Universidade Federal de Ouro Preto - UFOP

Instituto de Ciências Exatas e Biológicas - ICEB

Departamento de Computação - DECOM

Ciência da Computação

Trabalho Prático I

PCC104 - Projeto e Análise de Algoritmos

Fernando dos Santos Alves Fernandes

Professor: Pedro Henrique Lopes Silva

Ouro Preto
30 de abril de 2024

Sumário

1 Introdução **1**
1.1 O cálculo da multiplicação recursiva de matrizes 1
1.2 O algoritmo de Dijkstra 1
1.3 Visão geral do ambiente de desenvolvimento 1
1.3.1 Ambiente de desenvolvimento e ferramentas utilizadas 1
1.3.2 Instruções de compilação e execução 2

2 Implementação **2**
2.1 Implementação do cálculo da multiplicação de duas matrizes 2
2.2 Implementação do algoritmo de Dijkstra 9

3 Estudo de Complexidade **13**
3.1 Análise de complexidade do algoritmo de Strassen 13
3.2 Análise de complexidade do algoritmo de Dijkstra 14

4 Testes **15**
4.1 Testes com as instâncias de matrizes quadradas 15
4.2 Testes com as instâncias de caminhos mínimos 16

5 Análise **17**
5.1 Análise dos resultados para a multiplicação de matrizes 17
5.2 Análise dos resultados para o algoritmo de Dijkstra 18

6 Considerações Finais **18**

Lista de Figuras

1 Análise de complexidade do algoritmo de Dijkstra implementado. 15

Lista de Tabelas

1 Exemplo de matriz resultante do produto das matrizes de entrada. 15
2 Exemplo de uma instância de entrada e a saída correspondente, para o problema de caminhos mínimos. 16
3 Tempos de execução multiplicação de matrizes para diferentes instâncias. 17
4 Tempos de execução Dijkstra para diferentes instâncias. 18

Lista de Códigos Fonte

1 Pseudo-código - Produto de duas matrizes quadradas de ordem n pelo método tradicional. 2
2 Pseudo-código - Versão recursiva simples do produto de matrizes, usando divisão-e-conquista. 3
3 Pseudo-código - Algoritmo de Strassen para a multiplicação de matrizes. 3
4 Implementação da Tarefa 1 - Produto de duas matrizes quadradas de ordem n utilizando o paradigma **divisão-e-conquista**. 4
5 Implementação da Tarefa 2 - Algoritmo de Dijkstra para o problema de caminhos mínimos. 9

1 Introdução

Neste primeiro trabalho prático da disciplina de Projeto e Análise de Algoritmos, dois importantes problemas computacionais foram abordados: o ***cálculo da múltiplicação de duas matrizes quadradas*** e ***a busca de caminhos mínimos em grafos***. Os detalhes das especificações desses dois problemas são apresentados a seguir.

1.1 O cálculo da multiplicação recursiva de matrizes

Sejam A e B duas matrizes quadradas de inteiros de ordem n x n compostas de elementos a_{ij} e b_{ij}, respectivamente, com i, j = 1, 2, 3, ..., n. O produto C = A.B, pode ser representado como a matriz resultante, cujos elementos c_{ij} de C, para i, j = 1, 2, 3, ..., n, podem ser obtidos pela equação:

$$c_{ij} = \sum_{k=1}^{n} a_i j . b_i j$$

No cálculo tradicional da matriz produto C, deve-se calcular n^2 elementos, cada um com um custo de $\Theta(n)$ somas e produtos, levando a um custo total de $\Theta(n3)$. Utilizando o paradigma de **divisão-e-conquista** é possível realizar o cálculo do produto de duas matrizes quadradas em $O(n^3)$.

A ***Tarefa 1*** consistiu em implementar e apresentar um algoritmo com essa complexidade. As entradas são passadas pelo terminal, por meio de arquivos, em que a primeira linha corresponde à dimensão das matrizes, n, seguinda pelas linhas correspondentes aos valores das duas matrizes de entrada. A saída corresponde à matriz resultante da multiplicação [1].

1.2 O algoritmo de Dijkstra

O algoritmo de Dijkstra resolve o problema de encontrar caminhos mínimos de uma única fonte, num grafo ponderado *G = (V, E)*, em que V é o conjunto de vértices e E é o conjunto de arestas do grafo G, e os pesos associados às arestas têm valores não negativos.

O algoritmo de Dijkstra mantém um conjunto de vértices cujos pesos do caminho mínimo final da fonte já foram determinados. Repetidamente, o algoritmo seleciona o vértice com o menor caminho mínimo estimado até o momento, adiciona-o ao conjunto de vértices já determinados e atualiza os pesos dos caminhos já determinados, quando necessário. Além disso, o algoritmo armazena, para cada vértice já visitado, uma referência para o vértice predecessor que levou ao menor valor de custo do caminho até o momento. Ao final, é possível determinar o custo e reconstruir o caminho mínimo do vértice de origem até cada um dos outros vértices do grafo.

A ***Tarefa 2*** consistiu em implementar esse algoritmo e utilizar instâncias de uma competição de caminhos mínimos do DECOM[1] para realizar os testes. Os formatos, tanto das entradas, quanto das saídas, são fornecidos pelo próprio site da competição.

1.3 Visão geral do ambiente de desenvolvimento

A seguir, uma visão geral do ambiente de desenvolvimento e instruções de compilação e uso das soluções implementadas.

1.3.1 Ambiente de desenvolvimento e ferramentas utilizadas

Os algoritmos desenvolvidos para a solução das duas tarefas foram implementados em linguagem ***C++*** e compilados usando o ***gcc 6.3.0*** do conjunto de ferramentas MinGW[2]. Os programas gerados foram executados em ambiente ***Windows 10 Pro - 64 bits***, numa máquina ***Intel(R) Core(TM) i5-7500 CPU @ 3.40 GHz 3.41 GHz***, com ***16 GB RAM***.

[1] http://www.decom.ufop.br/haroldo/1acompcm/index.html
[2] https://sourceforge.net/projects/mingw/files/MinGW/Base/gcc/Version6/gcc-6.3.0/

1.3.2 Instruções de compilação e execução

Para a compilação do código associado à Tarefa 1, basta digitar, no terminal:

Compilando código para cálculo da multiplicação de matrizes

```
gcc task1-square-matrix-multiply.cpp -o task1-square-matrix-multiply
```

Para a execução do programa basta digitar:

```
./task1-square-matrix-multiply.exe <arquivo.dat> <algoritmo [m: square-matrix-multiply | s: strassen algorithm]>
```

Para a compilação do código associado à Tarefa 2, basta digitar, no terminal:

Compilando código para o algoritmo de Dijkstra

```
gcc task2-dijkstra-algorithm-shortint.cpp -o task2-dijkstra-algorithm-shortint
```

Para a execução do programa basta digitar:

```
./task2-dijkstra-algorithm-shortint.exe <arquivoProblema> <nrExecuções> <flagDepuração>
```

2 Implementação

Nesta seção serão apresentadas as soluções implementadas para o cálculo da multiplicação de matrizes e para o algoritmo de Dijkstra. Para cada solução, descreve-se o funcionamento das principais funções e procedimentos utilizados, bem como detalhes de implementação.

