DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Instituto Superior de Engenharia do Porto

Relatório Trabalho Prático 2

Grafos

Turma 2DN

Vasco Azevedo 1202016

Tiago Pinto 1200626

Introdução

Foi nos dado um enunciado com exercícios e alguns ficheiros de texto com informações relevantes para a resolução dos mesmos. Para isso achamos melhor desenvolver um package com o nome de model onde colocamos todas as classes.

Fizemos a classe DataReader que nos permite ler todos os ficheiros e fizemos a classe Utils que contém os métodos de resolução dos exercícios.

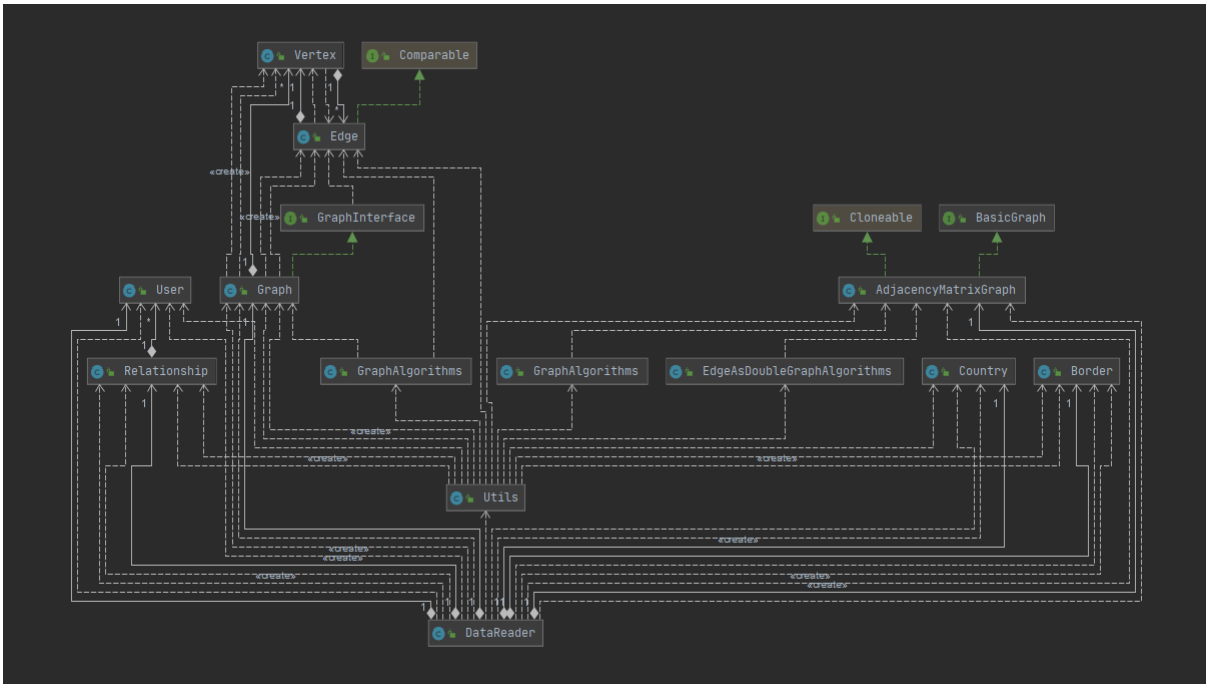

Diagrama de classes com dependências.

Algoritmos de todas as funcionalidades

Classe DataReader

Nesta classe desenvolvemos métodos para a leitura de ficheiros e para carregar os dados dos ficheiros nos respetivos construtores.

Utilizamos o “trim” para eliminar os espaços. Isto faz com que depois seja mais fácil de procurar pelos dados e reduz os erros nos testes unitários.

```
private void parseCountries() {
    String line;
    try (BufferedReader br = new BufferedReader(new java.io.FileReader(this.countriesFile))){
        while ( (line = br.readLine()) != null) {
            String[] line_split = line.split( regex: ",");

            String name = line_split[0].trim();
            String continent = line_split[1].trim();
            String population_str = line_split[2].trim();
            String capital = line_split[3].trim();
            String latitude_str = line_split[4].trim();
            String longitude_str = line_split[5].trim();

            double population = Double.parseDouble(population_str);
            double latitude = Double.parseDouble(latitude_str);
            double longitude = Double.parseDouble(longitude_str);

            Country country = new Country(name, continent, population, capital, latitude, longitude);

            countriesGraph.insertVertex(country);
        }
    } catch (FileNotFoundException e) {
        System.out.println("File doesn't exist!");
    } catch (IOException e) {
        System.out.println("Fail while reading the file.");
    }
}
```

