

# TRABALHO PRÁTICO 1:

## Grafos

Sandro Miccoli - 2009052409 - smiccoli@dcc.ufmg.br

<sup>1</sup>Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

17 de outubro de 2012

**Resumo.** *Esse relatório descreve como foi solucionado o problema proposto no Trabalho Prático 1, o qual será detalhado ao longo das próximas seções. Será descrito também a modelagem do problema e a solução proposta para tal. Finalmente será detalhado a análise de complexidade dos algoritmos, os testes utilizados para comprovar tais análises e uma breve conclusão do trabalho implementado.*

### 1. INTRODUÇÃO

A situação que deparamos neste trabalho é a seguinte: a distribuidora de produtos, Atlanticon, está precisando de uma consultoria para decidir em qual cidade deve instalar a nova filial para servir uma determinada região.

A empresa possui conhecimento das distâncias entre quaisquer duas cidades da região que sejam conectadas por algum trilha. Além disso, o trem, precariamente, realiza a entrega de apenas um produto por viagem, tornando necessário retornar para a filial para carregar um novo produto.

Existem 3 cenários distintos que a Atlanticon deseja saber qual a melhor opção de cidade para instalar a nova filial, e o objetivo do trabalho é solucionar esse problema.

Descreveremos aqui os três cenários:

**Cenário 1:** Escolher a cidade de modo a minimizar os custos de gasto em combustível. Considerando que cada cidade emite a mesma quantidade de pedidos em um mesmo intervalo de tempo.

**Cenário 2:** Escolher a cidade para também minimizar os custos de gasto em combustível. Porém, neste cenário cada cidade possui um volume diferente de pedidos, isso deve ser considerado para a escolha final.

**Cenário 3:** A Atlanticon deseja garantir que o produto seja entregue num prazo de  $X$  horas. A escolha da cidade se dá pelo menor valor possível de  $X$  que pudermos encontrar.

O restante deste relatório é organizado da seguinte forma. A Seção 2 descreve como foi feita a modelagem e manipulação dos grafos. A Seção 3 descreve rapidamente qual foi a solução proposta além do método utilizado para calcular as distâncias mínimas entre as cidades. A Seção 4 trata de detalhes específicos da implementação do trabalho: quais os arquivos utilizados; como é feita a compilação e execução; além de detalhar o formato dos arquivos de entrada e saída. A Seção 5 contém a avaliação experimental, quantificando o tempo de execução de cada operação com grafos de diversos tamanhos e formatos. A Seção 6 conclui o trabalho.

## 2. MODELAGEM

Inicialmente, para trabalhar com grafos, foi criada uma estrutura que contém a informação de quantos vértices o grafo possui, nesse caso, cidades, e qual o volume de pedidos de cada cidade:

```
typedef struct grafo {
    Matriz matrizAdj;
    int N; // Quantidade de cidades no grafo
    int *Vol; // Volume de pedidos de cada cidade
} Grafo;

typedef struct Matriz{
    int col, lin;
    int ** matriz;
} Matriz;
```

Já que o grafo que nos é passado como entrada é uma matriz de adjacências, e como já havíamos implementado uma estrutura de matriz no Trabalho Prático 0, reutilizamos o código neste trabalho.

A complexidade de espaço dessa estrutura pode ser considerada como  $O(n)$ , sendo  $n$  o tamanho bidimensional da matriz, ou seja, o produto entre a quantidade de linhas e colunas que ela possui.

Como foi detalhado na especificação, a matriz foi modelada para ser alocada e desalocada dinamicamente, através dos comandos *malloc* e *free*. Para confirmar que este processo de alocação dinâmica de memória estava ocorrendo como esperado, foi utilizado o comando *valgrind* para verificar qualquer tipo de vazamento de memória.

## 3. SOLUÇÃO PROPOSTA

O problema que deparamos é minimizar o custo de combustível dos trens da empresa Atlanticon. Como já modelamos nosso problema como um grafo, nada melhor que um algoritmo específico que nos traga a solução. A seguir uma definição de [Sedgewick] para o problema de caminho mínimo:

Dado um digrafo com custos não negativos nos arcos e um vértice  $s$ , encontrar uma arborescência de caminhos mínimos com raiz  $s$  no digrafo.