2.1 Implementação do cálculo da multiplicação de duas matrizes

O cálculo tradicional do produto de duas matrizes quadradas tem complexidade de tempo da ordem de $\Theta(n^3)$, o que pode ser observado, intuitivamente, no pseudo-código em 1.

```
SQUARE-MATRIX-MULTIPLY(A, B)
1 n = A.rows
2 let C be a new nxn matrix
3 for i = 1 to n
4     for j = 1 to n
5         cij = 0
6         for k = 1 to n
7             cij = cij + aik . bkj
8 return C
```

Código 1: Pseudo-código - Produto de duas matrizes quadradas de ordem n pelo método tradicional.

Utilizando o paradigma ***divisão-e-conquista*** é possível implementar um versão recursiva da multiplicação de duas matrizes quadradas, cujas dimensões sejam uma potência de 2. O pseudo-código dessa versão recursiva pode ser visto em 2.

Sejam duas matrizes quadradas, X e Y, onde $A, B, ..., H$ são todas matrizes $\frac{n}{2}x\frac{n}{2}$:

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

$$Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

O produto de X e Y, pode ser obtido pela expressão [5]:

$$X.Y = \begin{pmatrix} A.B + B.G & A.F + B.H \\ C.E + D.G & C.F + D.H \end{pmatrix}$$

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(X, Y)
1 n = X:rows
2 let C be a new nxn matrix
3 if n == 1
4     c11 = a11 . b11
5 else partition X, Y, and C
6     C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(X11, Y11) + SQUARE-MATRIX-MULTIPLY-RECURSIVE(X12, Y21)
7     C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(X11, Y12) + SQUARE-MATRIX-MULTIPLY-RECURSIVE(X12, Y22)
8     C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(X21, Y11) + SQUARE-MATRIX-MULTIPLY-RECURSIVE(X22, Y21)
9     C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(X21, Y12) + SQUARE-MATRIX-MULTIPLY-RECURSIVE(X22, Y22)
10 return C
```

Código 2: Pseudo-código - Versão recursiva simples do produto de matrizes, usando divisão-e-conquista.

Analisando a equação de recorrência para o custo de tempo de execução desse algoritmo recursivo, observa-se que a ordem de complexidade não é melhor do que a da versão iterativa ($\Theta(n^3)$). O método de Strassen ou algoritmo de Strassen [3] propõe uma solução que reduz o número de operações de multiplicação de matrizes no processo e garante uma solução com complexidade na ordem de $\Theta(n^{log_2 7}) \in O(n^3)$. Detalhes sobre as etapas do método de Strassen e cálculos algébricos associados podem ser encontrados em [1].

```
STRASSEN (X,Y, n)
1 let C be a new nxn matrix
2 if n == 1
3     return X . Y
4 else partition X, Y, and C
5     P1 = STRASSEN(A, F - H, n/2)
6     P2 = STRASSEN(A + B,H, n/2)
7     P3 = STRASSEN(C + D,E, n/2)
8     P4 = STRASSEN(D, G - E, n/2)
9     P5 = STRASSEN(A + D,E + H, n/2)
10    P6 = STRASSEN(B - D, G + H, n/2)
11    P7 = STRASSEN(A - C, E + F, n/2)
12    C11 = P5 + P4 - P2 + P6
13    C12 = P1 + P2
14    C21 = P3 + P4
15    C22 = P5 + P1 - P3 - P7
16 return C
```

Código 3: Pseudo-código - Algoritmo de Strassen para a multiplicação de matrizes.

Na solução completa [4], as duas versões recursivas foram implementadas e testadas para multiplicação de matrizes. Uma matriz é representada pela classe $Matrix$, cujos elementos ficam armazenados em $matrizElements$. Para auxiliar as operações, foram implementados métodos para adição, subtração, leitura dos elementos de uma matriz e impressão (***addMatrices***, ***subtractMatrices***, ***setElement***, ***printMatrix***), além dos métodos recursivos específicos para a multiplicação de duas matrizes (***squareMatrixMultiply*** e ***strassen***).

A princípio, pela forma como os algoritmos recursivos dividem o problema em instâncias menores ($\frac{n}{2}$), as matrizes de entrada precisariam ter sua ordem n como uma potência de 2. Uma possível solução para esse problema é atribuir zeros à matriz original, de forma que ela passe a ter o número de linhas e colunas igual à primeira potência de 2 maior do que n. Essa estratégia foi implementada e testada neste trabalho.

```cpp
#include <iostream>
#include <vector>
#include <fstream>
#include <chrono>

using namespace std;

class Matrix {
    public:
        Matrix(int size);
        Matrix(vector<vector<int>> matrixElements);
        //~Matrix();
        void setElement(int i, int j, int value);
        void printMatrix();
        void printMatrix(int n);
        void printMatrix(const vector<vector<int>>& matrix);
        vector<vector<int>> squareMatrixMultiply(Matrix B);
        vector<vector<int>> squareMatrixMultiply(const vector<vector<int>>& A, const vector<vector<int>>& B);
        vector<vector<int>> strassen(Matrix B);
        vector<vector<int>> strassen(const vector<vector<int>>& A, const vector<vector<int>>& B);
    private:
        vector<vector<int>> elements;
        vector<vector<int>> *ptE;
        vector<vector<int>> addMatrices(Matrix B);
        vector<vector<int>> addMatrices(const vector<vector<int>>& A, const vector<vector<int>>& B);
        vector<vector<int>> subtractMatrices(Matrix B);
        vector<vector<int>> subtractMatrices(const vector<vector<int>>& A, const vector<vector<int>>& B);

};

// Metodo construtor.
Matrix::Matrix(int size){
    // Se a ordem das matrizes nao for uma potencia de 2, preencha com zeros
    int nextPowerOfTwo = 1;
    while (nextPowerOfTwo < size) {
        nextPowerOfTwo *= 2;
    }

    ptE = new vector<vector<int>>(nextPowerOfTwo, vector<int>(nextPowerOfTwo));
    elements = *ptE;
}

// Metodo construtor.
Matrix::Matrix(vector<vector<int>> matrixElements){
    elements = matrixElements;
}

void Matrix::setElement(int i, int j, int value){
    elements[i][j] = value;
}

// Metodo para imprimir uma matriz.
void Matrix::printMatrix() {
    for (const auto& row : elements) {
        for (int val : row) {
            cout << val << " ";
        }
```

```cpp
        cout << endl;
    }
}

// Metodo para imprimir uma matriz.
void Matrix::printMatrix(int n) {
    for (int i = 0; i < n; ++i){
        for (int j = 0; j < n; ++j)
            cout << elements[i][j] << " ";
        cout << endl;
    }
}