Método para ler e carregar no construtor os países;

Algoritmos de todas as funcionalidades

Classe DataReader

```
private void parseBorders() {
    String line;
    try (BufferedReader br = new BufferedReader(new java.io.FileReader(this.bordersFile))){
        while ( (line = br.readLine()) != null) {
            String[] line_split = line.split( regex: ",");

            String country1 = line_split[0].trim();
            String country2 = line_split[1].trim();

            Border border = new Border(country1, country2);

            Country c1 = Utils.getCountryFromGraph(countriesGraph, country1);
            Country c2 = Utils.getCountryFromGraph(countriesGraph, country2);

            assert c1 != null;
            assert c2 != null;
            countriesGraph.insertEdge(c1, c2, border, Utils.distanceTwoCapitals(c1, c2));
        }
    } catch (FileNotFoundException e) {
        System.out.println("File doesn't exist!");
    } catch (IOException e) {
        System.out.println("Fail while reading the file.");
    }
}
```

Método para ler e carregar no construtor as borders;

```
private void parseUsers() {
    String line;
    try (BufferedReader br = new BufferedReader(new java.io.FileReader(this.usersFile))) {
        while ((line = br.readLine()) != null) {
            String[] line_split = line.split( regex: ",");

            String user_id = line_split[0].trim();
            String age_str = line_split[1].trim();
            String city = line_split[2].trim();

            int age = Integer.parseInt(age_str);

            User user = new User(user_id, age, city);

            relationshipsMatrix.insertVertex(user);
        }
    } catch (FileNotFoundException e) {
        System.out.println("File doesn't exist!");
    } catch (IOException e) {
        System.out.println("Fail while reading the file.");
    }
}
```

Método para ler e carregar no construtor os user;

Algoritmos de todas as funcionalidades

Classe DataReader

```java
private void parseRelationships() {
    String line;
    try (BufferedReader br = new BufferedReader(new java.io.FileReader(this.relationshipsFile))) {
        while ( (line = br.readLine()) != null) {
            String[] line_split = line.split( regex: ",");

            String user1_id = line_split[0].trim();
            String user2_id = line_split[1].trim();

            Relationship relationship = new Relationship(user1_id, user2_id);

            User user1 = Utils.getUserFromMatrix(relationshipsMatrix, user1_id);
            User user2 = Utils.getUserFromMatrix(relationshipsMatrix, user2_id);

            if (user1 == null || user2 == null) {
                //System.out.println("Erro ao carregar a relações entre o utilizador " + user1_id + " e " + user2_id + ".");
                continue;
            }
            relationshipsMatrix.insertEdge(user1, user2, relationship);
        }
    } catch (FileNotFoundException e) {
        System.out.println("File doesn't exist!");
    } catch (IOException e) {
        System.out.println("Fail while reading the file.");
    }
}
}
```

Método para ler e carregar no construtor as relationships;

Algoritmos de todas as funcionalidades

Classe Utils

Exercício 1

Neste exercício foram desenvolvidos quatro métodos (getCountryFromGraph, distanceTwoCapitals, getCountriesFromGraph, getBordersFromGraph).

O método getCountryFromGraph (parâmetros: Graph<Country, Border> graph, String country) percorre todos os vértices do grafo e verifica se o country está presente no grafo. Caso esteja retorna esse country.

```
public static Country getCountryFromGraph(Graph<Country, Border> graph, String country) {
    for (Country c : graph.vertices()) {
        if (c.getName().equals(country)) return c;
    }
    return null;
}
```

Complexidade: O(v)

O método distanceTwoCapitals (parâmetros: Country c1, Country c2) basicamente utiliza uma fórmula para calcular a distância entre duas capitais entre a longitude e a latitude das mesmas.

```
public static double distanceTwoCapitals(Country c1, Country c2) {
    double R = 6371e3;

    double a = Math.sin((c1.getLatitude() * Math.PI/180)/2) * Math.sin(((c2.getLatitude()-c1.getLatitude()) * Math.PI/180)/2)
            + Math.cos((c1.getLatitude() * Math.PI/180)) * Math.cos((c2.getLatitude() * Math.PI/180))
            * Math.sin(((c2.getLongitude()-c1.getLongitude()) * Math.PI/180)/2)
            * Math.sin(((c2.getLongitude()-c1.getLongitude()) * Math.PI/180)/2);

    double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));

    return Math.round(R * c);
}
```

