



# RELATÓRIO PROJETO ESTATÍSTICAS FAO

ESTRUTURAS DE INFORMAÇÃO

INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

André Barros \_ 1211299

Carlos Lopes \_ 1211277

Ricardo Moreira \_ 1211285

Tomás Russo \_ 1211288

Tomás Lopes \_ 1211289



# Índice

O Problema.....	2
A Solução .....	3
Diagrama de classes .....	3
Estruturas auxiliares .....	6
AVL Tree.....	6
2D Tree.....	8
Algoritmos usados .....	10
LoadData.....	10
AverageProductionForArea .....	13
BestLastYear .....	17
ClosestProductionArea.....	22
AccumulatedProductionInArea .....	24
Possíveis melhorias.....	26

# O Problema

O objetivo deste projeto passa por criar uma biblioteca de classes, respetivos métodos e testes que permitam gerir a informação relativa aos dados de produtos agrícolas e pecuários recolhidos pela *FAO*.

Os dados inicialmente incluem 5 tipos de valores (*Elements*) relativos a produtos (*Items*) agrupados por áreas geográficas.

Esta informação encontra-se disponível a partir de vários ficheiros com extensão *CSV* que devem ser carregadas pelo programa.

A proposta do trabalho é tornar a introdução e consulta destes dados o mais eficientes possível a partir da escolha das estruturas de informação que melhor se adaptam ao problema. Desta forma, a estrutura principal foi criada com recurso a árvores binárias de pesquisa.

Além de armazenar os dados relativos às produções é necessário realizar pesquisas relativas às localizações geográficas dos vários países, para isso foi usada uma outra estrutura (Kd Tree) que permite a pesquisa eficiente por registos quando os mesmos estão associados a um sistema de  $k$  eixos.

# A Solução

## Diagrama de classes

Toda a estrutura de dados utilizada foi abstraída por outras classes que permitem manipular os dados através de uma interface específica que não depende da estrutura utilizada internamente

Desta forma, ganhamos flexibilidade e permite que cada módulo seja testado independentemente dos outros com todas as suas especificidades e condições.

Depois de analisado o problema apresentado, a solução encontrada foi a seguinte:

