Licenciatura em Engenharia Informática

Estruturas de Informação

3ª Parte

Ano Letivo 2019/2020

Breno Pacheco – 1180005

João Ferreira – 1181436

Novembro 2020

Índice

Diagrama de Classes

Visual Paradigm Standard(joaoferreira(Instituto Superior de Engenharia do Porto))

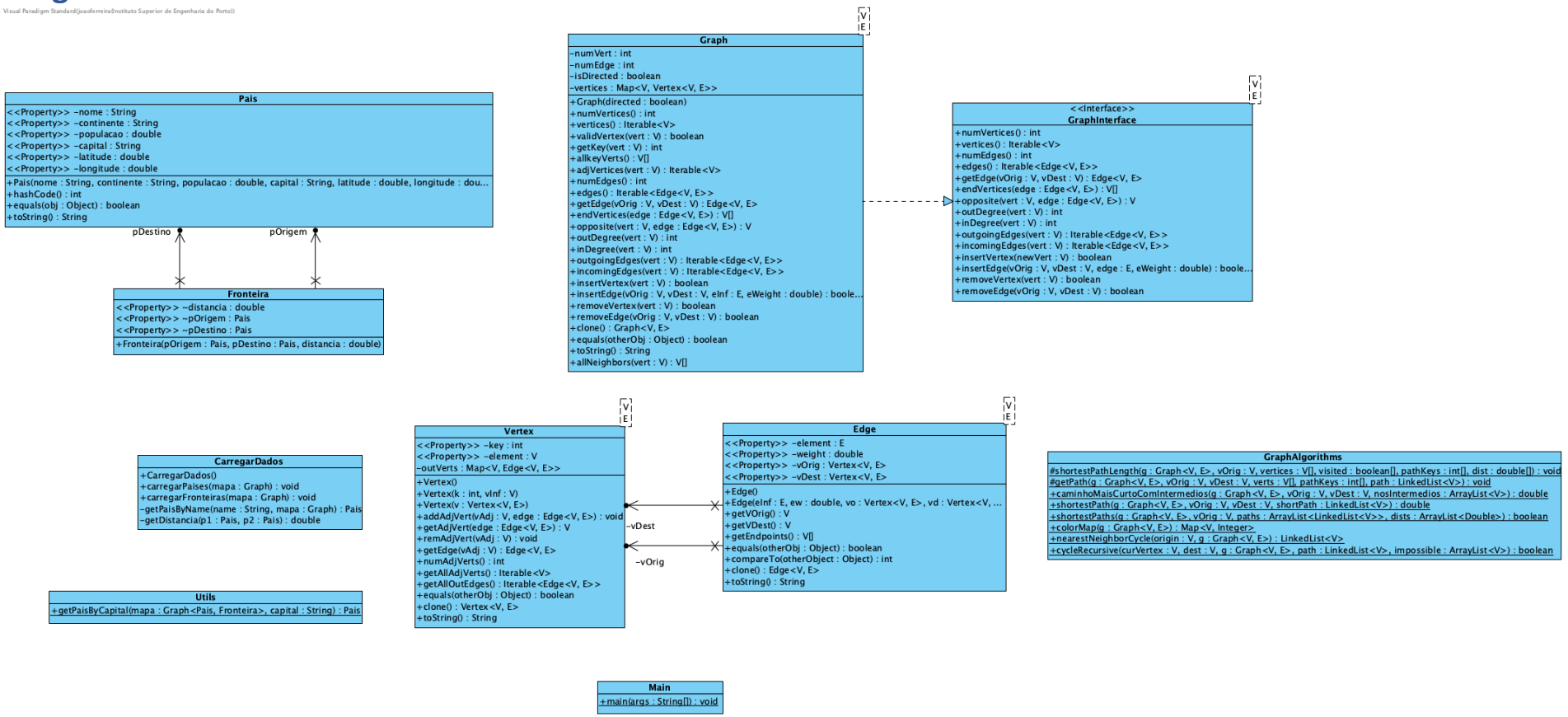

Figura 1 – Diagrama de classes do projeto criado

Binary Search Tree (BST)

Construção da BST

Uma *binary search tree* é uma árvore composta por nós e que possui as seguintes propriedades:

- A *subtree* esquerda de um determinado nó possui valores inferiores ao desse nó;
- A *subtree* direita de um determinado nó possui valores superiores ao desse nó;
- Quer a *subtree* direita como a esquerda também são BST;
- Não existem nós repetidos.

