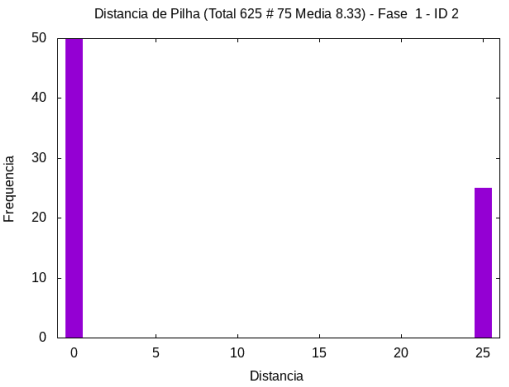
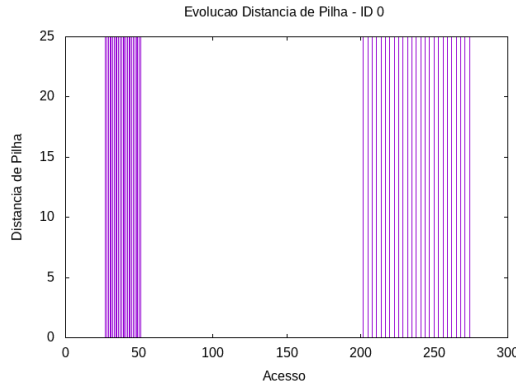


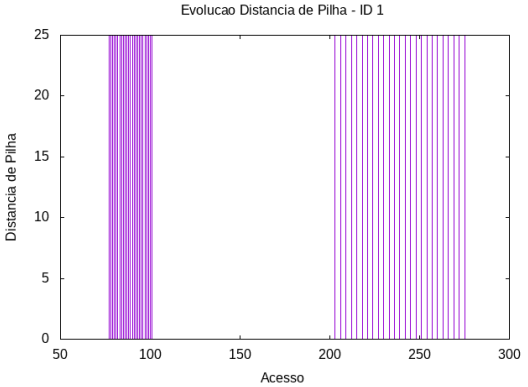
Grafico de acesso - ID 2
Endereco
Acesso
L
E

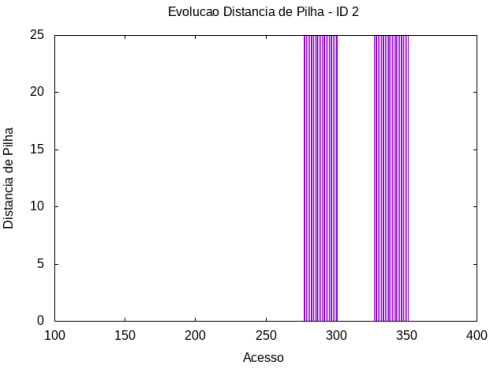

Distancia de Pilha (Total 625 # 50 Media 12.50) - Fase 1 - ID 0
Frequencia
Distancia


Distancia de Pilha (Total 625 # 50 Media 12.50) - Fase 1 - ID 1
Frequencia
Distancia

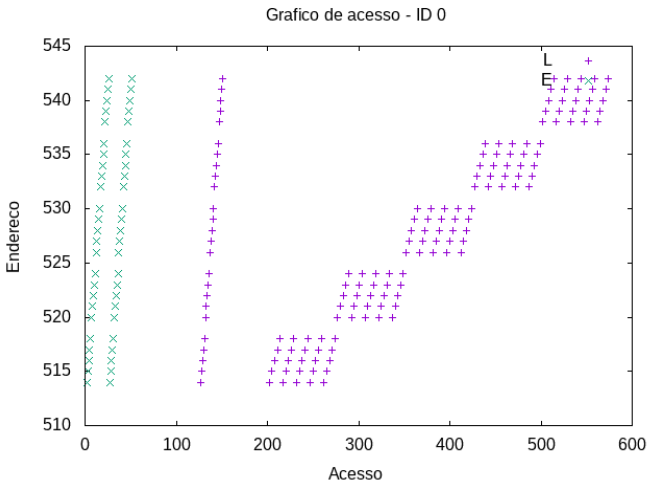

Distancia de Pilha (Total 625 # 75 Media 8.33) - Fase 1 - ID 2
Frequencia
Distancia