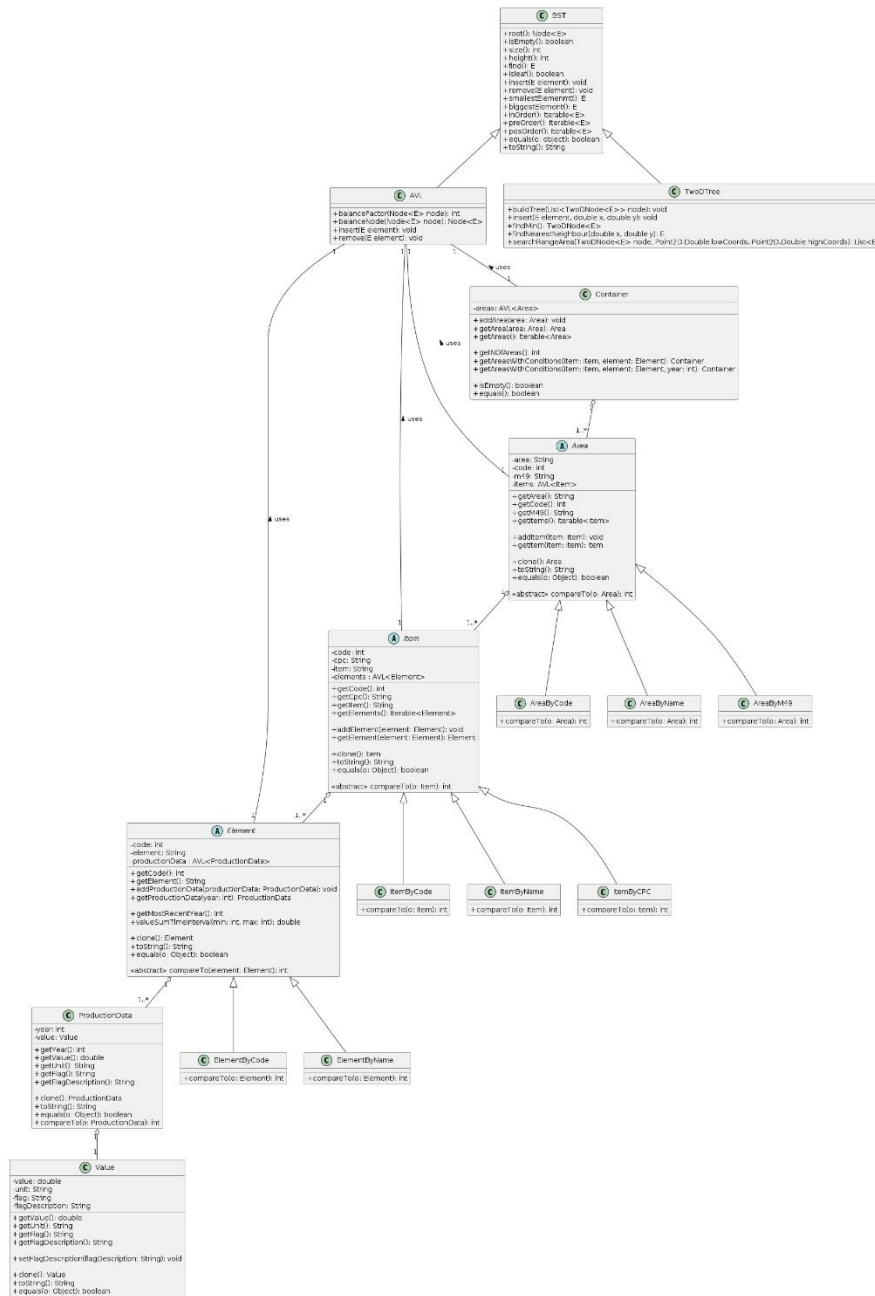


Figura 1 - Diagrama de classes

Container, Area, Item, Element, ProductionData, Value

Podemos também reparar que todas estas classes são abstratas. Como se pode ler no enunciado:

*“A pesquisa deverá ser efetuada não só através dos campos {Area Code, Item Code, Element Code, Year}, mas também [...] permitir a pesquisa por outras combinações, p.ex.: {Area, Item, Element, Year}”*

Desta forma, a forma mais simples de resolver este problema foi ter uma subclasse a implementar cada uma das classes apresentadas acima que dá *override* ao método *compareTo()*. Assim, ao construir a árvore e efetuar as comparações necessárias para decidir se o elemento pertence à subárvore esquerda ou direita, o método usado é o da subclasse.

## Estruturas auxiliares

### *AVL Tree \_ Carlos Lopes, Ricardo Moreira e Tomás Russo*

As árvores AVL, cujo nome advém dos seus criadores, Adelson-Velskii and Landis, são um tipo de árvore de binária de pesquisa (BST) cujo equilíbrio é garantido, sendo a sua altura sempre de ordem logarítmica.

A AVL foi a estrutura escolhida para este projeto na medida em que a aplicação utiliza predominantemente métodos de pesquisa, sendo que a inserção é efetuada apenas uma vez ao carregar inicialmente os dados através da leitura de um ficheiro.

O facto desta árvore se autoequilibrar irá garantir que as pesquisas feitas á mesma tenham complexidade  $O(\log n)$ . O equilíbrio é efetuado através de rotações entre os nós, garantindo que as alturas das subárvores descendentes de cada nó não difiram por mais de 1. Esse autoequilíbrio é realizado durante a inserção de dados na árvore, sempre que for encontrada uma discrepância de alturas superiores a 1 (fator de equilíbrio).

```
public Node<E> balanceNode(Node<E> node) {
    if (balanceFactor(node) < -1) {
        if (balanceFactor(node.getLeft()) > 0)
            node = twoRotations(node);
        else
            node = rightRotation(node);
    } else if (balanceFactor(node) > 1) {
        if (balanceFactor(node.getRight()) < 0)
            node = twoRotations(node);
        else
            node = leftRotation(node);
    }
    return node;
}
```

Figura 2 - Algoritmo de equilíbrio da AVL

Este autoequilíbrio irá fazer com que a inserção e remoção de dados seja um pouco mais demorada, pois a cada inserção feita é necessário garantir o equilíbrio da árvore. No entanto, como a altura da árvore será sempre de ordem logarítmica, a perda de tempo extra de inserção é compensada com uma possível redução significativa do tempo de pesquisa na árvore, que, numa árvore não equilibrada, podia atingir uma complexidade linear.

```
private Node<E> insert(E element, Node<E> node) {
    if (node == null)
        return new Node<>(element, null, null);

    int cmp = element.compareTo(node.getElement());
    if (cmp < 0)
        node.setLeft(insert(element, node.getLeft()));
    else if (cmp > 0)
        node.setRight(insert(element, node.getRight()));
    else
        return node;

    // after setting the nodes, balance the tree
    return balanceNode(node);
}
```

*Figura 3 - Método de inserção na AVL*



## 2D Tree \_ André Barros e Tomás Lopes

As *kD Trees*, são árvores que permitem pesquisas eficientes em *datasets* que usem um sistema de eixos coordenados de *k*-dimensões. Neste caso, como estamos a trabalhar no plano, é usada uma *2D Tree*, onde *X* representa a latitude e *Y* a longitude.

Como qualquer árvore binária de pesquisa, de forma a usufruir das suas vantagens nos tempos de execução a árvore deve estar equilibrada. Na *kd Tree*, devido à natureza da sua construção, não é possível mantê-la equilibrada ao longo de uma inserção de um número arbitrário de nós. No entanto, é possível garantir que a árvore fica a mais equilibrada possível se tivermos acesso a todos os elementos que vamos inserir, bem como as suas coordenadas. De forma a garantir o equilíbrio da árvore a lista deve ser ordenada, alternadamente, pela coordenada *X* e *Y*, e o elemento a ser inserido por cada iteração deve ser a mediana das sublists esquerda e direita.

```
private TwoDNode<E> buildTree(List<TwoDNode<E>> nodes, boolean divX) {
    ...

    List<TwoDNode<E>> sortedNodesList = (new MergeSort<TwoDNode<E>>()).sort(nodes, divX ? cmpX : cmpY);

    int mid = sortedNodesList.size() / 2;

    TwoDNode<E> node = new TwoDNode<E>(null, null, null, null);

    node.setCoords(sortedNodesList.get(mid).getCoords());
    node.setElement(sortedNodesList.get(mid).getElement());
    node.setLeft(buildTree(sortedNodesList.subList(0, mid), !divX));

    if (mid + 1 <= sortedNodesList.size() - 1)
        node.setRight(buildTree(sortedNodesList.subList(mid + 1, sortedNodesList.size()), !divX));

    return node;
}
```

Figura 4 - Método *buildTree()*

Desta forma, é criada uma lista de nós com os nós a inserir que é posteriormente ordenada. De seguida, a lista é dividida em duas sublists através da mediana e a mediana passa a ser a raiz dessa subárvore. O processo repete-se recursivamente até não existirem mais nós na lista.