// Metodo para adicionar duas matrizes.
vector<vector<int>> Matrix::addMatrices(Matrix B) {
    int n = elements.size();
    vector<vector<int>> C(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            C[i][j] = elements[i][j] + B.elements[i][j];
    return C;
}

// Metodo para subtrair duas matrizes.
vector<vector<int>> Matrix::subtractMatrices(Matrix B) {
    int n = elements.size();
    vector<vector<int>> C(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            C[i][j] = elements[i][j] - B.elements[i][j];
    return C;
}

// Metodo para multiplicacao de matrizes usando Dividir para Conquistar.
vector<vector<int>> Matrix::squareMatrixMultiply(Matrix B) {
    int n = elements.size();
    vector<vector<int>> C(n, vector<int>(n, 0));
    if (n == 1) {
        C[0][0] = elements[0][0] * B.elements[0][0];
    } else {
        // Dividindo as matrizes em submatrizes
        int newSize = n / 2;

        Matrix mA11(newSize);
        Matrix mA12(newSize);
        Matrix mA21(newSize);
        Matrix mA22(newSize);
        Matrix mB11(newSize);
        Matrix mB12(newSize);
        Matrix mB21(newSize);
        Matrix mB22(newSize);

        for (int i = 0; i < newSize; ++i) {
            for (int j = 0; j < newSize; ++j) {
                mA11.setElement(i, j, elements[i][j]);
                mA12.setElement(i, j, elements[i][j + newSize]);
                mA21.setElement(i, j, elements[i + newSize][j]);
                mA22.setElement(i, j, elements[i + newSize][j + newSize]);

                mB11.setElement(i, j, B.elements[i][j]);
                mB12.setElement(i, j, B.elements[i][j + newSize]);
                mB21.setElement(i, j, B.elements[i + newSize][j]);
```

```
                mB22.setElement(i, j, B.elements[i + newSize][j + newSize]);
            }
        }
        // Calculando os elementos da matriz C
        vector<vector<int>> C11 = Matrix(mA11.squareMatrixMultiply(mB11)).
            addMatrices(Matrix(mA12.squareMatrixMultiply(mB21)));
        vector<vector<int>> C12 = Matrix(mA11.squareMatrixMultiply(mB12)).
            addMatrices(Matrix(mA12.squareMatrixMultiply(mB22)));
        vector<vector<int>> C21 = Matrix(mA21.squareMatrixMultiply(mB11)).
            addMatrices(Matrix(mA22.squareMatrixMultiply(mB21)));
        vector<vector<int>> C22 = Matrix(mA21.squareMatrixMultiply(mB12)).
            addMatrices(Matrix(mA22.squareMatrixMultiply(mB22)));

        // Combinando as submatrizes para obter a matriz C
        for (int i = 0; i < newSize; ++i) {
            for (int j = 0; j < newSize; ++j) {
                C[i][j] = C11[i][j];
                C[i][j + newSize] = C12[i][j];
                C[i + newSize][j] = C21[i][j];
                C[i + newSize][j + newSize] = C22[i][j];
            }
        }
    }
    return C;
}

// Metodo para multiplicar matrizes usando o algoritmo de Strassen.
vector<vector<int>> Matrix::strassen(Matrix B) {
    int n = elements.size();
    vector<vector<int>> C(n, vector<int>(n, 0));
    if (n == 1) {
        C[0][0] = elements[0][0] * B.elements[0][0];
    } else {
        // Dividindo as matrizes em submatrizes
        int newSize = n / 2;

        Matrix mA11(newSize);
        Matrix mA12(newSize);
        Matrix mA21(newSize);
        Matrix mA22(newSize);
        Matrix mB11(newSize);
        Matrix mB12(newSize);
        Matrix mB21(newSize);
        Matrix mB22(newSize);

        for (int i = 0; i < newSize; ++i) {
            for (int j = 0; j < newSize; ++j) {
                mA11.setElement(i, j, elements[i][j]);
                mA12.setElement(i, j, elements[i][j + newSize]);
                mA21.setElement(i, j, elements[i + newSize][j]);
                mA22.setElement(i, j, elements[i + newSize][j + newSize]);

                mB11.setElement(i, j, B.elements[i][j]);
                mB12.setElement(i, j, B.elements[i][j + newSize]);
                mB21.setElement(i, j, B.elements[i + newSize][j]);
                mB22.setElement(i, j, B.elements[i + newSize][j + newSize]);
            }
        }

        // Calculando as submatrizes
        vector<vector<int>> P1 = mA11.strassen(Matrix(mB12.subtractMatrices(
            mB22)));
```

```
        vector<vector<int>> P2 = Matrix(mA11.addMatrices(mA12)).strassen(mB22);
        vector<vector<int>> P3 = Matrix(mA21.addMatrices(mA22)).strassen(mB11);
        vector<vector<int>> P4 = mA22.strassen(Matrix(mB21.subtractMatrices(mB11)));
        vector<vector<int>> P5 = Matrix(mA11.addMatrices(mA22)).strassen(Matrix(mB11.addMatrices(mB22)));
        vector<vector<int>> P6 = Matrix(mA12.subtractMatrices(mA22)).strassen(Matrix(mB21.addMatrices(mB22)));
        vector<vector<int>> P7 = Matrix(mA11.subtractMatrices(mA21)).strassen(Matrix(mB11.addMatrices(mB12)));

        // Calculando os elementos da matriz C
        vector<vector<int>> C11 = addMatrices(subtractMatrices(addMatrices(P5, P4), P2), P6);
        vector<vector<int>> C12 = addMatrices(P1, P2);
        vector<vector<int>> C21 = addMatrices(P3, P4);
        vector<vector<int>> C22 = subtractMatrices(subtractMatrices(addMatrices(P5, P1), P3), P7);

        // Combinando as submatrizes para obter a matriz C
        for (int i = 0; i < newSize; ++i) {
            for (int j = 0; j < newSize; ++j) {
                C[i][j] = C11[i][j];
                C[i][j + newSize] = C12[i][j];
                C[i + newSize][j] = C21[i][j];
                C[i + newSize][j + newSize] = C22[i][j];
            }
        }
    }
    return C;
}

// Funcao principal.
int main(int argc, char** argv){
    if (argc < 3){
        cout << "Execute: file_to_matrix.exe <arquivo.dat> <algoritmo [m: square-matrix-multiply | s: strassen algorithm]>" << endl;
        return -1;
    }
    //else
        //cout << "Arquivo de entrada: " << argv[1] << endl;

    ifstream inputFile(argv[1]);
    if (!inputFile.is_open()) {
        cerr << "Erro ao abrir o arquivo." << endl;
        return 1;
    }

    char algorithm = *argv[2];
    //cout << algorithm << endl;

    int n, value;
    inputFile >> n;

    int isPowerOf2 = n && !(n & (n - 1)); // Verifica se n e potencia de 2.
    //cout << isPowerOf2 << endl;