O algoritmo que utilizaremos foi inventado por Edsger Dijkstra [a pronúncia é algo entre "Dêcstra" e "Dêicstra"] e publicado em 1959. O algoritmo pode ser usado, em particular, para encontrar um caminho de custo mínimo de um dado vértice a outro [Sedgewick].

A solução proposta aqui foi realizar o algoritmo de Dijkstra para cada vértice do grafo, então descobrir o menor caminho dentre todos os possíveis para resolver nosso problema de distribuição de produtos pelas linhas de trem.

Para o **Cenário 1** utilizamos apenas as distâncias entre as cidades como peso nas aretas para calcular o caminho mínimo entre os vértices.

Já no **Cenário 2**, para cada cidade, calculamos o produto entre a distância das cidades com o volume de pedidos de cada uma. O menor produto resulta na nossa melhor escolha.

No **Cenário 3**, temos que minimizar o valor de  $X$ , para isso calculamos a maior distância percorrida para cada cidade. Assim garantimos a quantidade de horas mínima para o pedido chegar. Dentre essas maiores distâncias, escolhemos a menor delas, minimizando o valor de  $X$ .

### 3.1. Algoritmos

#### 3.1.1. Kronecker

O produto de Kronecker consiste em uma operação entre duas matrizes de dimensões arbitrárias que resulta em uma matriz de bloco. A ideia do algoritmo é que cada elemento da matriz **A** multiplique todos os elementos da matriz **B** gerando diversos blocos, que, eventualmente, irão gerar a matriz de blocos **C**.

A seguir uma definição do produto de Kronecker [?]:

Seja **A** uma matriz  $m \times n$  e **B** uma matriz  $p \times q$ , o produto de Kronecker  $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ , também conhecido como produto tensorial, é uma matriz  $(mp) \times (nq)$ , a qual elementos são definidos como:

$$c_{\alpha\beta} = a_{ij}b_{kl},$$

onde

$$\alpha = p(i - 1) + k$$

$$\beta = q(j - 1) + l.$$

Uma única diferença em relação à definição do produto de Kronecker de [?], foi a posição do resultado na matriz **C** do produto de dois valores das matrizes **A** e **B**. Como na nossa situação, o primeiro elemento se encontra na posição  $(0,0)$ , e não  $(1,1)$ , como esperado, então a posição do elemento na matriz **C** teve de ser alterada para o seguinte:

$$c_{\alpha\beta} = a_{ij}b_{kl},$$

onde

$$\alpha = (p * i + k) - 1$$

$$\beta = (q * j + l) - 1.$$

O algoritmo percorre as linhas e colunas da matriz **A** e da matriz **B** para realizar todas as operações. Então, podemos definir que sua complexidade terá um limite superior de  $O(n^4)$ .

## 4. IMPLEMENTAÇÃO

### 4.1. Código

#### 4.1.1. Arquivos .c

- **tp0.c** Arquivo principal do programa, lê todas as instâncias de matrizes do arquivo de entrada, realiza o produto de Kronecker e insere cada resultado no arquivo de saída.
- **matriz.c** Contém todas as funções de manipulação, leitura e escrita de matrizes.
- **kronecker.c** Contém a função que efetivamente calcula o produto de Kronecker.
- **arquivos.c** Um tipo abstrado de dados de manipulação de arquivos, contendo funções de abertura, leitura, escrita e fechamento.

#### 4.1.2. Arquivos .h

- **matriz.h** Biblioteca que define as funções relativas à matrizes, além de definir a estrutura que é utilizada a todo momento.
- **kronecker.h** Biblioteca que define a função que calcula o produto de Kronecker.
- **arquivos.h** Definição das funções utilizadas para ler, escrever e fechar corretamente um arquivo.

### 4.2. Compilação

O programa deve ser compilado através do compilador GCC através de um makefile ou do seguinte comando:

```
gcc -Wall -Lsrc src/tp0.c src/matriz.c src/kronecker.c src/arquivos.c -o tp0
```

### 4.3. Execução

A execução do programa tem como parâmetros:

- Um arquivo de entrada contendo várias instâncias de matrizes.
- Um arquivo de saída que irá receber o resultado do produto de Kronecker de cada instância de matriz.

O comando para a execução do programa é da forma:

```
./tp0 <arquivo_de_entrada> <arquivo_de_saída>
```

#### 4.3.1. Formato da entrada

A primeira linha do arquivo de entrada contém o valor  $k$  de instâncias (pares de matrizes) que o arquivo contém. A próxima linha contém as dimensões  $m$  e  $n$  da matriz  $A_1$ . As próximas  $m$  linhas contém os elementos de cada linha de  $A_1$  separados por um espaço. Em seguida, as dimensões e os elementos da matriz  $B_1$  são especificadas da mesma forma. E assim sucessivamente nas instâncias seguintes.