Complexidade: O(1)

O método getCountriesFromGraph (parâmetros: Graph<Country, Border> graph) percorre todos os vértices, adiciona-os a uma lista de países e retorna essa lista.

```
public static List<Country> getCountriesFromGraph(Graph<Country, Border> graph) {
    List<Country> countries = new ArrayList<>();
    for (Country c : graph.vertices()) {
        countries.add(c);
    }
    return countries;
}
```

Complexidade: O(v)

Algoritmos de todas as funcionalidades

Classe Utils

Exercício 1

O método getBordersFromGraph (parâmetros: Graph<Country, Border> graph) percorre as edges do grafo, adiciona a uma lista de borders e retorna essa border.

```
public static List<Border> getBordersFromGraph(Graph<Country, Border> graph) {
    List<Border> borders = new ArrayList<>();
    for (Edge<Country, Border> c : graph.edges()) {
        borders.add(new Border(c.getVOrig().getName(), c.getVDest().getName()));
    }
    return borders;
}
```

Complexidade: O(e)

Algoritmos de todas as funcionalidades

Classe Utils

Exercício 2

Para a realização do exercício 2 foram desenvolvidos seis métodos (getUserFromMatrix, getUsersFromMatrix, getRelationshipsFromMatrix, getMostPopularUsers, getCommon, getCommonFriends, getCommon).

O método getUserFromMatrix (parâmetros: AdjacencyMatrixGraph<User, Relationship> matrixGraph, String name) percorre todos os vértices da matriz e verifica se há algum user com o name. Caso haja retorna esse user.

```
public static User getUserFromMatrix(AdjacencyMatrixGraph<User, Relationship> matrixGraph, String name) {
    for (User u : matrixGraph.vertices()) {
        if (u.getUser().equals(name)) {
            return u;
        }
    }
    return null;
}
```

Complexidade: O(v)

O método getUsersFromMatrix (parâmetros: AdjacencyMatrixGraph<User, Relationship> matrixGraph) percorre todos os vértices da matriz, coloca todos os user numa lista e retorna essa lista.

```
public static List<Relationship> getRelationshipsFromMatrix(AdjacencyMatrixGraph<User, Relationship> matrixGraph) {
    List<Relationship> relationships = new ArrayList<>();
    for (Relationship r : matrixGraph.edges()) {
        relationships.add(r);
    }
    return relationships;
}
```

Complexidade: O(e)

Algoritmos de todas as funcionalidades

Classe Utils

Exercício 2

O método getRelationshipsFromMatrix (parâmetros: AdjacencyMatrixGraph<User, Relationship> matrixGraph) percorre todas as edges do grafo, adiciona-as a uma lista e retorna essa lista.

```
public static List<Relationship> getRelationshipsFromMatrix(AdjacencyMatrixGraph<User, Relationship> matrixGraph) {
    List<Relationship> relationships = new ArrayList<>();
    for (Relationship r : matrixGraph.edges()) {
        relationships.add(r);
    }
    return relationships;
}
```

Complexidade: O(e)

O método getMostPopularUsers (parâmetros: AdjacencyMatrixGraph<User, Relationship> matrixGraph, int N) percorre todos os vértices do grafo e adiciona todos os amigos desse utilizador a uma lista chamada friends. Os amigos são aqueles que tem uma edge entre o utilizador e eles, ou seja, se existir uma edge com o utilizador1 e o utilizador2 então eles são amigos. Depois dá set no número de amigos do user e adiciona o user e os seus amigos a um mapa.

Depois compara os utilizadores e ordena-os pelo número de amigos.