    Matrix A(n);
    Matrix B(n);

```

```cpp
    // Leitura das matrizes A e B do arquivo
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j){
            inputFile >> value;
            A.setElement(i, j, value);
        }

    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j){
            inputFile >> value;
            B.setElement(i, j, value);
        }

    inputFile.close();

    if (algorithm == 'm') {
        // Multiplicacao das matrizes usando o algoritmo Square Matrix Multiply.
        // Inicio medicao de tempo
        auto begin = chrono::high_resolution_clock::now();
        Matrix C(A.squareMatrixMultiply(B));
        // Fim medicao de tempo
        auto end = chrono::high_resolution_clock::now();
        auto elapsed = chrono::duration_cast<chrono::nanoseconds>(end - begin);
        cout << "\nTempo de execucao [Square Matrix Multiply]: " << elapsed.count() * 1e-9 << endl;
        // Impressao da matriz resultante
        //cout << "\nMatrix C:" << endl;
        if (isPowerOf2)
            C.printMatrix();
        else
            C.printMatrix(n);
    } else {
        // Multiplicacao das matrizes usando o algoritmo de Strassen.
        // Inicio medicao de tempo
        auto begin = chrono::high_resolution_clock::now();
        Matrix C(A.strassen(B));
        // Fim medicao de tempo
        auto end = chrono::high_resolution_clock::now();
        auto elapsed = chrono::duration_cast<chrono::nanoseconds>(end - begin);
        cout << "\nTempo de execucao [Algoritmo de Strassen]: " << elapsed.count() * 1e-9 << endl;
        // Impressao da matriz resultante
        //cout << "\nMatrix C:" << endl;
        if (isPowerOf2)
            C.printMatrix();
        else
            C.printMatrix(n);
    }
    return 0;
}
```

Código 4: Implementação da Tarefa 1 - Produto de duas matrizes quadradas de ordem n utilizando o paradigma **divisão-e-conquista**.

Na linha de comando do terminal, conforme descrito na seção 1.3.2, o usuário informa o nome do arquivo de entrada com os dados das duas matrizes para as quais quer realizar a multiplicação e pode escolher qual o algoritmo de ***divisão-e-conquista*** deseja usar, o ***squareMatrixMultiply*** ou o ***strassen***. Ambos os métodos são quase literalmente uma tradução ou uma transcrição dos pseudo-códigos 2 e 3 para a linguagem de programação *C++*.

Como já mencionado, a solução verifica se a ordem das matrizes é uma potência de 2 ou não e, caso não seja, preenche automaticamente com zeros as matrizes de entrada, de forma que elas passem a ter número de linhas e número de colunas igual à primeira potência de 2 maior do n, antes de realizar a multiplicação.

Para que a matriz resultante seja adequadamente apresentada na saída, um método de impressão específico foi implementado (***printMatrix(n)***), em que a ordem original das matrizes é passada e apenas os elementos nos limites dessa ordem sejam considerados elementos do produto. Se a ordem é uma potência de 2, o método padrão de impressão (***printMatrix()***) é utilizado.

2.2 Implementação do algoritmo de Dijkstra

Na 1ª Competição de Caminhos Mínimos do DECOM, os participantes tinham que implementar o algoritmo de Dijkstra para computação de caminhos mínimos de uma fonte para todos os outros vértices, e o algoritmo de Floyd-Warshall para computação dos caminhos mínimos de todos os vértices para todos os vértices do grafo. Neste trabalho prático, a *Tarefa 2* consiste em implementar apenas o algoritmo de Dijkstra para caminhos mínimos e utilizar as instâncias dessa competição para testar o código implementado.

Ainda, segundo as regras da competição, o vértice fonte para o qual os caminhos mínimos são computados deve ser sorteado a cada execução do programa. Além disso, na linha de comando, o usuário deve passar o número de execuções desejada e uma *flag* de depuração, indicando se o programa deve imprimir em arquivo os resultados. Quando a *flag* de depuração é igual a 0, o programa executa o algoritmo pelo número de execuções indicado pelo usuário, escolhendo, em cada execução, um vértice fonte diferente. Se a *flag* de depuração é igual a 1, o programa deve executar o algoritmo de caminhos mínimos uma única vez, sempre tendo o vértice 1 como o vértice fonte e o resultado deve ser gravado em arquivo. O formato das instâncias de entrada e o formato dos arquivos de saída são apresentado na seção 4.2.

O código 5 apresenta a implementação do algoritmo de Dijkstra para o problema de caminhos mínimos em grafos desenvolvido neste trabalho. Um grafo é representado pela classe *Graph*, que possui como atributos o número de vértices e uma lista de adjacências como a representação para as arestas do grafo.

Um objeto da classe *Graph* também apresenta métodos para adicionar uma aresta ao grafo (*addEdge*), imprimir o grafo (*printGraph*) e reconstruir o caminho de um vértice fonte até um vértice de destino (*reconstructPath*), além do método para o cálculo dos caminhos mínimos (*dijkstra*).

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <string>
#include <limits>
#include <queue>
#include <algorithm>
#include <chrono>

using namespace std;

// Estrutura para representar uma aresta
struct Edge {
    unsigned int to;
    unsigned short int weight;
    Edge(unsigned int t, unsigned short int w) : to(t), weight(w) {}
};

// Estrutura para representar um grafo
class Graph {
private:
    unsigned int numVertices;
    vector<vector<Edge>> adjacencyList;

```

```cpp
public:
    Graph(unsigned int n) : numVertices(n), adjacencyList(n) {}

    // Adiciona uma aresta ao grafo
    void addEdge(unsigned int from, unsigned int to, unsigned short int weight) {
        adjacencyList[from].emplace_back(to, weight);
    }

    void printGraph() {
        for (unsigned int v = 0; v < numVertices; v++) {
            cout << "[" << (v + 1) << "]: {";
            for (const Edge& edge : adjacencyList[v]) {
                cout << "(" << (edge.to + 1) << ", " << edge.weight << ")";
            }
            cout << "}" << endl;
        }
    }

    // Funcao para encontrar o caminho minimo usando o algoritmo de Dijkstra
    pair<vector<unsigned int>, vector<unsigned int>> dijkstra(unsigned int start) {
        vector<unsigned int> distance(numVertices, numeric_limits<int>::max());
        vector<unsigned int> predecessor(numVertices, -1);

        distance[start] = 0;

        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
        pq.push({0, start});

        while (!pq.empty()) {
            unsigned int u = pq.top().second;
            unsigned int dist = pq.top().first;
            pq.pop();

            if (dist > distance[u]) continue;


            //cout << "u = " << u << endl;
            for (const Edge& edge : adjacencyList[u]) {
                unsigned int v = edge.to;
                unsigned short int weight = edge.weight;
                //cout << "v = " << v << endl;
                if (distance[v] > distance[u] + weight) {
                    distance[v] = distance[u] + weight;
                    predecessor[v] = u;
                    pq.push({distance[v], v});
                }
            }
        }

        return {distance, predecessor};
    }

    vector<unsigned int> reconstructPath(unsigned int source, unsigned int destination, const vector<unsigned int>& predecessor) {
        vector<unsigned int> path;
        //path.reserve(numVertices);
        unsigned int current = destination;