O algoritmo utilizado para ordenar a lista de nós foi o *Merge Sort*. O método *sort()* recebe dois parâmetros, uma lista e um comparador, podendo assim ser reutilizado devido ao seu nível de abstração. Este algoritmo apresenta uma complexidade de  $O(n \log(n))$ .

A complexidade do método *buildTree* é  $O(n \log(n))$ . Uma prática comum é saltar o passo de ordenar a lista a cada iteração e usar apenas a mediana do vetor de números desordenados. Esta prática resulta em árvores relativamente equilibradas. Como se trata de um projeto académico e a performance do método foi satisfatório decidimos manter a ordenação a cada iteração.

A eficiência desta estrutura fica evidente principalmente no método *findNearestNeighbor()*, que dados dois pontos (x,y) devolve o valor do elemento do nó presente na árvore mais próximo das mesmas. Tirando partido da estrutura da 2D Tree é possível devolver o nó em tempo logarítmico,  $O(\log n)$ .

```
public E findNearestNeighbor(TwoDNode<E> node, double x, double y, int depth) {
    if (node == null)
        return null;

    double d = Point2D.distanceSq(node.getCoords().x, node.getCoords().y, x, y);
    double closestDist = Point2D.distanceSq(closest.getCoords().x, closest.getCoords().y, x, y);

    if (d < closestDist)
        closest = node;

    double delta = distanceToAxis(node, x, y, depth);
    double deltaSq = delta * delta;

    TwoDNode<E> node1 = delta < 0 ? node.getLeft() : node.getRight();
    TwoDNode<E> node2 = delta > 0 ? node.getRight() : node.getLeft();

    findNearestNeighbor(node1, x, y, depth + 1);
    if (deltaSq < closestDist)
        findNearestNeighbor(node2, x, y, depth + 1);

    return closest.getElement();
}
```

Figura 5 - Método *findNearestNeighbor()*

Para além disso, foi implementado o método *searchRangeArea()*, que dados dois pontos (x1,y1);(x2,y2) devolve uma lista de nós da árvore que estejam dentro da região retangular por esta formada. Este método é não determinístico pois o pior caso terá complexidade de  $O(n)$ , no caso da área que estamos a procurar tenha todas as áreas possíveis que estejam no Container. Por outro lado, temos o melhor caso em que a complexidade será  $O(\log(n))$ , no caso em que a área que pretendemos procurar não contenha nenhuma área do Container, ou apenas uma área.

```
public void searchRangeArea(TwoDNode<E> node, Point2D.Double lowCoords, Point2D.Double highCoords, int depth) {
    if (node == null)
        return;

    boolean isXAxis = depth % 2 == 0;

    int cmpLowResult = (isXAxis ? cmpX : cmpY).compare(node, new TwoDNode<E>(null, null, null, lowCoords));
    int cmpHighResult = (isXAxis ? cmpX : cmpY).compare(node, new TwoDNode<E>(null, null, null, highCoords));


    if (isInRange(node, lowCoords, highCoords)) {
        contained.add(node.getElement());
        searchRangeArea(node.getRight(), lowCoords, highCoords, depth + 1);
        searchRangeArea(node.getLeft(), lowCoords, highCoords, depth + 1);
    } else {
        if (cmpLowResult <= 0)
            searchRangeArea(node.getRight(), lowCoords, highCoords, depth + 1);
        if (cmpHighResult >= 0)
            searchRangeArea(node.getLeft(), lowCoords, highCoords, depth + 1);
    }
}
}
```

Figura 6 - Método *searchRangeArea()*

## Algoritmos usados

### *LoadData* \_ Alínea 1. Ricardo Moreira

A *LoadData* é a classe responsável por processar os dados lidos a partir do ficheiro “.csv” pretendido. De forma a minimizar dependências, esta classe recebe uma lista de mapas, em que cada elemento da lista corresponde a cada linha do ficheiro de dados, cada chave do mapa corresponde ao correspondente nome do campo especificado no cabeçalho do CSV e cada valor corresponde ao valor lido desta coluna do CSV. Esta lista de mapas é criada pela classe auxiliar *CSVReader* que, no fundo, dado um ficheiro “.csv”, lê-o e retorna os dados numa estrutura adequada.