Por insere os N amigos mais populares num mapa e retorna esse mesmo mapa.

```
public static Map<User, List<User>> getMostPopularUsers(AdjacencyMatrixGraph<User, Relationship> matrixGraph, int N) {
    Map<User, List<User>> users = new HashMap<>();

    for (User user : matrixGraph.vertices()) {
        List<User> friends = new ArrayList<>();
        for (User user1 : matrixGraph.vertices()) {
            if (matrixGraph.getEdge(user, user1) != null) {
                friends.add(user1);
            }
        }
        user.setNumberOfFriends(friends.size());
        users.put(user, friends);
    }

    Map<User, List<User>> treeMap = new TreeMap<>((o1, o2) -> o2.getNumberOfFriends() - o1.getNumberOfFriends());
    treeMap.putAll(users);

    Map<User, List<User>> toReturn = new HashMap<>();
    int count = 0;
    for (Map.Entry<User,List<User>> entry : treeMap.entrySet()) {
        if (count == N) break;
        count++;
        toReturn.put(entry.getKey(), entry.getValue());
    }
    return toReturn;
}
```

Complexidade: O(v^2)

Algoritmos de todas as funcionalidades

Classe Utils

Exercício 2

O método getCommon (Parâmetros: Map<User, List<User>> map) percorre o mapa, adiciona todos os seus valores a um set e retorna esse set.

```
public static Set<User> getCommon(Map<User, List<User>> map) {
    Set<User> commonFriends = new HashSet<>();
    for (Map.Entry<User,List<User>> entry : map.entrySet()) {
        commonFriends.addAll(entry.getValue());
    }
    return commonFriends;
}
```

Complexidade: O(n)

O método getCommon Friends tem como parâmetros: (AdjacencyMatrixGraph<User, Relationship> matrixGraph, int N) e basicamente chama o método getMostPopularUsers e guarda os resultados num mapa e depois retorna o tamanho de um getCommon dos Users.

```
public static int getCommonFriends(AdjacencyMatrixGraph<User, Relationship> matrixGraph, int N) {
    Map<User, List<User>> users = getMostPopularUsers(matrixGraph, N);
    Set<User> commonFriends = getCommon(users);
    return commonFriends.size();
}
```

Complexidade: O(n^2)

Algoritmos de todas as funcionalidades

Classe Utils

Exercício 3

Para a resolução deste exercício foi criado unicamente um método (checkIfConnectedAndMinNum).

Este método tem como parâmetros: AdjacencyMatrixGraph<User, Relationship> graph.

Basicamente este método começa por inicializar uma linkedlist e um set de relationships.

Verifica se a matriz é ou não conectada. Para fazer esta verificação basta igualar a linkedlist a um algoritmo de busca em largura e ver se o tamanho do linkedlist é igual ao número de vértices da matriz.

Depois basicamente faz o shortest path entre todos os users e se este path for diferente de 1 então adiciona essa relationship ao set. No entanto isto so acontece se !relationships.contains(new Relationship(u2.getUser(), u.getUser()) para que não registe a mesma amizade duas vezes.

```
public static int checkIfConnectedAndMinNum(AdjacencyMatrixGraph<User, Relationship> graph){
    LinkedList<User> ll;

    Set<Relationship> relationships = new HashSet<>();

    for (User u : graph.vertices()) {
        ll = matrixGraph.GraphAlgorithms.BFS(graph, u);
        assert ll != null;
        if (ll.size() == graph.numVertices()) {
            // determinar quantas ligações são necessárias para tornar o
            // grafo fortemente conexo (todos vertices ligados entre si)
            for (User u2 : graph.vertices()) {
                if (u2 != u) {
                    LinkedList<User> linkedList = new LinkedList<>();
                    double size = EdgeAsDoubleGraphAlgorithms.shortestPath(graph, u, u2, linkedList);
                    if (size != 1.0) {
                        // para que não adicione amizades repetidas (u1-u2 = u2-u1)
                        if (!relationships.contains(new Relationship(u2.getUser(), u.getUser()))) {
                            relationships.add(new Relationship(u.getUser(), u2.getUser()));
//                              System.out.println("size: " + size);
//                              System.out.println("user1: " + u);
//                              System.out.println("user2: " + u2);
//                              System.out.println(linkedList);
                        }
                    }
                }
            }
        }
        else return -1; // significa que não é conexo
    }

    return relationships.size();
}
```

Complexidade: v^4*(V+E)

Algoritmos de todas as funcionalidades

Classe Utils

Exercício 4

Para este exercício foram desenvolvidos 3 métodos (getCountryFromGraphWithCapital, getUserDepth, getNearFriends).