```

```
        while (current != source) {
            path.push_back(current + 1);
            current = predecessor[current];
            //cout << "current = " << current << endl;
        }
        path.push_back(source + 1);
        //for (int i = 0; i < predecessor.size(); i++)
        //    cout << " " << predecessor[i] << " ";
        //cout << endl;
        reverse(path.begin(), path.end());

        return path;
    }
};

int main(int argc, char* argv[]) {
    if (argc != 4) {
        cerr << "Uso: " << argv[0] << " <arquivoProblema> <nrExecucoes> <
            flagDepuracao>" << endl;
        return 1;
    }

    string filename = argv[1];
    unsigned short int numExecutions = stoi(argv[2]);
    unsigned short int debugFlag = stoi(argv[3]);

    ifstream file(filename);
    if (!file.is_open()) {
        cerr << "Erro ao abrir o arquivo " << filename << endl;
        return 1;
    }

    string line;
    unsigned int numVertices = 0;
    unsigned int numEdges = 0;

    // Ler o cabecalho do arquivo e obter o numero de vertices e arestas
    while (getline(file, line)) {
        if (line.empty() || line[0] == 'p') {
            stringstream ss(line);
            string token;
            ss >> token; // descartar 'p'
            ss >> token; // esperado 'sp'
            ss >> numVertices;
            ss >> numEdges;
            break;
        }
    }

    Graph graph(numVertices);

    // Ler as arestas e adicionar ao grafo
    while (getline(file, line)) {
        if (line.empty() || line[0] != 'a') continue;
        stringstream ss(line);
        string token;
        ss >> token; // descartar 'a'
        unsigned int from, to, weight;
        ss >> from >> to >> weight;
        graph.addEdge(from - 1, to - 1, weight); // Os vertices no arquivo
            comecam de 1
    }
```

```
    file.close();

    //graph.printGraph();

    unsigned int startVertex;
    if (debugFlag == 1){
        startVertex = 0;
        numExecutions = 1;
    } else {
        startVertex = rand() % numVertices;
    }
    // Executar Dijkstra para cada vertice
    for (unsigned short int i = 0; i < numExecutions; ++i) {
        // Imprimir os caminhos minimos em arquivo, se a flag de depuracao estiver ativada.
        if (debugFlag) {
            auto begin = chrono::high_resolution_clock::now();
            pair<vector<unsigned int>, vector<unsigned int>> result = graph.dijkstra(startVertex);
            auto end = chrono::high_resolution_clock::now();
            auto elapsed = chrono::duration_cast<chrono::nanoseconds>(end - begin);
            //cout << "\nTempo de execucao [graph.dijkstra]: " << elapsed.count() * 1e-9 << endl;
            // Reconstroi o caminho minimo
            //int targetVertex;// = stoi(argv[4]);
             string s = ".gr";
            string::size_type i = filename.find(s);
            if (i != string::npos){
                filename.erase(i, s.length());
                filename = "spaths"+filename+".txt";
            }

            ofstream spath(filename);
            if (!spath.is_open()) {
                cerr << "Erro ao abrir o arquivo " << filename << endl;
                return 1;
            } else {
                for (unsigned int targetVertex = 2; targetVertex <= numVertices; targetVertex++) {
                    vector<unsigned int> path = graph.reconstructPath(startVertex, targetVertex - 1, result.second);
                    // Exibe o caminho minimo no formato especificado
                    spath << "[" << startVertex + 1 << "," << targetVertex << "](" << result.first[targetVertex - 1] << ") ";
                    for (unsigned int i = 0; i < path.size(); ++i) {
                        spath << path[i] << " ";
                    }
                    spath << endl;
                }
            }
            spath.close();
        } else {
            cout << "\nstartVertex = " << (startVertex + 1) << endl;
            auto begin = chrono::high_resolution_clock::now();
            pair<vector<unsigned int>, vector<unsigned int>> result = graph.dijkstra(startVertex);
            auto end = chrono::high_resolution_clock::now();
            auto elapsed = chrono::duration_cast<chrono::nanoseconds>(end - begin);
            //cout << "\nTempo de execucao [graph.dijkstra]: " << elapsed.
```

```
                count() * 1e-9 << endl;
            for (unsigned int targetVertex = 1; targetVertex <= numVertices;
                targetVertex++) {
                vector<unsigned int> path = graph.reconstructPath(startVertex,
                     targetVertex - 1, result.second);
                // Exibe o caminho minimo no formato especificado
                cout << "\n[" << startVertex + 1 << "," << targetVertex << "](
                    " << result.first[targetVertex - 1] << ") ";
                for (unsigned int i = 0; i < path.size(); ++i) {
                    cout << path[i] << " ";
                }
            }
            cout << endl;
            startVertex = rand() % numVertices;
        }
    }
    return 0;
}
```

Código 5: Implementação da Tarefa 2 - Algoritmo de Dijkstra para o problema de caminhos mínimos.

Conforme as instruções de uso (seção 1.3.2), na execução do programa desenvolvido, o usuário passa o nome do arquivo de instância, o número de execuções desejadas e a *flag* de depuração como uma linha de comando no terminal.

A função principal faz a leitura do arquivo de entrada, ignorando as linhas de comentário (iniciadas com o caractere 'c'), e obtém o número total de vértices e número total de de arestas. Em seguida, cria um objeto do tipo *Graph*, e continua a leitura do arquivo de entrada, atribuindo ao grafo, para cada linha lida do arquivo, uma nova aresta com seu respectivo peso ou custo. Nesse momento, o método de impressão do grafo pode ser chamado, para exibir sua representação na tela.

A partir daí, dependendo do valor da *flag* de depuração, o programa atribui ao vértice de origem o valor 0 (correspondente ao vértice 1) ou "sorteia" um novo vértice fonte. Se o usuário informou desejar que o resultado seja impresso em arquivo, o número de execuções do algoritmo para o cálculo dos caminhos mínimos será igual a 1. Do contrário, o método será executado conforme indicado pelo usuário.

O método ***dijkstra*** do objeto criado realiza a busca pelos caminhos mínimos, do vértice fonte a todos os outros vértices do grafo, e retorna dois vetores, um com as distâncias, que correspondem aos custos dos caminhos mínimos da origem para cada um dos outros vértices, e outro vetor com os predecessores de cada vértice, utilizado para a reconstrução dos caminhos.