A screenshot of a code editor with a dark background and light-colored text. The code defines a public class CSVReader with a private CustomScanner scanner. It has two methods: a constructor CSVReader(String fileName) that initializes the scanner and throws FileNotFoundException, and a read() method that returns a List of Maps. The code is as follows:

```
public class CSVReader {
    private CustomScanner scanner;

    public CSVReader(String fileName) throws FileNotFoundException {
        scanner = new CustomScanner(fileName);
    }

    public List<Map<String, String>> read() { return list; }
}
```

Figura 7 - Excerto da classe CSVReader

O método *Execute* da classe *LoadData* recebe uma lista de mapas e informações acerca de como deve ordenar os dados (*Area* por nome, código ou M49, *Item* por nome, código ou CPC e *Element* por nome ou código). Este começará por criar uma instância da classe *Container*, que basicamente funciona como um contentor de dados onde no caso será guardada a árvore de *Areas*. De seguida, é necessário ler o ficheiro que contém as correspondências das *Flags* para as respetivas descrições – a qual a classe *FlagReader* assumirá a responsabilidade de ler este ficheiro. A partir dela, podemos saber a que descrição corresponde cada *Flag*, bem como obter a *Flag* que indica que há falta de dados para efetuar a inserção.

A partir daqui o método irá percorrer cada elemento da lista de mapas. Como referido anteriormente, podemos obter a *Flag* que representa a falta de dados num registo através da instância do *FlagReader* criada inicialmente. Isto é útil porque pretendemos ignorar os registos que contiverem esta *Flag* e avançar para o próximo elemento da lista.

```
public class FlagReader {
    public FlagReader(String filepath) throws FileNotFoundException {
        csvreader = new CSVReader(filepath);
        missingValueFlag = null;
    }

    public void read() { ... }
    public boolean hasFlag(String flag) { ... }
    public String getFlagDesc(String flag) { ... }
    public String getMissingValueFlag() { ... }
}
```

Figura 8 - Excerto da classe *FlagReader*

O processamento que se seguirá é a inserção destes dados nas respetivas *Binary Trees*, percorrendo todos os elementos da lista, tendo em conta a forma de ordenação pedida, [tal como referido anteriormente](#). Para cada elemento da lista, o método irá verificar se uma *Area* com o mesmo código de área correspondente ao dos dados de produção deste elemento já existe na árvore de *Areas*. Caso não exista, irá ser inserida uma nova *Area* na árvore.

A partir desta *Area*, a mesma pesquisa ocorrerá para o *Item* e assim em diante para os *Elements* e *ProductionData*, o qual corresponde aos dados de produção num certo ano.

No final, é retornado o *Container* preenchido com os dados lidos no ficheiro.

```

public Container execute(List<Map<String, String>> data, Class<? extends Area> areaClass, Class<? extends Item>
itemClass, Class<? extends Element> elementClass) throws FileNotFoundException {

    Container container = new Container();

    data.forEach(row -> {
        try {
            ...

            // ignore lines with "missing value" flag
            if (flag.equals(flagReader.getMissingValueFlag()))
                throw new MissingValueException();

            ...
            Area found = container.getArea(newArea);

            if (found == null) {
                container.addArea(newArea);
                found = newArea;
            }

            ...
            Item foundItem = found.getItem(newItem);

            if (foundItem == null) {
                found.addItem(newItem);
                foundItem = newItem;
            }

            ...
            Element foundElement = foundItem.getElement(newElement);

            if (foundElement == null) {
                foundItem.addElement(newElement);
                foundElement = newElement;
            }

            ...
            ProductionData production =
                new ProductionData(year, new Value(value, unit, flag, flagReader.getFlagDesc(flag)));

            foundElement.addProductionData(production);
        } catch (Exception e) { ... }
    });

    return container;
}

```

Figura 9 - Excerto do método que insere os dados em AVL's e retorna-os numa estrutura ideal

As Árvores AVL no fundo são *Binary Trees* mas que têm um passo extra na inserção de elementos, de modo que a árvore esteja sempre balanceada. É efetuada uma pesquisa em cada árvore de modo a obter a respetiva instância, a qual concatena outra árvore. A pesquisa tem complexidade no pior caso de  $O(\log n)$ , o que corresponde à altura da árvore. A inserção nas AVL's também tem complexidade de  $O(\log n)$ .

## *AverageProductionForArea \_ Alínea 2. Carlos Lopes*

A classe *AverageProductionForArea* é responsável por retornar as médias de produção agregadas por *Item* e *Element* para uma dada *Area* e intervalo de tempo. As médias retornadas devem estar ordenadas de forma decrescente.

A classe recebe uma *String* correspondente à área, o primeiro ano e último ano do intervalo do tipo *int* e o *Container* (classe apresentada anteriormente) que contém os dados todos.

Para começar é preciso fazer algumas verificações, para ter a certeza de que todos os campos introduzidos são validos, para assim, podermos ir buscar os dados referentes à área que queremos tratar.

```
public AverageProductionForArea(String area, int firstYear, int lastYear, Container container)
throws InvalidTimeIntervalException, NullAreaException, NullContainerException{
    if(area == null || area.equals(""))
        throw new NullAreaException();

    if(container == null)
        throw new NullContainerException();

    Area a = new AreaByName(0, "", area);
    this.area = container.getArea(a);

    setTimeInterval(firstYear, lastYear);
}
```

Figura 10 – Construtor da *AverageProductionForArea*

O método *getArea* apresenta uma complexidade de  $O(\log(n))$ , pois é apenas uma simples pesquisa na *BST*.