O método getCountryFromGraphWithCapital tem como parâmetros: (Graph<Country, Border> graph, String capital). Este percorre o os vértices do grafo e retorna o país cuja capital seja igual à dos parâmetros.

```
public static Country getCountryFromGraphWithCapital (Graph<Country, Border> graph, String capital) {
    for (Country c : graph.vertices()) {
        if (c.getCapital().equals(capital)) {
            return c;
        }
    }
    return null;
}
```

Complexidade: O(V)

O método getNearFriends tem como parâmetros: (AdjacencyMatrixGraph<User, Relationship> matrixGraph, Graph<Country, Border> graph, String username, int NumberOfBorders).

Este método pesquisa por um nome em todos os vértices do grafo e se encontrar guarda o user correspondente.

Depois percorre basicamente as arestas de saída do user e a uma lista chamada “friends” o user2. Isto so acontece se !relationshipEdge.getUser2().equals(username), caso contrário adiciona o user1. Depois disto guarda numa o país do user.

Depois disto basicamente faz um clone do grafo que é passado nos parâmetros para um grafo intitulado de “auxGraphUnweighted”. Percorre todos os users na list friends e faz o shortpath entre o país do user e o país de cada um dos amigos. Se o tamanho da linkedlist-1 for igual ao número de bordas então adiciona o amigo à lista “listUser”.

Seguidamente percorre todos os users desta lista e adiciona os seus respetivos países a um set chamado de “list Cities”, percorre essa lista e percorre também a listUser e, se a cidade do user for igual à capital do país, adiciona esse user a uma lista chamada de “city friends” . Por fim adiciona a um Map<Country, List<User>> o país e a respetiva “city Friends” e retorna essa mesma lista.

```java
public static Map<Country, List<User>> getNearFriends(AdjacencyMatrixGraph<User, Relationship> matrixGraph, Graph<Country, Border> graph, String username, int NumberOfBorders) {
    User user = null;
    for (User u : matrixGraph.vertices()) {
        if (u.getUser().equals(username)) {
            user = u;
            break;
        }
    }
    if (user == null) return null;
    List<User> friends = new ArrayList<>();
    for (Relationship relationshipEdge : matrixGraph.outgoingEdges(user)) {
        if (!relationshipEdge.getUser2().equals(username))
            friends.add(Utils.getUserFromMatrix(matrixGraph, relationshipEdge.getUser2()));
        else
            friends.add(Utils.getUserFromMatrix(matrixGraph, relationshipEdge.getUser1()));
    }

    Country userCountry = getCountryFromGraphWithCapital(graph, user.getCity());

//      LinkedList<Country> ll = graphbase.GraphAlgorithms.DepthFirstSearch(graph, userCountry);

    List<User> listUser = new ArrayList<>();

    Graph<Country, Border> auxGraphUnweighted = new Graph<>( directed: false);
    for (Country c : graph.vertices())
        auxGraphUnweighted.insertVertex(c);
    for (Edge<Country, Border> edge : graph.edges())
        auxGraphUnweighted.insertEdge(edge.getVOrig(), edge.getVDest(), edge.getElement(),  eWeight: 1);


    for (User friend : friends) {
        LinkedList<Country> ll = new LinkedList<>();
        graphbase.GraphAlgorithms.shortestPath(auxGraphUnweighted, userCountry, getCountryFromGraphWithCapital(graph, friend.getCity()), ll); /* shortestPath */
        int depth = ll.size();

        if (depth - 1 == NumberOfBorders) {
            listUser.add(friend);
        }
    }

    for (Country c : listCities) {
        List<User> cityFriends = new ArrayList<>();
        for (User u : listUser) {
            if (u.getCity().equals(c.getCapital())) {
                cityFriends.add(u);
            }
        }
        countriesFriends.put(c, cityFriends);
    }
    return countriesFriends;
}
```

Complexidade: O(v^3)

Algoritmos de todas as funcionalidades

Classe Utils

Exercício 5

Para a resolução deste exercício foram elaborados 4 métodos (avgNearby, centralizedCity, morePercentage, citiesMoreCentralized).