A reconstrução dos caminhos é realizada pelo método ***reconstructPath***, que recebe o vértice de origem, o vértice de destino e o vetor de predecessores. Cada predecessor indica, para o seu vértice correspondente, qual foi o último vértice através do qual se obteve o caminho mínimo, a partir do vértice de origem. Ao percorrermos os índices do vetor de predecessores, a partir do vértice de destino, até alcançar o vértice de origem, obtemos uma lista que, invertida, corresponde ao caminho mínimo, da origem até o destino.

3 Estudo de Complexidade

Nesta seção são apresentados os estudos da complexidade do tempo de execução dos procedimentos implementados e do programa como um todo, considerando entradas de tamanho n.

3.1 Análise de complexidade do algoritmo de Strassen

Vimos, na seção 2.1 que o algoritmo de Strassen realiza 7 chamadas recursivas, que correspondem a operações de multiplicação de submatrizes de ordem $\frac{n}{2}$ x $\frac{n}{2}$. Essa é a etapa de ***divisão*** do problema em subproblemas menores. Como a computação das submatrizes desejada, C_{11}, C_{12}, C_{21} e C_{22}, da matriz resultante C é realizada por meio de operações de adição e subtração de matrizes, temos para a etapa de ***conquista*** um custo de tempo igual a $\Theta(n^2)$. Portanto, temos a seguinte relação de recorrência [1]:

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 1, \\ 7T(\frac{n}{2}) + \Theta(n^2), & \text{se } n > 1. \end{cases}$$

Se $M(n)$ é o número de multiplicações realizadas, temos: $M(n) = 7M(\frac{n}{2})$, para $n > 1$, com $M(1) = 1$.

Então:

$$M(n) = 7M(\frac{n}{2}) = 7[7M(\frac{n}{2^2})] = 7^2M(\frac{n}{2^2}) = 7^3M(\frac{n}{2^3}) = ... = 7^iM(\frac{n}{2^i}) = ... = 7^kM(\frac{n}{2^k})$$

Digamos que, nesse passo da divisão do problema, $\frac{n}{2^k}$ seja igual a 1, ou seja, $M(\frac{n}{2^k}) = M(1)$, portanto, $\frac{n}{2^k} = 1$. Isso implica que $n = 2^k$ e $k = \log_2 n$.

Voltando à relação de recorrência $M(n) = 7M(\frac{n}{2})$, temos:

$$M(2^k) = 7M(\frac{2^k}{2}) = 7M(2^{k-1}) = 7(7M(2^{k-2})) = 7(7(7M(2^{k-3}))) =$$
$$= ... = 7^iM(2^{k-i}) = ... = 7^kM(2^{k-k}) = 7^kM(1) = 7^k.$$

Mas $k = log_2 n$. Então, $M(n) = 7^{log_2 n} = n^{log_2 7} \approx n^{2,803}$. Portanto, $M(n) \in O(n^3)$. O número de operações de adição (ou subtração) realizadas pelo algoritmo de Strassen é da ordem de $6n^{2,803} - 6n^2$ [4], portanto, o algoritmo mantém sua ordem de complexidade $\Theta(n^{2,803}) \in O(n^3)$.

Em termos da implementação, considerando que as estruturas de dados utilizadas têm custo de tempo similar ao algoritmo, em termos do tamanho da entrada, para as operações de adição, subtração e multiplicação de matrizes, pode-se considerar que o programa como um todo, tem complexidade de tempo de execução igual a $\Theta(n^3)$, caso o método de multiplicação de duas matrizes escolhido pelo usuário seja o ***squareMatrixMultiply***, e igual a $O(n^3)$, caso o usuário tenha optado pelo algoritmo de Strassen (método ***strassen***).

Mesmo no caso de instâncias cuja ordem não seja uma potência de 2, em que o programa preenche automaticamente a matriz com zeros, no pior caso o custo continua a ser o acima mencionado, uma vez que, para essa operação de preenchimento, o custo é no máximo igual a $O(n^2)$.

3.2 Análise de complexidade do algoritmo de Dijkstra

O tempo de execução do algoritmo de Dijkstra depende da estrutura de dados utilizada para implementar a fila de prioridade e para representar o grafo $G = (V, E)$, em que V é o conjunto de vértices e E é o conjunto de arestas.

Se o grafo for representado por uma matriz de adjacências e a fila de prioridades for um array, o custo é $\Theta(|V|^2)$. Se o grafo for representado por listas de adjacências e a fila de prioridade for um *min-heap*, a ordem de complexidade do algoritmo vai para $O((|V| + |E|).log|V|)$ [1][2][3].

Inicialmente, o vetor de distâncias do vértice de origem até cada um dos outros vértices é preenchido com maior valor possível ($+\infty$). Essa etapa tem custo $\Theta(|V|)$. Com uma fila de prioridade implementada como um *heap*, a inserção e a remoção de um vértice tem custo $\Theta(log|V|)$. No pior caso, todos os vértices são inseridos e removidos da fila de prioridade, tomando um tempo de $O(|V|.log|V|)$. Se for necessário 'relaxar' todas as arestas, tem-se um custo adicional de $O(|E|.log|V|)$. Pode-se concluir portanto que, com essa implementação o algoritmo de Dijkstra tem complexidade de tempo na ordem de $O((|V| + |E|).log|V|)$. Uma análise linha a linha do algoritmo implmementado pode ser vista na Figura 1.

Conforme as instruções na seção 1.3.2, para executar o programa que implementa o algoritmo de Dijkstra apresentado neste trabalho, o usuário deve passar, no terminal, o nome do arquivo de instância, o número de execuções desejado e a *flag* de depuração, que indica se o resultado deve ser gravado em arquivo. O grafo foi representado com uma lista de adjacências para cada vértice e a fila de prioridade utilizada, de acordo com a documentação, tem custo de inserção e remoção logaritmico ($log_2(n)$).

Durante a leitura do arquivo de entrada, cada uma das arestas, com seu peso, é adicionada ao grafo criado. Essa operação cria as listas de adjacências do grafo e tem custo linear em função do número de arestas e do número de vértices ($O(|V|.|E|)$), dependendo do quão denso é o grafo.

```
pair<vector<unsigned int>, vector<unsigned int>> dijkstra(unsigned int start) {
  vector<unsigned int> distance(numVertices, numeric_limits<int>::max());          O(|V|)
  vector<unsigned int> predecessor(numVertices, -1);                               O(|V|)
  distance[start] = 0;                                                             O(1)
  priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;  O(log|V|)
  pq.push({0, start});                                                             O(1)
  while (!pq.empty()) {                                                            O(|V|)
    unsigned int u = pq.top().second;                            O(|V|) * O(1)
    unsigned int dist = pq.top().first;                          O(|V|) * O(1)
    pq.pop();                                                    O(|V|) * O(log|V|)
    if (dist > distance[u]) continue;                            O(|V|) * O(1)
    for (const Edge& edge : adjacencyList[u]) {                                    O(|E|)
      unsigned int v = edge.to;                                  O(|E|) * O(1)
      unsigned short int weight = edge.weight;                   O(|E|) * O(1)
      if (distance[v] > distance[u] + weight) {                  O(|E|) * O(1)
        distance[v] = distance[u] + weight;                      O(|E|) * O(1)
        predecessor[v] = u;                                      O(|E|) * O(1)
        pq.push({distance[v], v});                               O(|E|) * O(log|V|)
      }
    }
  }
  return {distance, predecessor};                                                  O(1)
}
```

Figura 1: Análise de complexidade do algoritmo de Dijkstra implementado.