A seguir dois pares de matrizes de exemplo:

$$A_1 = \begin{bmatrix} 3 & 7 & 4 \end{bmatrix} \quad B_1 = \begin{bmatrix} 0 \\ 7 \\ 6 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 6 \end{bmatrix} \quad B_2 = \begin{bmatrix} 6 \\ 6 \end{bmatrix}$$

Esse arquivo de entrada tem a seguinte configuração:

```
2
1 3
3 7 4
3 1
0
7
6
1 1
6
2 1
6
6
```

#### 4.3.2. Formato da saída

O arquivo de saída tem a mesma configuração, sendo a primeira linha o valor  $k$ , seguido das  $k$  matrizes.

A seguir o resultado do produto de Kronecker das instâncias de matrizes definidas anteriormente:

$$C_1 = \begin{bmatrix} 0 & 0 & 0 \\ 21 & 49 & 28 \\ 18 & 42 & 24 \end{bmatrix} \quad C_2 = \begin{bmatrix} 36 \\ 36 \end{bmatrix}$$

Esse arquivo de saída tem a seguinte configuração:

```
2
3 3
0 0 0
21 49 28
18 42 24
2 1
36
36
```

## 5. AVALIAÇÃO EXPERIMENTAL

Foram gerados arquivos de vários tipos para testar, depois foi usado o comando *time* para calcular o tempo de execução de cada configuração de teste diferente. Nas próximas duas subseções vamos detalhar os scripts que foram criados para gerar esses arquivos de testes, além de demonstrar os resultados de cada teste.

## 5.1. Scripts

Primeiro, foi criado um script, em Python, para gerar arquivos de entrada para o trabalho. Esse programa recebe como parâmetro o número de instâncias a ser criado e o número máximo de linhas e colunas que essas matrizes terão.

Posteriormente foi feito outro script para gerar um lote de arquivos de entrada e já armazenar o tempo de processamento de cada um.

Primeiro, o script que gera o arquivo de entrada:

```
import random, sys

k = int(sys.argv[1]) # pares de matrizes
maxX = int(sys.argv[2]) # tamanho máximo de linhas da matriz
maxY = int(sys.argv[3]) # tamanho máximo de colunas da matriz

print k

for i in range(k)*2: # Multiplica por 2 pois são pares de matrizes
    m = maxX
    n = maxY
    print m,n
    for j in range(m):
        for l in range(n):
            print random.randint(0,100),
        print ''
```

Segundo, o script que utiliza o primeiro para fazer um *batch* de testes:

```
import sys, os, random

ent = 'entrada/'
sai = 'saida/'
tempo = 'tempo/'
script = './testes/entrada.py '
arqAppend = ' >> '
arqNew = ' > '

for i in range(100):
    i+=1
    i*=5
    #teste1 = script+' 1 '+str(i)+' '+str(i)+' ' # Matrizes quadradas
    #teste2 = script+' 1 10 '+str(i) # linhas fixas em 10
    #teste3 = script+' 1 '+str(i)+' 10' # colunas fixas em 10
    teste = script+' '+str(i)+' 50 50' # linhas e colunas fixas em 50 +
                                     nº de instâncias variando

    arqEnt= 'ent'+str(i)+'.txt'
    arqSai= 'sai'+str(i)+'.txt'
    arqTmp= 'tmp'+str(i)+'.txt'
    os.system(teste+arqNew+ent+arqEnt)
```

```
os.system('/usr/bin/./time -o '+tempo+arqTmp+' ./tp0 '+arqEnt+' '+  
arqSai)
```

## 5.2. Resultado

O primeiro teste realizado foi o seguinte: foi fixado o número de linhas e colunas das matrizes para 50. Ou seja, cada matriz tinha 2.500 elementos, e as matrizes resultado tinham 6.250.000 elementos cada uma.

Na Figura 1, podemos ver que o tempo de processamento cresceu de uma maneira esperada. Como a complexidade do algoritmo é definida basicamente pelo tamanho das matrizes, então uma variação do número de instâncias resulta em um crescimento praticamente constante no tempo de processamento do programa.