Estas propriedades fazem com que exista uma ordenação dos nós dentro da BST o que permite que operações como pesquisa, ordenação ou encontrar máximos e mínimos sejam realizados com relativa rapidez. Caso isto não acontecesse, teríamos de recorrer a método *brute force* para, por exemplo, criar um algoritmo de pesquisa.

Para a construção da BST com países foi implementada a classe *Tree_Pais* que estende a classe BST. Esta classe possui como atributo um comparador que, ao ser invocado o construtor sem parâmetros, se mantém igual ao método *compareTo* da classe país. Caso seja utilizado o construtor com argumentos, é dado como parâmetro um comparador alternativo que passará a ser utilizado dentro da classe. Para isso, também existe um *override* aos métodos *insert* da classe *BST* (Figura 2).

```
@Override
public void insert(Pais element) {
    root = insert(element, root());
}
```

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

    if (comp.compare(node.getElement(), element) < 0) {
        node.setRight(insert(element, node.getRight()));
        return node;
    }

    if (comp.compare(node.getElement(), element) > 0) {
        node.setLeft(insert(element, node.getLeft()));
        return node;
    }
    return node;
}
```

Figura 2 — Implementação dos métodos insert recorrendo ao método compare

Pesquisa de elementos

Para a pesquisa de um país com um determinado nome implementou-se o método *getPaisInstance*. Para isso, percorreu-se o array list proveniente do método *inOrder* invocando o método *getNome* da classe *País*. Ao ser encontrado um país com o nome inserido como parâmetro a sua instância é retornada. Caso não seja encontrada qualquer ocorrência é retornado o valor *null*.

```
public Pais getPaisInstance(String nomePais) {
    for (Pais p : inOrder()) {
        if (p.getNome().equalsIgnoreCase(nomePais)) {
            return p;
        }
    }
    return null;
}
```

Figura 3 — Implementação do método getPaisInstance

Ordenação de elementos

Para a ordenação dos países por ordem decrescente do número de fronteiras e por ordem crescente da população foi criado um *Comparator* (Figura 4) para ser passado por parâmetro no construtor da instância de *Tree_Pais*.

```
Comparator<Pais> comp = new Comparator<Pais>() {
    @Override
    public int compare(Pais o1, Pais o2) {
        if (o2.getFronteiras().size() > o1.getFronteiras().size()) {
            return 1;
        } else if (o2.getNumFronteiras() == o1.getNumFronteiras()) {
            if (o1.getPopulacao() > o2.getPopulacao()) {
                return 1;
            } else if (o1.getPopulacao() == o2.getPopulacao()) {
                return 0;
            } else {
                return -1;
            }
        } else {
            return -1;
        }
    }
};

Tree_Pais bstTreeB = new Tree_Pais(comp);
```

Figura 4 – Comparator que ordena os países por ordem decrescente de fronteiras e crescente de população

De seguida, foram filtrados os países apenas pertencentes a um determinado continente. E foram adicionados apenas esses à instância da árvore binária criada. Por fim foi chamado o método inOrder, que retornou os países na ordem previamente referida.

Kd-tree

Construção da kd-tree

A kd-tree de dimensão 2 exigida no exercício partilha da maior parte dos métodos definidos na BST<E>, com exceção dos métodos *insert* e de pesquisa de elementos. Dessa forma, optou-se por criar uma classe que estendesse BST, específica para agregação de países por latitude e longitude. Identificamos a implementação da árvore na Figura 5.

Para geração da árvore, foi reescrito o método *insert* para aceitar um país como argumento e realizar a alocação do novo nó. A inserção procura recursivamente uma posição para alocação deste, sendo verificado dentro de cada chamada de função pilha se o nó a ser comparado se encontra no "level" 0 ou 1, ou seja, se deve ser comparada a propriedade de latitude ou longitude do país.

A diferença entre a latitude ou longitude do nó a ser inserido e o nó comparado define se o novo nó deve ser inserido à direita ou à esquerda do elemento comparado.