Independentemente do valor da *flag* de depuração, a reconstrução do caminho mínimo, do vértice de origem para todos os outros vértices, é realizada. A reconstrução tem custo $O(|V|)$, caso o caminho mínimo de um vértice de origem até um vértice de destino passe por todos os vértices. Essa chamada é realizada $|V| - 1$ vezes, no pior caso, levando a um tempo total da ordem de $O(|V|^2)$.

Portanto, apesar de um algoritmo de Dijkstra ter tempo de execução $O((|V| + |E|).log|V|)$, o programa como um todo tem complexidade de tempo $O(|V|^2)$.

4 Testes

Esta seção descreve os experimentos realizados para os dois problemas abordados neste trabalho prático: ***Tarefa 1***, ***Cálculo da multiplicação de matrizes***; e, ***Tarefa 2***, ***Algoritmo de Dijkstra***.

O objetivo principal dos testes realizados foi avaliar se as soluções atendiam às especificações de cada tarefa, em termos de ser capaz de entregar corretamente, para cada instância de entrada, a saída desejada. O tempo de execução para diferentes instâncias também foi medido com o auxílio da biblioteca <chrono>.

4.1 Testes com as instâncias de matrizes quadradas

Durante a implementação da solução para o cálculo do produto de duas matrizes quadradas de ordem n, algumas instâncias de teste foram geradas, de acordo com o formato do ***Exemplo de caso de teste*** apresentado na especificação da ***Tarefa 1***. O exemplo citado pode ser visto na Tabela 1.

Entrada	Saída
2 1 1 `// Primeira linha da primeira matriz` 1 1 `// Segunda linha da primeira matriz` 2 2 `// Primeira linha da segunda matriz` 2 2 `// Segunda linha da segunda matriz`	4 4 4 4

Tabela 1: Exemplo de matriz resultante do produto das matrizes de entrada.

Uma característica comum aos dois principais algoritmos para o cálculo da multiplicação de duas matrizes quadradas de ordem n, a ***multiplicação recursiva simples*** (***squareMatrixMultiply***) e o ***método de Strassen*** (***strassen***), é que em razão do problema ser dividido por 2 em cada etapa recursiva, o valor de n deve ser uma potência de 2. Uma solução para isso é preencher as matrizes originais com zeros, até que se tenha uma ordem igual à potência de 2 imediatamente maior do que n.

O custo do preenchimento da matriz com zeros descrito acima é linear em relação ao número de elementos de entrada, segundo a documentação de referência, portanto, para uma matriz quadrada é da ordem de $O(n^2)$. A solução implementada [4] verifica o valor de n lido no arquivo de entrada, e faz esse preenchimento automaticamente, caso não seja uma potência de 2.

Algumas instâncias (mat-3x3-w0.dat e mat-5x5-w0.dat) foram geradas para testar se a solução implementada calcula corretamente a multiplicação de matrizes preenchidas com zeros. Instâncias com ordem n não múltipla de 2 (mat-3x3.dat e mat-7x7.dat) também foram testadas e o resultado foi o desejado. Quando a entrada tem ordem n não potência de 2, apenas os elementos da matriz resultado, dentro da ordem n, são apresentados na saída.

Todas as instâncias de teste geradas encontram-se disponíveis num repositório no GitHub (paa_tp1).

4.2 Testes com as instâncias de caminhos mínimos

As instâncias de teste para o algoritmo de Dijkstra foram fornecidas pelo Professor da disciplina e também estão disponíveis no site da competição. Além delas, algumas instâncias mais simples foram geradas para teste. Uma delas (teste.gr), com sua saída resultante (spathsteste.txt) é apresentada na Tabela 2.

Entrada	Saída
c teste.gr c p sp 10 24 c graph contains 10 nodes and 24 arcs c a 1 2 803 a 2 1 803 a 2 3 500 a 1 3 900 a 3 1 900 a 3 2 500 a 3 4 158 a 4 3 158 a 3 5 201 a 5 3 201 a 5 6 774 a 6 5 774 a 4 7 105 a 7 4 105 a 7 8 1531 a 8 7 1531 a 2 6 710 a 6 2 710 a 6 9 378 a 9 6 378 a 8 10 967 a 10 8 967 a 9 10 1673 a 10 9 1673	[1,2] (803) 1 2 [1,3] (900) 1 3 [1,4] (1058) 1 3 4 [1,5] (1101) 1 3 5 [1,6] (1513) 1 2 6 [1,7] (1163) 1 3 4 7 [1,8] (2694) 1 3 4 7 8 [1,9] (1891) 1 2 6 9 [1,10] (3564) 1 2 6 9 10

Tabela 2: Exemplo de uma instância de entrada e a saída correspondente, para o problema de caminhos mínimos.

Nos arquivos de entrada, encontram-se o número de vértices e o número de arestas, seguidos pelas linhas correspondentes às arestas e seus pesos. A saída apresenta os vértices de origem e destino, os custos e os caminhos mínimos do vértice de origem até o vértice de destino, encontrados pelo algoritmo de Dijkstra: $[s, t]$ $(custo)$ $s, v1, v2, ..., t$.

Todas as instâncias de teste (fornecidas e geradas) foram executadas com a solução implementada, e seus respectivos arquivos de saída encontram-se disponíveis no repositório paa_tp1. A única exceção em relação à disponibilidade da saída é para a instância USA-road-d.NYc.gr, cujo resultado produziu um arquivo com cerca de 800 MB.

5 Análise

Nessa seção, apresenta-se uma breve análise dos resultados obtidos com o trabalho a partir da apresentação dos tempos consumidos pelas soluções implementadas para cada um dos problemas estudados, considerando-se os diferentes tamanhos (n) das instâncias de entrada.

5.1 Análise dos resultados para a multiplicação de matrizes

A Tabela 3 apresenta os resultados de tempo de execução do programa de cálculo da multiplicação de matrizes para todas as instâncias testadas.

As instâncias com sufixo **"-w0"** (*mat-3x3-w0.dat* e *mat-5x5-w0.dat*) são matrizes de teste que foram geradas para verificar se os algoritmos implementados eram capazes de calcular corretamente o produto de matrizes de ordem não potência de 2 preenchidas com zeros. Elas têm, portanto, ordem igual à potência de 2 imediatamente maior do que a ordem n das matrizes e já foram geradas com todos os zeros necessários. Já as demais instâncias, cuja ordem não é uma potência de 2 (*mat-3x3.dat* e *mat-7x7.dat*), foram "transformadas" automaticamente pelo programa, antes do cálculo da multiplicação de matrizes.