Na Figura 2 foi feito um teste com matrizes quadradas. Tanto a matriz **A**, quanto a matriz **B** variaram o tamanho de  $1 \times 1$  para  $100 \times 100$ . Ou seja, num momento tinham apenas um elemento e no último tinham 10.000 elementos. A matriz **C**, no final desse teste, possuía 100.000.000 elementos. O tempo de processamento aqui já cresceu de maneira exponencial, como era de se esperar por causa da complexidade exponencial do produto de Kronecker.

Os últimos dois testes foram configurados da seguinte maneira: fixamos as colunas e variamos as linhas e vice-versa.

No primeiro, Figura 3, fixamos as colunas das matrizes em 10 e variamos as linhas até 500, ou seja, a matriz **C** começou com tamanho  $1 \times 100$  (100 elementos) e terminou com um tamanho de  $250.000 \times 100$  (25.000.000 elementos); no segundo, Figura 4, fixamos as linhas das matrizes em 10 e variamos as colunas até 500. O tempo de processamento desses testes se encontra nas figuras abaixo:

## 6. CONCLUSÃO

O problema de calcular o produto de Kronecker das matriz foi solucionado sem muitas complicações. As matrizes foram alocadas dinamicamente com sucesso, de acordo com o *Valgrind*. Além disso, um padrão de código bem modularizado foi seguido, para que os módulos possam ser reutilizados futuramente.

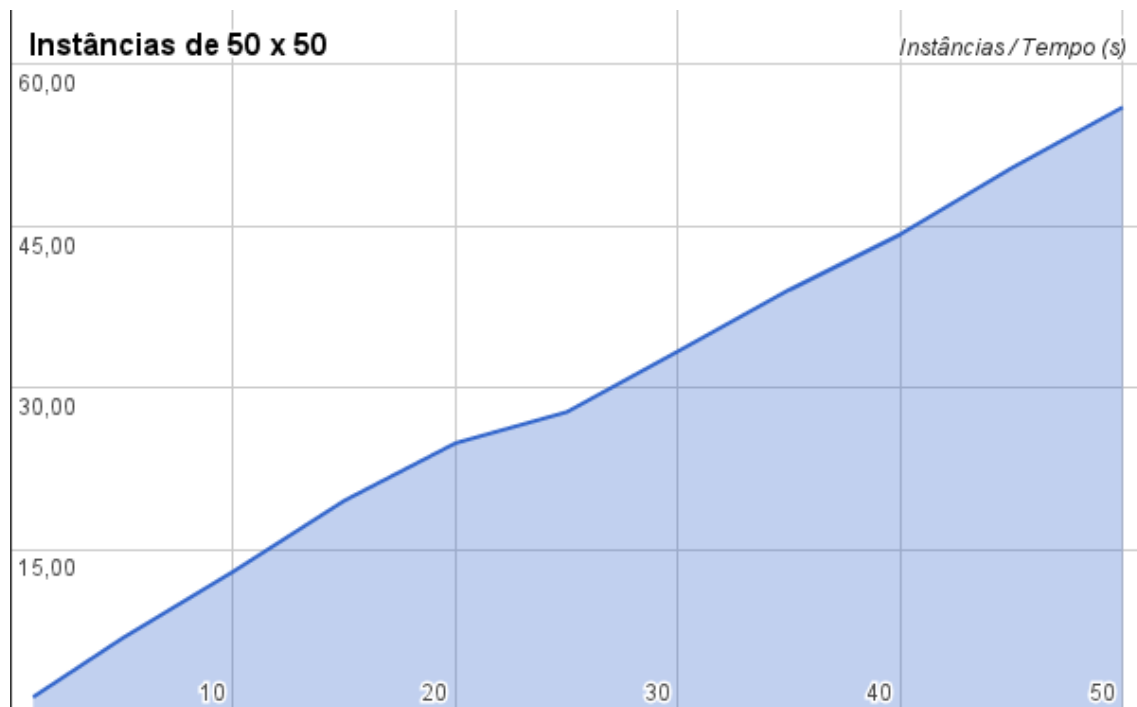
O código também já foi construído de uma maneira em que, caso futuramente seja necessário paralelizá-lo, poucas mudanças no código precisarão ser feitas para tal.

Os testes foram bem sucedidos pois o tempo de processamento do algoritmo em diversas situações ocorreu como esperado.