```
public class Paises2DTree extends BST<Pais> {
        @Override
        public void insert(Pais p) {
                root = insert(p, root, 0);
        }
        public Node<Pais> insert(Pais p, Node<Pais> node, int level) {
                if (node == null) {
                        return new Node(p, null, null);
                }

                double diffLat = p.getLatitude() - node.getElement().getLatitude();
                double diffLong = p.getLongitude() - node.getElement().getLongitude();
                double diff;
                if (level == 0) {
                        diff = diffLat;
                        level = 1;
                } else {
                        diff = diffLong;
                        level = 0;
                }

                if (diff < 0) {
                        node.setLeft(insert(p, node.getLeft(), level));
                } else {
                        node.setRight(insert(p, node.getRight(), level));
                }

                return node;
        }
```

Figura 5 – Implementação do método de pesquisa exata findPais

Pesquisa exata

O algoritmo de pesquisa exata é semelhante ao algoritmo de de inserção descrito na construção da kd-tree. A árvore é percorrida da raíz até alguma folha, sendo comparado a latitude ou longitude, conforme o nível em que o nó de comparação se encontra. No entanto, não é inserido o elemento e sim comparado se tanto a latitude e longitude são as mesmas. Utiliza-se o método compare da classe Double para essa comparação em razão de erros de truncamento. O método de pesquisa exata findPais pode ser visto na Figura 6.

```
public Pais findPais(double latitude, double longitude) {
        Node<Pais> node = findPais(latitude, longitude, root, 0);
        if(node != null)
                return node.getElement();
        return null;
}

private Node<Pais> findPais(double lat, double lon, Node<Pais> node, int level) {
        if (node == null) {
                return null;
        }

        double diffLat = lat - node.getElement().getLatitude();
        double diffLong = lon - node.getElement().getLongitude();
        double diff;

        if(Double.compare(diffLat, 0) == 0 && Double.compare(diffLong, 0) == 0)
                return node;

        if (level == 0) {
                diff = diffLat;
                level = 1;
        } else {
                diff = diffLong;
                level = 0;
        }

        if (diff < 0) {
                return findPais(lat, lon, node.getLeft(), level);
        } else {
                return findPais(lat, lon, node.getRight(), level);
        }
}
```

Figura 6 – Implementação do método de pesquisa exata findPais

Pesquisar vizinho mais próximo

Para encontrar o vizinho mais próximo, dada uma localização, utilizamos um método recursivo que encontra o ponto onde um novo nó com essa localização deveria ser inserido (ponto de inserção). Esta etapa é semelhante ao método de inserção. O início do método é descrito na Figura 7.

```
protected Node<Pais> findNearestPais(double lat, double lon, Node<Pais> node, int level) {

        // if we hit an empty node, go back
        if (node == null) {
                return null;
        }

        // calculate dx and dy
        double diffLat = lat - node.getElement().getLatitude();
        double diffLong = lon - node.getElement().getLongitude();

        // select whether to compare dx or dy and flip level
        double diff;
        if (level == 0) {
                diff = diffLat;
                level = 1;
        } else {
                diff = diffLong;
                level = 0;
        }

        // define which node to visit and which node is the other branch
        Node<Pais> nodeToVisit;

        // the node not visited by default  (other branch) is also saved as a variable
        Node<Pais> otherBranchNode;
        if (diff < 0) {
                nodeToVisit = node.getLeft();
                otherBranchNode = node.getRight();
        } else {
                nodeToVisit = node.getRight();
                otherBranchNode = node.getLeft();
        }
```

Figura 7 – Implementação do método findNearestPais (parte 1)

Os nós visitados são então percorridos a partir do ponto de inserção (quando é feita a última chamada recursiva à função findPais) até a raiz da árvore. O nó de menor distância (currentBest) é retornado por cada chamada da pilha (Figura 8).

```
// visit one level down the three and hold teh current nearest neihbor
Node<Pais> currentBest = findNearestPais(lat, lon, nodeToVisit, level);

// if the currentBest is null, we have hit the point where the
// target would be inserted. so we return this node as the best
if (currentBest == null) {
        return node;
}
```

Figura 8 – Implementação do método findNearestPais (parte 2)

É possível que o vizinho mais próximo não se encontre neste trajeto, de forma que devemos verificar, a partir de cada nó do trajeto, se o outro ramo (não incluso no trajeto) pode conter o vizinho mais próximo. Se for o caso, este outro ramo deve ser percorrido (Figura 9).

