

As caixinhas de caixinhas de caixinhas...

Alunos: Nicolas Fonseca Docolas, Lucas de Fraga Silva

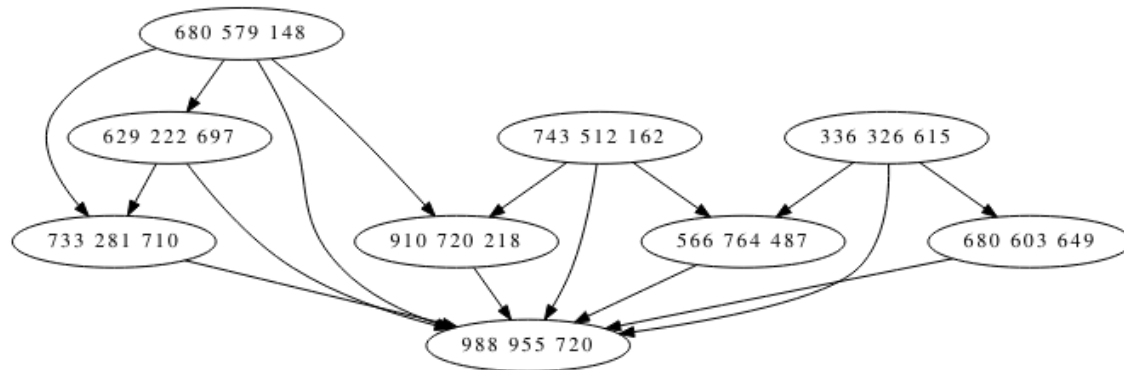


Imagem 1.1

O problema resolvido consiste na implementação de um algoritmo capaz de encontrar a maior sequência possível de caixas, considerando suas dimensões, que caibam uma dentro da outra.

Solução:

Recebendo um arquivo de texto ([Imagem 2.2](#)) simulando o banco de dados das caixas, com suas respectivas dimensões, a solução implementada foi:

Passo 1: Percorrer linha por linha armazenando os valores lidos no objeto Caixa e adicionar, iterativamente, cada uma a uma lista de caixas (List<Caixa>), conforme representado no código da linha 20 a linha 32 ([Imagem 1.2](#)).

Passo 2: Ordenar a Lista de caixas ([1*](#)) na linha 34 ([Imagem 1.2](#)).

Passo 3: Seguindo a implementação do método 'cabeDentro' ([2*](#)), adicionar as arestas válidas e criar o caminho para, depois, verificar qual será o mais longo, conforme representado na expressão lambda da linha 36 ([Imagem 1.2](#)).

Passo 4: A partir dos caminhos criados pelos métodos explicados anteriormente, basta chamar o método 'encontrarCaminhoMaisLongo' ([3*](#)), que funciona rodando paralelamente com o método 'buscaEmProfundidade' ([4*](#)), da classe 'BuscaEmProfundidade'. Este método retorna um int contendo o caminho mais longo, dado o banco de dados recebido.

Passo 5: Imprimir o valor para o usuário ([5*](#)).

[Caso de testes](#)

Ilustrações

```
19 private void lerCaminhoMaximo() {
20     try (BufferedReader br = new BufferedReader(new FileReader(arquivo))) {
21         String linha;
22         int id = 0;
23
24         List<Caixa> caixas = new ArrayList<>();
25
26         grafo = new Grafo(Integer.parseInt(br.readLine()));
27
28         while((linha = br.readLine()) != null) {
29             caixas.add(new Caixa(id++, Arrays.stream(linha.split(regex:" ")).mapToInt(Integer::parseInt).toArray()));
30         }
31
32         br.close();
33
34         Collections.sort(caixas);
35
36         IntStream.range(startInclusive:0, caixas.size()).forEach(i ->
37             IntStream.range(i + 1, caixas.size()).forEach(j -> {
38                 if (caixas.get(i).cabeDentro(caixas.get(j)))
39                     grafo.adicionarAresta(caixas.get(i).getId(), caixas.get(j).getId());
40             }));
41     } catch (Exception e) {}
42 }
```

Imagem 1.2

[\(voltar\)](#)

1*: Implementação do método 'compareTo' da classe 'Caixa'.