Tabela 3: Tempos de execução multiplicação de matrizes para diferentes instâncias.

instância	**ordem**	**método**	**tempo [s]**
mat-2x2.dat	2	squareMatrixMultply	0
mat-2x2.dat	2	strassen	0
mat-2x2-1.dat	2	squareMatrixMultply	0
mat-2x2-1.dat	2	strassen	0
mat-3x3.dat	3	squareMatrixMultply	0
mat-3x3.dat	3	strassen	0
mat-3x3-w0.dat	4	squareMatrixMultply	0
mat-3x3-w0.dat	4	strassen	0
mat-4x4.dat	4	squareMatrixMultply	0
mat-4x4.dat	4	strassen	0
mat-4x4-1.dat	4	squareMatrixMultply	0
mat-4x4-1.dat	4	strassen	0
mat-5x5-w0.dat	8	squareMatrixMultply	0
mat-5x5-w0.dat	8	strassen	0
mat-7x7.dat	8	squareMatrixMultply	0
mat-7x7.dat	8	strassen	0
mat-8x8.dat	8	squareMatrixMultply	0
mat-8x8.dat	8	strassen	0
mat-16x16.dat	16	squareMatrixMultply	0,019969
mat-16x16.dat	16	strassen	0,010028
mat-32x32.dat	32	squareMatrixMultply	0,110031
mat-32x32.dat	32	strassen	0,100029
mat-64x64.dat	64	squareMatrixMultply	0,119999
mat-64x64.dat	64	strassen	0,109971

Observa-se que, para matrizes de ordem muito pequena, os dois métodos para o cálculo do produto têm praticamente o mesmo desempenho, mas à medida em que a dimensão (n) das matrizes aumenta, o algoritmo de Strassen passa a ter um desempenho ligeiramente melhor.

5.2 Análise dos resultados para o algoritmo de Dijkstra

Para todas as instâncias de entrada, o método ***dijkstra*** encontrou os encaminhos mínimos da origem para todos os outros vértices alcançáveis em tempo inferior a 330 milissegundos, conforme a Tabela 4.

Tabela 4: Tempos de execução Dijkstra para diferentes instâncias.

instância	**# de vértices**	**# de arestas**	**tempo dijkstra [s]**	**# caminhos**
rg300_768.gr	300	768	0	299
rg300_4730.gr	300	4730	0	299
rome99c.gr	3353	8859	0	3352
comp-2007-2-22c.gr	600	276666	0,010002	15
USA-road-d.NYc.gr	264346	730100	0,319999	264345

Observa-se que o número de arestas, em relação ao número de vértices, teve maior impacto no tempo de execução do algoritmo. À medida em que o grafo vai ficando mais denso e, portanto, possui maior número de arestas, o tempo para encontrar os caminhos mínimos fica maior.

De todas as instâncias, apenas em *comp-2007-2-22c.gr* os demais vértices não são alcançáveis a partir do vértice de origem e, embora apenas 15 caminhos mínimos tenham sido encontrados, como essa instância possui muitas arestas, o tempo de execução foi consideravelmente maior do que, por exemplo, a instância *rome99c.gr*, que possui 5 vezes mais vértices.

6 Considerações Finais

Esse trabalho foi dividido em duas principais tarefas: ***Tarefa 1*** - implementar um algoritmo com complexidade $O(n^3)$ para resolver o problema de multiplicação de duas matrizes de ordem n, usando a estratégia de ***divisão-e-conquista***; ***Tarefa 2*** - implementar o algoritmo de Dijkstra para resolver o problema de caminhos mínimos em grafos, a partir de um vértice fonte.

O ***algoritmo de Dijkstra*** para **caminhos mínimos em grafos** é bastante conhecido e certamente sempre é, ao menos uma vez, implementado em disciplinas de graduação dos cursos de Computação. Já **o problema de multiplicação de matrizes** nem sempre é abordado como um exemplo do paradigma de ***divisão-e-conquista***.

Para resolver essas duas tarefas, dois programas foram implementados. O primeiro recebe como entrada um arquivo de instância com a ordem das matrizes e seus respectivos valores e entrega como saída o produto dessas matrizes. Dois métodos foram implementados, o algoritmo padrão que divide repetidamente as matrizes originais em submatrizes de ordem $\frac{n}{2}$, chamado de ***squareMatrixMultiply*** ($\Theta(n^3)$), e outro que implementa o algoritmo de Strassen (***strassen***) e tem ordem de complexidade $O(n^{log_2 7}) \in O(n^3)$.

O segundo programa utiliza o método ***dijkstra*** para resolver o problema de caminhos mínimos, e recebe como entrada instâncias com o número de vértices e o número de arestas do grafo a ser percorrido, seguidos por todas as arestas com seus pesos. A saída, que pode ser exibida na tela ou armazenada em arquivo, apresenta todos os caminhos mínimos, e seus custos, de um vértice de origem s a todos os vértices t alcançáveis a partir de s.

As implementações, em linguagem de programação *C++*, foram feitas de forma gradativa, a partir do entendimento das especificações de cada uma das duas tarefas principais e de exemplos de instâncias de menor tamanho, geradas durante o processo. Em seguida, as instâncias de teste fornecidas pelo Professor foram utilizadas como entradas. Os principais desafios envolveram detalhes de sintaxe e no uso de estruturas de dados como ***vector***, tanto para a representação de matrizes, quanto para representação de grafos, e ***priority_ queue***, como fila de prioridade para o algoritmo de Dijkstra.

A realização deste primeiro trabalho prático mostrou a importância desses dois problemas para o estudo de projeto e análise de algoritmos. Aspectos associados às estruturas de dados utilizadas na implementação, bem como a representação escolhida para o problema, têm impacto importante no desempenho do algoritmo e no tempo de execução dos programas.

Como exemplo, tomemos o algoritmo de Dikjstra, que tem diferentes ordens de complexidade para o problema de caminhos mínimos, a depender se o grafo é representado por matrizes de adjacências ou por listas de adjacências, e se a lista de vértices a serem visitados é implementada com um *array* ou com um *heap*.

Referências

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms - Third Edition.* The MIT Press, Cambridge, Massachussets, 2009.

[2] Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. *Data Structures and Algorithms in Python.* Wiley, 2013.

[3] Anany Levitin. *Introduction to the design & analysis of algorithms - 3rd ed.* Pearson Education, Upper Saddle River, New Jersey, 2012.

[4] Jeffrey J. McConnell. *The analysis of algorithms: an active learning approach.* Jones and Barllett Pubhishers, Inc., Sudbury, Massachussets, 2001.

[5] Tim Roughgarden. *Algorithms Illuminated - Part 1: The Basics.* Soundlikeyourself Publishing, LLC, San Francisco, CA, 2017.