```
// if the currentBest is null, we have hit the point where the
// target would be inserted. so we return this node as the best
if (currentBest == null) {
        return node;
}

// calculate squared distance from this node and current best node to the target
// squaredDist is used to avoid having to compute squared roots
double distNode = node.getElement().squaredDist(lat, lon);
double distCurrentBestNode = currentBest.getElement().squaredDist(lat, lon);

// check whether this node is better than the current best
// in this case, set the this node as the current best
if (distNode < distCurrentBestNode) {
        currentBest = node;
        distCurrentBestNode = distNode;
}

//  check if there could be nearer points on the other side of
// the splitting plane. if this is true, visit down the the other branch of the tree.
// and check if it returns a better node
if (diff < distCurrentBestNode && otherBranchNode != null) {
        Node<Pais> otherBranchBest =
                findNearestPais(lat, lon, otherBranchNode, level);

        double distOtherBranchBest =
                otherBranchBest.getElement().squaredDist(lat, lon);

        if (distCurrentBestNode < distCurrentBestNode) {
                currentBest = otherBranchBest;
        }
}

return currentBest;
```

Figura 9 – Implementação do método findNearestPais (parte 3)

Pesquisa por área geográfica

O método de pesquisa por área geográfica squareSearch toma como argumento as dimensões de um retângulo dentro do qual devem se encontrar os nós retornados pelo método (Figura 10).

```
public List<Pais> squareSearch(double lat1, double lon1, double lat2, double lon2) {
        List<Pais> listaPaises = new LinkedList<>();
        squareSearch(lat1, lon1, lat2, lon2, root, 0, listaPaises);
        return listaPaises;
}
```

Figura 10 – Método público de pesquisa por área geográfica

É implementado como um método recursivo que percorre os elementos da árvore a partir da raiz, verificando se os elementos se encontram dentro dos limites de um retângulo fornecido.

Para tanto, definimos algumas variáveis (Figura 11) para realizar as comparações necessárias. Xn e Yn representam os limites "inferior" e "superior" do retângulo (visto a partir do nó), conforme o nível do nó visitado. Por defeito Xn representa latitude no nível 0 e longituded no nível 1.

```
private void squareSearch(double lat1, double lon1, double lat2, double lon2,
        Node<Pais> node, int level, List<Pais> listaPaises) {
        if (node == null) {
                return;
        }
        // by default, we make x -> latitude and y -> longitude
        double x1 = lat1;
        double x2 = lat2;
        double xk = node.getElement().getLatitude();
        double y1 = lon1;
        double y2 = lon2;
        double yk = node.getElement().getLongitude();

        // select whether to compare dx or dy and flip level
        if (level == 0) {
                level = 1;
        } else {
                level = 0;
                // if we are comparing longitude, flip values for xn and yn
                double tmp;
                tmp = x1;
                x1 = y1;
                y1 = tmp;
                tmp = x2;
                x2 = y2;
                y2 = tmp;
                tmp = xk;
                xk = yk;
                yk = tmp;
        }
```

Figura 11 – Implementação do método de busca por área geográfica (parte 1)

Verificamos se à esquerda ou à esquerda do nó podem existir elementos dentro do retângulo. Se for o caso, visitamos as sub-árvores a partir destes. Se o nó estiver dentro dos limites, este é adicionado à lista de nós dentro do retângulo.

```
        // if node is to the left of square, visit left children.
        // and if it is to the right, visit right children
        if(xk < x2)
                squareSearch(lat1, lon1, lat2, lon2, node.getRight(), level, listaPaises);
        if(xk > x1)
                squareSearch(lat1, lon1, lat2, lon2, node.getLeft(), level, listaPaises);

        // if node is inside square in x, verify that it is inside
        // square in y too before inserting into the list
        if (xk >= x1 && xk <= x2)
                if (yk > y1 && yk < y2)
                        listaPaises.add(node.getElement());
}
```

Figura 12 – Implementação do método de busca por área geográfica (parte 2)

Conclusão

Este trabalho permitiu consolidar conhecimentos relativamente ao uso de árvores binárias assim como ao uso de classes genéricas. Nomeadamente, o seu uso para a implementação de uma árvore que armazenada instâncias de países.

Foi ainda útil no sentido em que permitiu adquirir conhecimentos acerca de outros tipos de árvores como é o caso da kd-tree. Que apresenta algumas diferenças em relação às árvores binárias, nomeadamente na inserção e pesquisa de elementos.