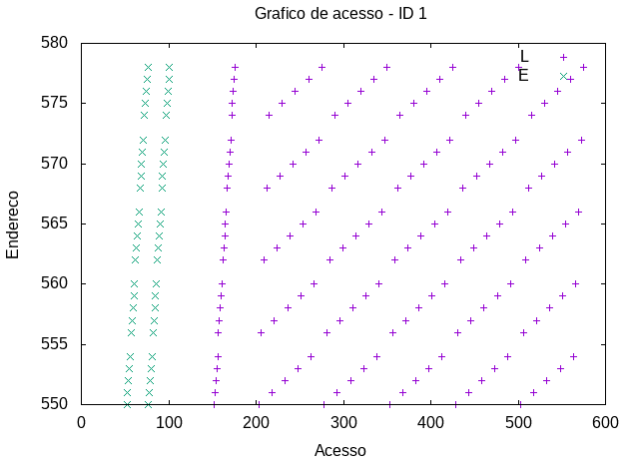

Evolucao Distancia de Pilha - ID 0
Distancia de Pilha
Acesso

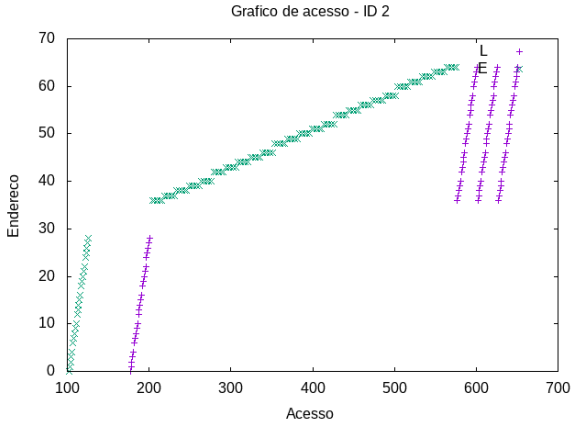

Evolucao Distancia de Pilha - ID 1
Distancia de Pilha
Acesso

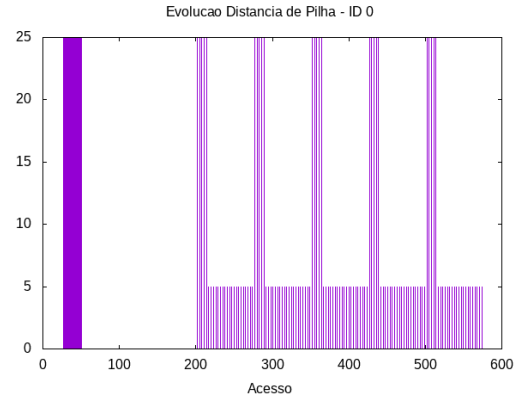

Evolucao Distancia de Pilha - ID 2
Distancia de Pilha
Acesso

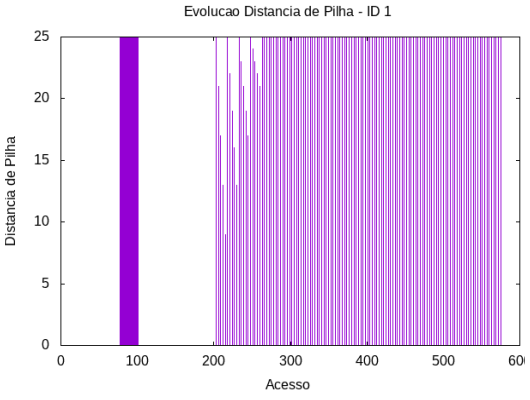
5.2.2.1. Multiplicação

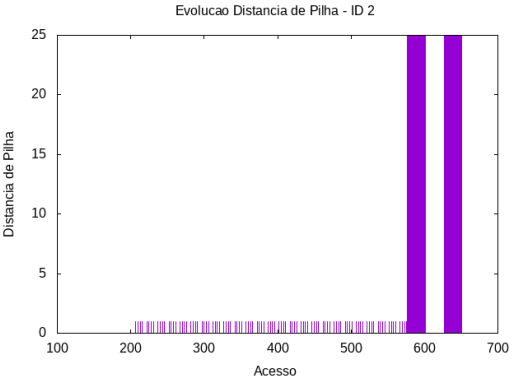


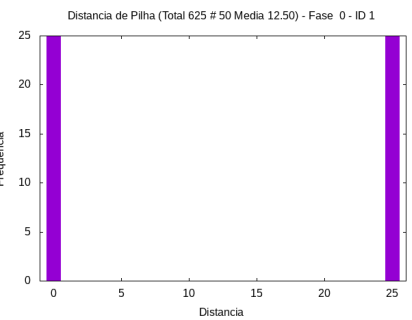


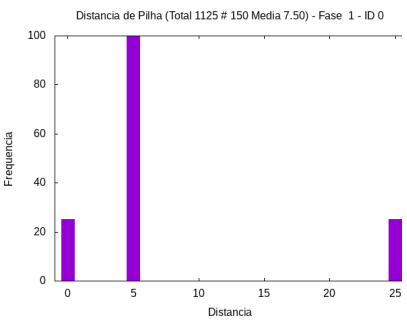


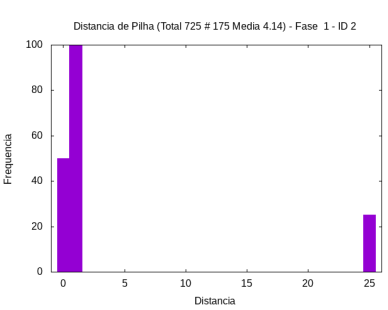


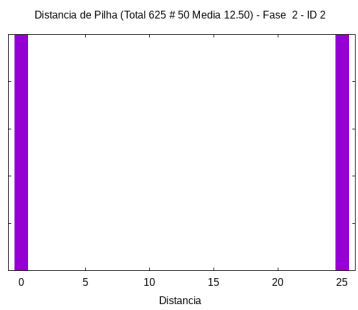




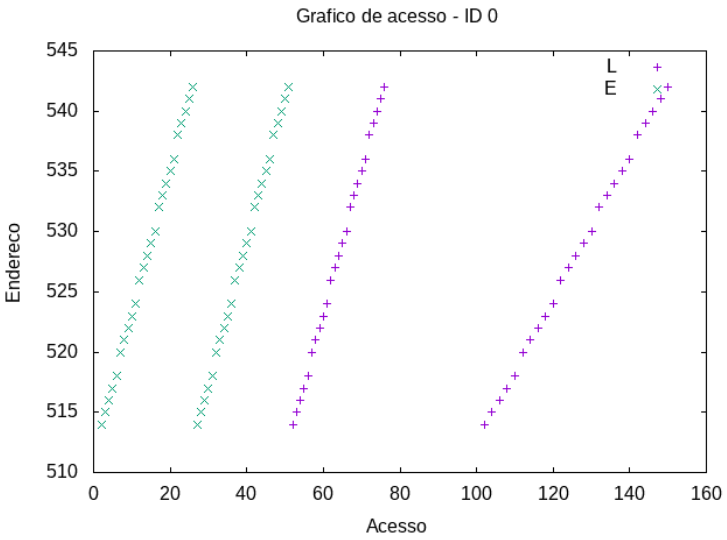


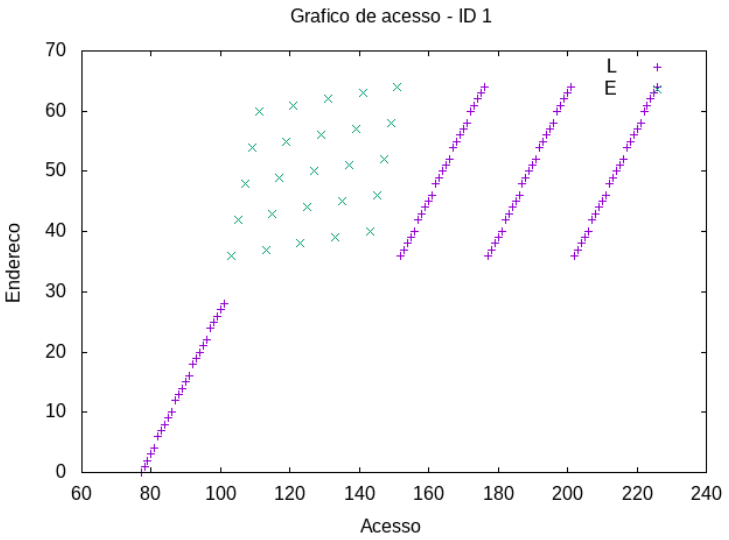


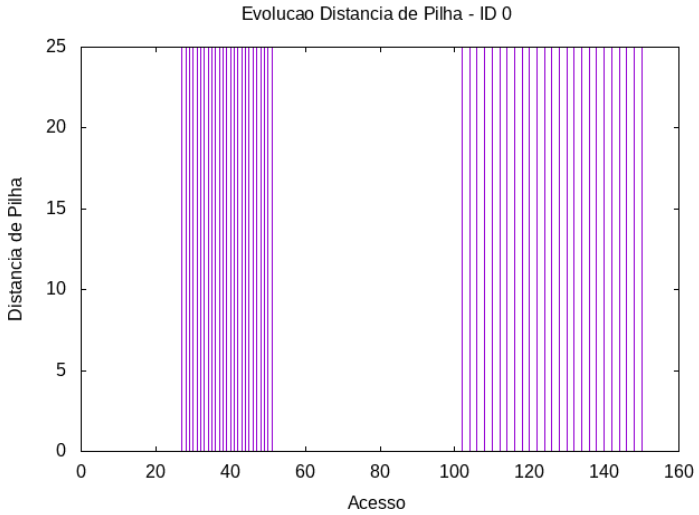


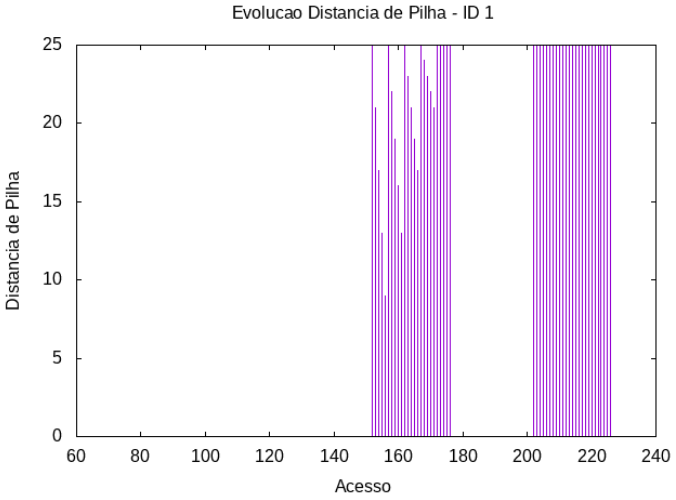