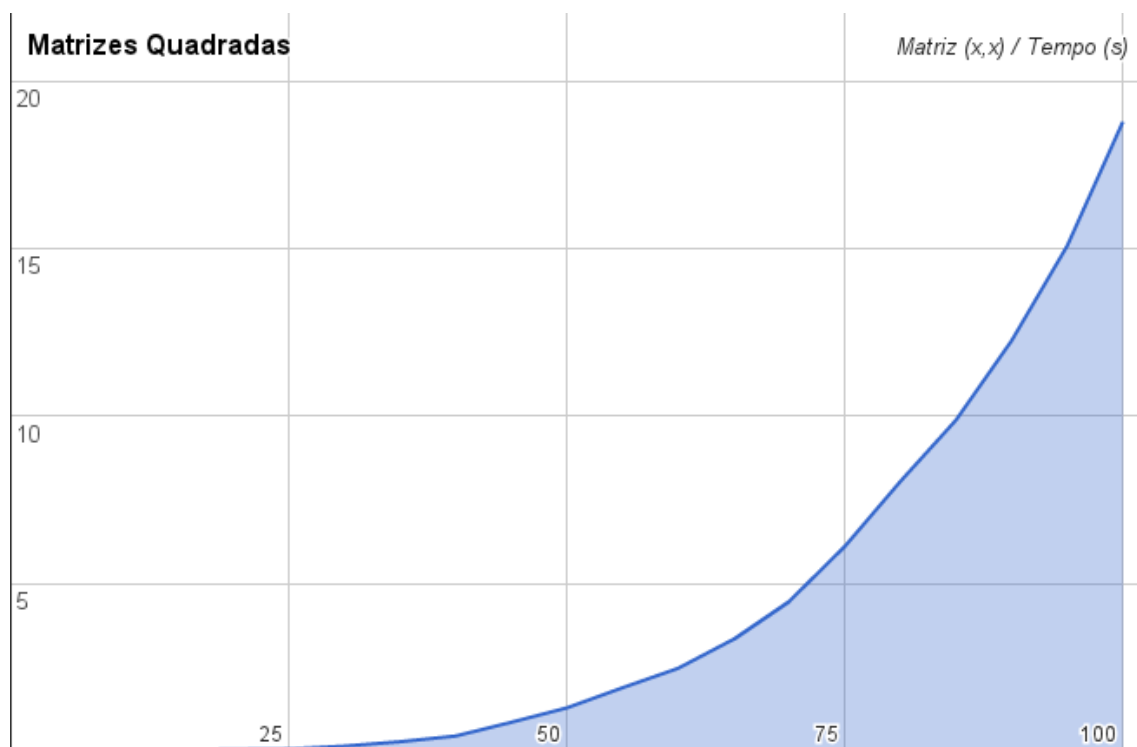
As primitivas básicas da linguagem C, como alocação dinâmica de memória e manipulação de arquivos, foram bem exploradas no trabalho prático. Além disso, procurei construir uma documentação bem detalhada sobre toda a estrutura e funcionamento do programa implementado.

## Referências

Sedgewick, R. *Algorithms in C: Graph Algorithms*.