O método responsável por realizar o tratamento dos dados é o método *execute* e a estrutura de dados adotada para retornar os dados foi uma lista.

A lista foi escolhida, por ser ótima para armazenar dados de forma sequencial, sendo fácil de os colocar com o critério de ordenação desejado. Cada elemento da lista é uma *Entry*, onde a *key* é uma outra *Entry*, que contém o par *Item-Element*, onde *Item* é a *key* e o *Element* é o seu *value*, e o seu *value* é a média correspondente.

```
public List<Map.Entry<Map.Entry<String,String>, Double>> execute()
```

Figura 11 – Estrutura de dados retornado pelo método *execute* de *AverageProductionForArea*

O método *execute* começa por ir buscar a área correspondente à *String area* passada como parâmetro no *container* passado também por parâmetro. De seguida vai buscar todos os *Items* da área obtida e para cada *Item*, vai buscar os todos os seus *Elements*.

Para cada par item e elemento calcula o somatório dos valores produzidos no intervalo de tempo especificado, para assim calcular a média de produção.

```
List<Map.Entry<Map.Entry<String,String>, Double>> list = new ArrayList<>();

Iterable<Item> items = area.getItems();

for (Item item : items) {
    Iterable<Element> elements = item.getElements();
    for (Element element : elements) {
        Map.Entry<String,String> entry = new AbstractMap.SimpleEntry<String, String>(item.getItem(), element.getElement());
        double sum = element.valueSumTimeInterval(firstYear, lastYear);
        double average = sum / (lastYear - firstYear + 1); // + 1 because first and last year are included
        list.add(new AbstractMap.SimpleEntry<Map.Entry<String,String>, Double>(entry, average));
    }
}
```

Figura 12 – Método *execute* *AverageProductionForArea*

O método *getItems* e *getElements* retornam todos os itens de uma área e todos os elementos de um item, respetivamente. Ambos os métodos têm complexidade de  $O(n)$ , pois correspondem a uma iteração da *BST*.

O método *valueSumTimeInterval* pertence à classe *Element* e recebe o primeiro e último ano do intervalo de tempo desejado. Este método retornará o somatório de produção para um dado intervalo de tempo, chamando um outro método que recebe também a *root* da árvore por parâmetro e tem o mesmo nome. Este, por sua vez, utiliza recursividade e, a partir da *root*, se o *node* que estamos a iterar estiver entre o intervalo desejado retorna o valor da produção mais o valor retornado pelo método para o seu *node* sucessor da esquerda e direita. Caso o *node* que estamos a iterar esteja do lado esquerdo do intervalo desejado, retorna apenas o valor retornado pelo método para o seu *node* sucessor da esquerda. Para a direita a mesma coisa.

```

public double valueSumTimeInterval(int min, int max) {
    return valueSumTimeInterval(productionData.root(), min, max);
}

private double valueSumTimeInterval(Node<ProductionData> node, int min, int max) {
    if (node == null) {
        return 0;
    }

    if (node.getElement().getYear() >= min && node.getElement().getYear() <= max) {
        return node.getElement().getValue() +
            valueSumTimeInterval(node.getLeft(), min, max) +
            valueSumTimeInterval(node.getRight(), min, max);
    } else if (node.getElement().getYear() < min) {
        return valueSumTimeInterval(node.getRight(), min, max);
    } else {
        return valueSumTimeInterval(node.getLeft(), min, max);
    }
}

```

Figura 13 – Método ValueSumTimeInterval

Este método apresenta uma complexidade temporal de  $O(n)$ , onde o intervalo corresponde à árvore completa. No melhor caso a complexidade é  $O(\log(n))$ , onde a árvore não tem dados para aquele intervalo, ou seja, o número de iterações é igual à altura da árvore.

Por fim, antes de retornar os dados, é necessário ordenar a lista por ordem decrescente. Para isso foi utilizado um algoritmo *Merge Sort*, referido anteriormente, utilizado pela *TwoDTree*. Este método apresenta uma complexidade de  $O(n \log(n))$ , como já referido.

```

return (new MergeSort<Map.Entry<Map.Entry<String,String>, Double>>()).sort(list, descCmp);

```

Figura 14 – Return do método execute de AverageProductionForArea

Este método recebe a lista e também comparador:

```

private final Comparator<Map.Entry<Map.Entry<String,String>, Double>> descCmp = new
Comparator<Map.Entry<Map.Entry<String,String>, Double>>(){
    @Override
    public int compare(Map.Entry<Map.Entry<String,String>, Double> o1, Map.Entry<Map.Entry<String,String>, Double> o2) {
        return - Double.compare(o1.getValue(), o2.getValue());
    }
};

```

Figura 15 – Comparador da AverageProductionForArea



Analizando, por fim, a complexidade desta funcionalidade este apresenta uma complexidade de  $O(n^3)$ , pois é necessário percorrer todos os itens de uma área, todos os elementos desse mesmo item, e calcular o somatório das produções num dados intervalo de tempo. No melhor caso a complexidade é  $O(n^2 \log(n))$ , caso não haja dados para o intervalo desejado. Sendo assim o método não é determinista.