Note que o método possui a notação '@Override', que consiste na sobrescrita do mesmo método da Interface 'Comparable'. Neste caso, a implementação deste método serve para definir quais serão os requisitos para comparar duas variáveis do objeto 'Caixa':

```
@Override
public int compareTo(Caixa outra) {
    Comparator<Caixa> comparador = Comparator.comparingInt(
        caixa -> Arrays.compare(caixa.dimensoes, outra.dimensoes));
    return comparador.compare(this, outra);
}
```

Imagem 1.3

[\(voltar\)](#)

2*: Implementação do método 'cabeDentro' (Imagem 1.4) da classe 'Caixa'. O método segue a mesma lógica do código da Imagem 1.5:

```
public boolean cabeDentro(Caixa outra) {
    return IntStream.range(startInclusive:0, endExclusive:3).
        allMatch(i -> this.dimensoes[i] < outra.dimensoes[i]);
}
```

Imagem 1.4

```
public boolean cabeDentro(Caixa outra) {
    for (int i = 0; i < 3; i++) {
        if(this.dimensoes[i] < outra.dimensoes[i]) return false;
    }
    return true;
}
```

Imagem 1.5

[\(voltar\)](#)

3*: Implementação do método 'encontrarCaminhoMaisLongo' da classe 'BuscaEmProfundidade'.

Explicação: Itera sobre todos os vértices do grafo, mapeando todas as possibilidades, recursivamente, utilizando o método 'buscaEmProfundidade' para determinar o comprimento máximo do caminho a partir de cada vértice. Em seguida, seleciona o maior comprimento encontrado usando a função 'max'. Caso não haja valores no fluxo, retorna zero com a função 'orElse(0)'.

```
public int encontrarCaminhoMaisLongo() {
    int[] dp = new int[grafo.getNumVertices()];
    Arrays.fill(dp, -1);

    return IntStream.range(startInclusive:0, grafo.getNumVertices())
        .map(i -> buscaEmProfundidade(this.grafo, i, dp))
        .max()
        .orElse(other:0);
}
```

Imagem 1.6

[\(voltar\)](#)

4*: Implementação do método 'buscaEmProfundidade' da classe 'BuscaEmProfundidade'.

Explicação: Recebe a lista dos adjacentes do vértice v, mapeia todas as possibilidades, recursivamente, através do método 'buscaEmProfundidade' para determinar o comprimento máximo de cada caminho a partir dos vértices adjacentes. Em seguida, seleciona o maior comprimento encontrado usando a função max. Caso não existam vértices adjacentes, retorna zero com a função 'orElse(0)'.

```
private int buscaEmProfundidade(Grafo grafo, int v, int[] dp) {
    if (dp[v] != -1) return dp[v];

    int comprimentoMax;
    comprimentoMax = 1 + grafo.adjacentes(v).stream()
        .mapToInt(vizinho -> buscaEmProfundidade(grafo, vizinho, dp))
        .max()
        .orElse(other:0);

    dp[v] = comprimentoMax;
    return comprimentoMax;
}
```

Imagem 1.7

[\(voltar\)](#)

5*: Sobrescrevendo o método 'toString', da classe 'Leitura', basta instanciar um novo objeto desta classe, passando o diretório do banco de dados como parâmetro (Imagem 1.9).

```
@Override
public String toString() {
    calculaResultado();
    return "Caminho mais longo para " + arquivo.split(regex:"_")[1].split(Pattern.quote(s:"."))[0] + " caixas: " + caminhoMaximo;
}
```

Imagem 1.8

[\(voltar\)](#)

```
public static void main(String[] args) {
    System.out.println(new Leitura(arquivo:"./arquivos/tamanho_10.txt"));
}
```

Imagem 1.9

Testes

Executando para estes casos, obteve-se os resultados apresentados na [Imagem 2.1](#).

```
public static void main(String[] args) {  
    String dir = "./arquivos/";  
    System.out.println(new Leitura(dir + "tamanho_10.txt"));  
    System.out.println(new Leitura(dir + "tamanho_50.txt"));  
    System.out.println(new Leitura(dir + "tamanho_100.txt"));  
    System.out.println(new Leitura(dir + "tamanho_798.txt"));  
    System.out.println(new Leitura(dir + "tamanho_1000.txt"));  
    System.out.println(new Leitura(dir + "tamanho_5000.txt"));  
    System.out.println(new Leitura(dir + "tamanho_10000.txt"));  
}
```

Imagem 2.0

```
nicolasdoc@Nicolass-MacBook-Pro Alest2-T2 % java App.java  
Caminho mais longo para 10 caixas: 4  
Tempo: 2ms  
Caminho mais longo para 50 caixas: 11  
Tempo: 2ms  
Caminho mais longo para 100 caixas: 12  
Tempo: 3ms  
Caminho mais longo para 798 caixas: 29  
Tempo: 91ms  
Caminho mais longo para 1000 caixas: 34  
Tempo: 53ms  
Caminho mais longo para 5000 caixas: 59  
Tempo: 432ms  
Caminho mais longo para 10000 caixas: 79  
Tempo: 1585ms
```

Imagem 2.1

10
991 443 126
733 281 710
910 720 218
743 512 162
988 955 720
680 603 649
336 326 615
566 764 487
680 579 148
629 222 697

Imagem 2.2

[\(voltar\)](#)