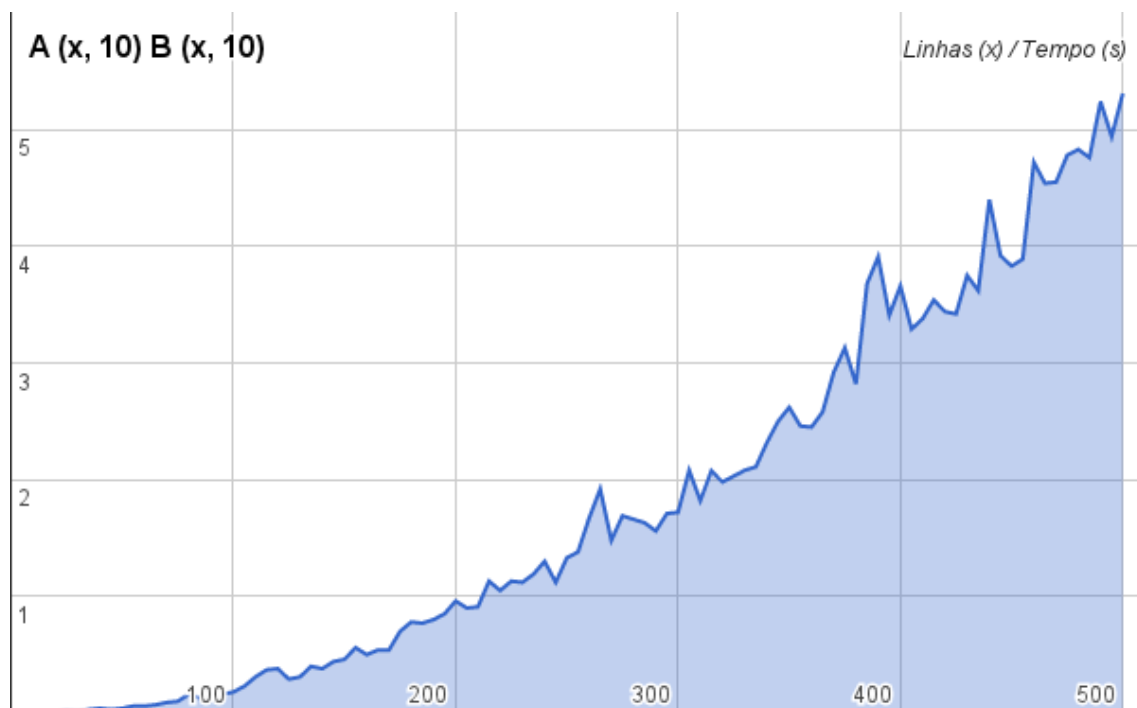


**Figura 1. A(50,50) B(50,50)**

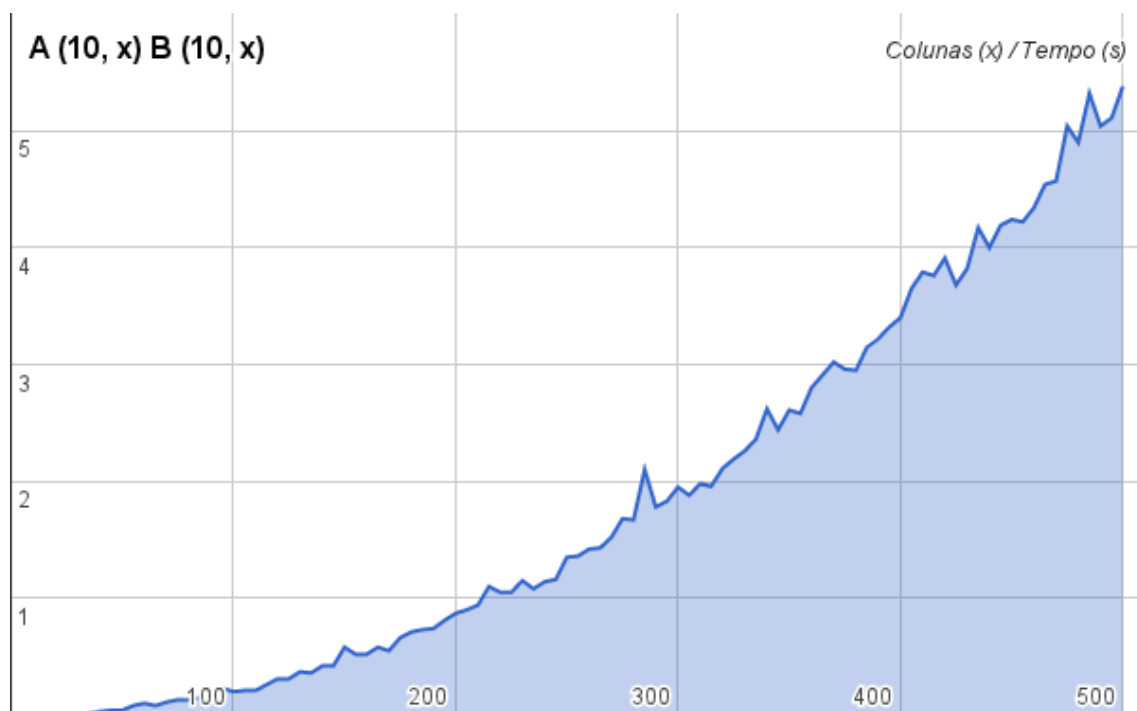


**Figura 2. Matrizes Quadradas A(x,x) B(x,x)**





**Figura 3. Linhas - A(x,10) B(x,10)**



**Figura 4. Colunas - A(10,x) B(10,x)**