### *BestLastYear* \_ Alínea 3. Tomás Russo

A classe *BestLastYear* é a responsável pela terceira funcionalidade do projeto, que consiste na obtenção das top-N áreas com maior valor para um dado item e elemento, no ano mais recente registado no conjunto de dados (para esse par item/elemento).

Para executar esta funcionalidade, primeiramente é necessário criar um objeto da classe *BestLastYear*, onde são passados por parâmetro os dados do ficheiro (contidos num objeto da classe *Container*), bem como o item e o elemento que queremos analisar. De reparar que as áreas são, logo no construtor, filtradas utilizando o método *getAreasWithConditions*, que retorna apenas as áreas que contêm informação para o respetivo item/elemento.

```
public BestLastYear(Container data, Item item, Element element) {  
    this.data = data.getAreasWithConditions(item, element);  
    this.item = item;  
    this.element = element;  
}
```

Figura 16 - Construtor da classe *BestLastYear*

Sendo o retorno da função um conjunto de N áreas, a estrutura que melhor se adequa para esse retorno é um *ArrayList*, onde as áreas retornadas estarão ordenadas de forma descendente do respetivo valor do par item/elemento, isto é, da área com maior valor à área com menor valor.

```
public ArrayList<Area> execute(int N) {  
    /* Create an ArrayList to store the top N areas */  
    ArrayList<Area> topN = new ArrayList<>();  
    ...  
}
```

Figura 17 - Declaração da estrutura de retorno da função

Depois de criada a estrutura de retorno, o segundo passo consiste em obter o último ano registado na árvore de Produções para o par item/elemento a analisar. Para tal, devemos iterar por todas as áreas contidas na variável *data* (inicializada no construtor), e para cada uma, obter o ano máximo registado para o nosso par item/elemento, isto é, obter o maior valor contido na árvore de Produções para esse par.

```

public ArrayList<Area> execute(int N) {
    ...
    /*
     * Get the most recent year registered in the data file,
     * for the given item and element
     */
    int lastYear = getMostRecentRegisteredYearForItemAndElement();
    ...
}

```

Figura 18 - Obtenção do último ano contido na árvore de Produções

Para obter o maior elemento numa árvore (a árvore de Produções é ordenada pelo respetivo ano), devemos obter o elemento que se encontra mais à direita. Este procedimento tem complexidade  $O(\log n)$ , na medida em que devemos iterar tantas vezes quanta a altura da árvore, que, numa AVL, é garantida ser sempre de ordem logarítmica.

```

private int getMostRecentRegisteredYearForItemAndElement() {
    int max = 0;

    for (Area area : data.getAreas()) {
        Element e = area.getItem(item).getElement(element);

        if (e.getMostRecentYear() > max) {
            max = e.getMostRecentYear();
        }
    }

    return max;
}

```

Figura 19 - Método para a obtenção do último ano registado na árvore

Tal como o método de obtenção do elemento mais à direita, o procedimento de procura do elemento  $E$  é de ordem logarítmica (tal como toda a pesquisa numa AVL). Como essas procuras são efetuadas para cada área contida no objeto *data*, a complexidade total do método acima é  $O(n \cdot \log n)$ .

Tendo o último ano de registo daquele item/elemento, e como apenas nos interessa analisar as árvores que possuem dados para aquele par item/elemento e para o ano obtido, procedemos agora a um novo filtro às áreas que tínhamos anteriormente, mas agora filtrando também pelo ano que obtivemos anteriormente, de maneira a restarem apenas as áreas que importam analisar.

```

public ArrayList<Area> execute(int N) {
    ...
    /*
     * Get only Areas with the given Item and Element for the last
     * registered year
     */
    Container newAreas = data.getAreasWithConditions(item, element, lastYear);
    ...
}

```

Figura 20 - Obtenção das áreas filtradas por item, elemento e ano

Fazendo a análise da complexidade do método acima utilizado, podemos concluir que a mesma é  $O(n)$ , na medida em que este itera por todas as áreas contidas na árvore, de maneira a fazer a seleção de todas as que possuem informação para os três parâmetros do método.

Depois de obtidas as áreas onde existem dados de produção do par item/elemento para o último ano de registo, basta agora obter as  $N$  áreas com o maior valor nesse ano. Como temos em posse apenas as áreas com um valor válido, o *Container newAreas* pode ser visto como um mapa, em que a cada área corresponde um e um só valor de produção.

Para obter as top- $N$  áreas, a estratégia utilizada passou por ordenar as áreas obtidas no passo anterior de forma descendente do seu valor, e posteriormente selecionar as primeiras  $N$  áreas, retornando-as.

```

public ArrayList<Area> execute(int N) {
    ...
    /* Get an ArrayList of areas ordered by value */
    topN = orderAreasByValueOfYear(newAreas, lastYear);

    /* Return only the first N areas */
    return topN.subList(0, N - 1);
}