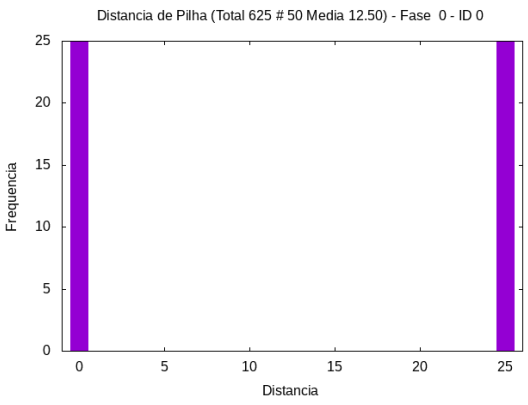
5.2.2.1. Transposição

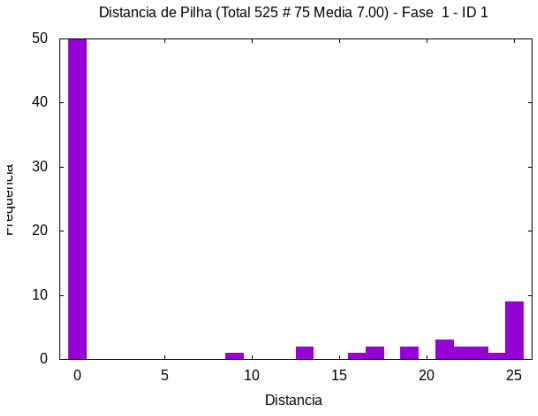


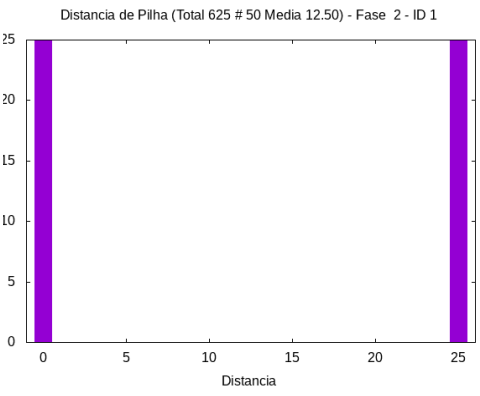












6. Análise experimental

7. Conclusões

O presente trabalho propôs-se a analisar os algoritmos de operações básicas em uma das estruturas de dados mais importantes, as matrizes. Para tal, foram realizadas as análises de desempenho computacional e uso da memória,

evidenciando o comportamento dessas matrizes diante de diferentes entradas. Com isso, foi possível comparar a análise experimental com a teórica feita previamente, em relação a complexidade de tempo dos algoritmos e a forma que eles acessam a memória.