O método avgNearby tem como parâmetros: (String capital, Graph<Country,Border> graph). Este percorre todos os vértices do grafo (country), vai incrementando uma variável chamada "size" e se a capital do país for diferente da capital dos parâmetros então vai somando na variável total o shortpath de uma capital até à outra. Depois retorna a média.

```
public static double avgNearby(String capital, Graph<Country,Border> graph){
    double total=0;
    int size = 0;
    for (Country c : graph.vertices()) {
        LinkedList<Country> lk = new LinkedList<>();
        size++;
        if (!c.getCapital().equals(capital))
            total = total + GraphAlgorithms.shortestPath(graph, getCountryFromGraphWithCapital(graph, capital), c, lk);
    }
    return total/(size - 1);
}
```

Complexidade: O(v^3)

O método centralizedCity tem como parâmetros: (Graph<Country,Border> graph). Este método percorre todos os vértices do grafo e insere num hashmap a capital e a avgNearby da capital relativa ao país. Por fim retorna o sort comparado pelo value do mapa.

```
public static Map<String, Double> centralizedCity(Graph<Country,Border> graph){
    Map<String, Double> topCentralized = new HashMap<>();
    for(Country c : graph.vertices()) {
        topCentralized.put(c.getCapital(), avgNearby(c.getCapital(), graph));
    }
    return topCentralized.entrySet().stream().sorted(comparingByValue()).collect(
            toMap(Map.Entry::getKey, Map.Entry::getValue, (e1, e2) -> e2, LinkedHashMap::new));
}
```

Complexidade: O(v)

Algoritmos de todas as funcionalidades

Classe Utils

Exercício 5

O método morePercentage tem como parâmetros: (AdjacencyMatrixGraph<User, Relationship> graph, Graph<Country, Border> graph2, double percentage) .

Este método percorre os vértices do grafo2 e inicializa uma variável count a 0. Depois percorre os vértices do grafo1 e se a capital do país do grafo2 for igual à cidade do utilizador do grafo1 então incrementa count.

Depois é calculada a percentagem de capitais e se esta percentagem for maior do que a percentagem dos parâmetros então adiciona a capital do país do grafo2 a uma lista chamada "cities". Por fim retorna essa lista.

```
public static ArrayList<String> morePercentage(AdjacencyMatrixGraph<User, Relationship> graph, Graph<Country, Border> graph2, double percentage) {
    ArrayList<String> cities = new ArrayList<>();

    for (Country c : graph2.vertices()) {
        double count = 0;
        for (User u : graph.vertices()) {
            if (c.getCapital().equals(u.getCity())) {
                count++;
            }
        }
        if (count / graph.numVertices() * 100 >= percentage)
            cities.add(c.getCapital());
    }
    return cities;
}
```

Complexidade: O(v^2)

O método citiesMoreCentralized tem como parâmetros: (AdjacencyMatrixGraph<User, Relationship> graph, Graph<Country, Border> graph2, double percentage, int numCities).

É inicializado um array com os valores do *morePercentage*(graph, graph2, percentage) e um arraylist res. Também é inicializado um mapa com os valores *centralizedCity*(graph2).

Percorre o keyset das centralizedCities e caso o tamanho do arraylistsize seja menor que o numCities então inicia-se um ciclo for para adicionar a city à lista res caso city.equals(morePercentage.get(i));

```
public static ArrayList<String> citiesMoreCentralized (AdjacencyMatrixGraph<User, Relationship> graph, Graph<Country, Border> graph2, double percentage, int numCities) {
    ArrayList<String> res = new ArrayList<>();
    ArrayList<String> morePercentage = morePercentage(graph, graph2, percentage);
    Map<String, Double> centralizedCities = centralizedCity(graph2);

    for (String city : centralizedCities.keySet()) {
        if (res.size() < numCities) {
            for (int i = 0; i < morePercentage.size(); i++) {
                if (city.equals(morePercentage.get(i)))
                    res.add(city);
            }
        }
    }
    return res;
}
}
```

Complexidade: O(v^2)

Conclusão

Para finalizar, não nos foi possível a conclusão do projeto por inteiro uma vez que não tivemos possibilidade de realizar o exercício 6.

Isto deveu-se à correção de uns bugs que fomos encontrando aquando do desenvolvimento do projeto e acabamos por deixar que consumisse demasiado tempo e tempo esse que seria vital para o último exercício.

No entanto, achamos que o projeto ficou bem estruturado e com o código organizado.

Em relação a melhoramentos, seria imprescindível o aumento do número de testes unitários que, também devido ao tempo, não nos foi possível aprofundar tanto quanto gostaríamos.