```

Figura 21 - Ordenação e retorno das top- $N$  áreas

O algoritmo de ordenação utilizado no método para a ordenação das áreas foi o *Tree Sort*, fazendo uso da classe AVL criada para o projeto. Para fazer a ordenação das áreas por valor, com recurso a uma AVL, é necessário definir esse critério de ordenação. Para tal, foi criada uma *inner class*, *AreaValue*, criada no escopo da função responsável pela ordenação das áreas. Essa classe tem dois atributos: a área e o respetivo valor de produção para o ano passado por parâmetro. A classe implementa a interface *Comparable*, sendo que no método *compareTo* o atributo alvo de comparação é o valor de produção.

```

private ArrayList<Area> orderAreasByValueOfYear(Container areas, int year) {
    /* Inner class to order areas by it's value */
    class AreaValue implements Comparable<AreaValue> {
        Area area;
        double value;

        /* Constructor for AreaValue */
        public AreaValue(Area area, double value) {
            this.area = area;
            this.value = value;
        }

        /* Compare AreaValues by it's value */
        @Override
        public int compareTo(AreaValue o) {
            if (this.value > o.value)
                return 1;
            else if (this.value < o.value)
                return -1;
            else
                return 0;
        }
    }
    ...
}

```

Figura 22 - Declaração da inner class AreaValue no método de ordenação das áreas

Depois de criada a nova classe, é necessário então criar uma AVL para efetuar a ordenação, que armazenará objetos dessa mesma classe.

```

private ArrayList<Area> orderAreasByValueOfYear(Container areas, int year) {
    ...
    /* Create an AVL tree to store the areas ordered by it's value */
    AVL<AreaValue> avl = new AVL<>();

    /* Iterate over all areas to add them to the AVL tree */
    for (Area area : areas.getAreas()) {
        avl.insert(new AreaValue(area,
            area.getItem(item).getElement(element).getProductionData(year).getValue()));
    }
    ...
}

```

Figura 23 - Criação da AVL de AreaValue's e respectivo preenchimento

Tendo agora todos os objetos inseridos na AVL, e de maneira a obter esses objetos por ordem crescente, basta agora iterar por todos os elementos contidos na árvore utilizando a travessia em ordem, que nos permite obter esses elementos por ordem crescente.

```

private ArrayList<Area> orderAreasByValueOfYear(Container areas, int year) {
    ...
    /* Create an ArrayList to store the areas ordered by it's value */
    ArrayList<Area> orderedAreas = new ArrayList<>();

    /* Iterate in order over the AVL tree to add the areas to the ArrayList */
    for (AreaValue areaValue : avl.inOrder()) {
        orderedAreas.add(areaValue.area);
    }
    ...
}

```

Figura 24 - Preenchimento de um ArrayList com as áreas ordenadas por ordem crescente do seu valor

Utilizando a travessia em ordem conseguimos obter os elementos contidos na árvore por ordem crescente. No entanto, o nosso objetivo é obter uma lista com as áreas ordenadas por ordem decrescente do seu valor. Para tal, utilizamos o método *reverse* da classe *Collections*, nativa do Java. Depois, estamos prontos para retornar a lista com a ordenação correta.

```
private ArrayList<Area> orderAreasByValueOfYear(Container areas, int year) {  
    ...  
    /*  
     * Reverse the ArrayList to get the areas ordered  
     * by value in descending order  
     */  
    Collections.reverse(orderedAreas);  
    return orderedAreas;  
}
```

Figura 25 - Inversão das áreas e posterior retorno das mesmas

Analisemos agora a complexidade do método de ordenação *orderAreasByValueOfYear*. A inserção dos elementos na AVL é de complexidade  $O(n \cdot \log n)$ : para cada área ( $n$ ), devemos percorrer, no máximo, a altura da árvore para encontrar o seu correto lugar, sendo que essa altura, numa AVL, será sempre de ordem logarítmica. Depois, na travessia em ordem, são iterados todos os elementos da árvore, o que corresponde a uma complexidade linear ( $O(n)$ ). Também é de complexidade linear o método de inversão da lista, na medida em que essa tarefa pode ser feita com apenas uma iteração por todos os elementos dessa lista. Posto isto, podemos concluir que o método de ordenação utilizado é de complexidade  $O(n \cdot \log n)$ , equiparando, em termos de complexidade temporal, os melhores métodos de ordenação.

Depois de analisadas todas as complexidades dos métodos desta funcionalidade, podemos concluir que a complexidade total da funcionalidade, que corresponde à complexidade máxima de todos os métodos, é  $O(n \cdot \log n)$ .

## *ClosestProductionArea \_ Alínea 4. Tomás Lopes*

O objetivo desta funcionalidade é devolver todos os detalhes da área mais próxima de uma dada latitude e longitude que cumpra certos requisitos passados por parâmetro dos campos *Item*, *Element* e *Year*.

Logo, é necessário filtrar primeiro as áreas que cumprem esses requisitos e de seguida encontrar a mais próxima das coordenadas indicadas.

```
public Area execute(double latitude, double longitude, String item, String element, int year,
    Container data, List<Map<String, String>> geoDataMap) {
    ...

    Container filteredData = data.getAreasWithConditions(i, e, year);

    ...

    TwoDTree<Area> geoData = loadGeographicalData.execute(filteredData, geoDataMap);

    ...
    return geoData.findNearestNeighbor(latitude, longitude);
}
```

Figura 26 - Método execute() da classe ClosestProductionArea (simplificado)