A análise feita sobre a matriz acaba se tornando mais complexa, devido a sua estrutura em duas dimensões. No entanto, com a geração de gráficos e tabelas feitas durante as análises, é possível visualizar o comportamento da estrutura com diferentes tamanhos de entrada ao longo da execução do programa, tornando a análise mais simples.

8. Bibliografia

CHORMEN, Thomasb H. *et al*. **Algoritmos - Teoria e Prática**. 3ª edição.
Capítulo 3: Crescimento de funções. Editora Elsevier, 10 de abril de 2012.

APÊNDICE A - INSTRUÇÕES PARA COMPILAÇÃO E EXECUÇÃO

Antes de executar o programa em si, caso o usuário queira executar as análises junto,, é necessário a compilação do programa “analisamem”, responsável pela análise de localidade de memória e distância de pilha. Para tal, acessar o arquivo “analisamem” e executar:

1) make bin

Isso gerará o binário do programa, necessário para o funcionamento do programa principal.

Após isso, já é possível executar o programa principal, acessando “TP” e executando:

2) make all

Isso executará o programa e gerará todas as análises, disponíveis no diretório **out.**

Caso as análises não sejam necessárias, o programa em si pode ser executado com:

bin/matop **-t** **-p** ${output log} **-1** ${input matriz 1} **-2** ${input matriz 2} **-o** ${output da operação}

O parâmetro ***-t*** indica que a operação a ser realizada é a de transposição. Para operação de soma é o parâmetro **-s** e multiplicação **-m**. Os *inputs* e *outputs* representam o caminho para os arquivos de origem ou destino. Esse caminho pode ser relativo ou absoluto.