Analisando os métodos desta chamada, temos o método *getAreasWithCondition()* que itera por todos os nós da árvore e verifica se o nó cumpre uma dada condição, esta condição é verificar se um dado elemento existe na sua árvore.

```
public Container getAreasWithConditions(Item item, Element element, int year) {
    Container filteredAreas = new Container();

    for (Area area : areas.inOrder()) {
        Item i = area.getItem(item);
        if (i == null)
            continue;

        Element e = i.getElement(element);
        if (e == null)
            continue;

        ProductionData p = e.getProductionData(year);
        if (p == null)
            continue;

        filteredAreas.addArea(area);
    }

    return filteredAreas;
}
```

Figura 27 - Método getAreasWithConditions()

Logo, vamos ter de iterar por quatro árvores, a primeira por todos os elementos e todas as seguintes uma única vez. Como estamos a usar árvores AVL como estrutura para armazenar os dados temos a garantia que as árvores estão equilibradas, desta forma, acabamos com uma complexidade de  $O(\log n)$  para encontrar um elemento numa árvore. Logo, ao iterar por todos os elementos da primeira árvore, obtemos uma complexidade de  $O(n \log n)$ .

De seguida, é necessário criar a 2D *tree* a partir do *Container* fornecido, como [analisado anteriormente](#) este método é de complexidade  $O(n \log(n))$ . Por fim, é usado o método *findNearestNeighbor()*, também já analisado anteriormente de ordem  $O(\log n)$ .

Desta forma, a complexidade final de todo o método é de  $O(n \log n)$ .



## *AccumulatedProductionInArea \_ Alínea 5. André Barros*

Esta classe é responsável por devolver o acumulado dos valores de produção para uma área geográfica dada por dois pontos (latitude inicial, longitude inicial, latitude final, longitude final), com recurso à 2d-Tree, para um Item Code, Element Code e Year Code.

Logo, é necessário filtrar em primeiro lugar as áreas que correspondem aos requisitos.

```
public Double execute(double x1, double y1, double x2, double y2, int itemCode, int elementCode,
    int year, Container container, List<Map<String, String>> geoData) {

    ...

    Container filteredData = container.getAreasWithConditions(i, e, year);

    ...

    return getSumProduction(areas, i, e, year);
}
```

*Figura 28 - Container com as áreas que cumprem os requisitos do método execute(simplificado)*

Desta forma, o Container vai conter todas as áreas relevantes para devolver os valores pretendidos. Como já referido anteriormente, vamos ter de iterar por 4 árvores, a complexidade desta operação vai ser de  $O(n \log(n))$

Após isso, vai ser inserido numa TwoDTree de áreas todos os países que estejam tanto no Container como na geoData, balanceado pelas coordenadas do X e Y. Como já referido anteriormente, este método tem complexidade  $O(n \log(n))$ .

```
public Double execute(double x1, double y1, double x2, double y2, int itemCode, int elementCode,
    int year, Container container, List<Map<String, String>> geoData) {

    ...

    Container filteredData = container.getAreasWithConditions(i, e, year);

    LoadGeographicalData loadGeographicalData = new LoadGeographicalData();
    TwoDTree<Area> tree = loadGeographicalData.execute(filteredData, geoData);

    ...

    return getSumProduction(areas, i, e, year);
}
```

*Figura 29 - Método execute da classe LoadGeographicalData que vai preencher a TwoDTree<Area>*

Caso não tenha a sido inserido nenhum dado na TwoDTree, é retornado -1 a soma de produção.

Caso contrário, vai ser invocado o método `searchRangeArea()`, que pertence à classe TwoDTree, em que é passado por parâmetro a área retangular que queremos pesquisar, ou seja, as coordenadas dos pontos. Este método vai devolver uma lista de áreas que cumpram os requisitos. Como já referido anteriormente, este método tem complexidade  $O(n)$ .

```
public Double execute(double x1, double y1, double x2, double y2, int itemCode, int elementCode,
    int year, Container container, List<Map<String, String>> geoData) {
    ...
    Container filteredData = container.getAreasWithConditions(i, e, year);
    LoadGeographicalData loadGeographicalData = new LoadGeographicalData();
    TwoDTree<Area> tree = loadGeographicalData.execute(filteredData, geoData);
    ...
    List<Area> areas = tree.searchRangeArea(x1, y1, x2, y2);
    return getSumProduction(areas, i, e, year);
}
```

Figura 30 - Lista de áreas que estão dentro da área pretendida usando o método SearchRangeArea

Por fim, o método `getSumProduction()` vai percorrer todas as áreas e somar os valores de produção, tendo assim o valor acumulado das produções na região pretendida. Este método terá complexidade  $O(n)$ , pois depende do número de áreas que é percorrido.

```
private double getSumProduction(List<Area> areas, ItemByCode item, ElementByCode element,
    int year) {
    double sum_production = 0;
    for (Area area : areas)
        sum_production += area.getItem(item).getElement(element).getProductionData(year).getValue();
    return sum_production;
}
```

Figura 31 - Método getSumProduction

## Possíveis melhorias

Ao introduzir os dados através do ficheiro, como temos todos os dados de antemão, a inserção pode ser efetuada de forma mais eficientemente ordenando os valores obtidos através do ficheiro antes de os inserir na árvore. Ao usarmos uma lista ordenada, se escolhermos os valores médios de cada subdivisão da árvore, podemos minimizar o número de rotações necessárias para a manter equilibrada.