Projeto Final - EDBII & LPII

Gregorio Pinheiro Cunha
Rafael Gomes Dantas Gurgel Siqueira

Introdução

O problema proposto consiste em um conjunto de n casas, onde se quer fazer conexões entre elas, cada qual com o seu próprio custo (distância).

A solução visa encontrar um conjunto de conexões que possuam o menor custo possível e que conectem todas as casas, considerando cada casa limitada a ter no máximo d conexões.

Resolução do Problema

- Solução Simples
- Solução Aprimorada

Resolução do Problema

- **Solução Simples**
- Solução Aprimorada

Resolução do Problema - Solução Simples

Esta solução se baseia na estratégia de testar todas as possíveis árvores e consiste no seguinte funcionamento:

1. Realiza a leitura do arquivo de entrada, armazenando o número de casas, o máximo de conexões permitidas por casa e as arestas representando o custo de conexão de duas casas.

Resolução do Problema - Solução Simples

Esta solução se baseia na estratégia de testar todas as possíveis árvores e consiste no seguinte funcionamento:

1. Realiza a leitura do arquivo de entrada, armazenando o número de casas, o máximo de conexões permitidas por casa e as arestas representando o custo de conexão de duas casas.
2. Calcula o número de possibilidades de arestas conectadas.

Resolução do Problema - Solução Simples

Esta solução se baseia na estratégia de testar todas as possíveis árvores e consiste no seguinte funcionamento:

1. Realiza a leitura do arquivo de entrada, armazenando o número de casas, o máximo de conexões permitidas por casa e as arestas representando o custo de conexão de duas casas.
2. Calcula o número de possibilidades de arestas conectadas.

   Por exemplo, para $n=5$, teríamos o número de arestas igual a $n(n-1)/2 = 5*4/2 = 10$ arestas e o número de possibilidades igual a $2^{10} = 1024$.

Resolução do Problema - Solução Simples

Esta solução se baseia na estratégia de testar todas as possíveis árvores e consiste no seguinte funcionamento:

1. Realiza a leitura do arquivo de entrada, armazenando o número de casas, o máximo de conexões permitidas por casa e as arestas representando o custo de conexão de duas casas.
2. Calcula o número de possibilidades de arestas conectadas.
3. São testadas todas as possibilidades, analisando exclusivamente as que atenderem ao critério de ter apenas *n-1* arestas, e adiciona essas arestas caso não formem ciclo e não ultrapassem o limite de conexões permitido.

Resolução do Problema - Solução Simples

Esta solução se baseia na estratégia de testar todas as possíveis árvores e consiste no seguinte funcionamento:

1. Realiza a leitura do arquivo de entrada, armazenando o número de casas, o máximo de conexões permitidas por casa e as arestas representando o custo de conexão de duas casas.
2. Calcula o número de possibilidades de arestas conectadas.
3. São testadas todas as possibilidades, analisando exclusivamente as que atenderem ao critério de ter apenas *n-1* arestas, e adiciona essas arestas caso não formem ciclo e não ultrapassem o limite de conexões permitido.
4. Armazena o custo e o índice da árvore caso esse custo seja menor do que menor custo já encontrado.

Resolução do Problema - Solução Simples

Esta solução se baseia na estratégia de testar todas as possíveis árvores e consiste no seguinte funcionamento:

1. Realiza a leitura do arquivo de entrada, armazenando o número de casas, o máximo de conexões permitidas por casa e as arestas representando o custo de conexão de duas casas.
2. Calcula o número de possibilidades de arestas conectadas.
3. São testadas todas as possibilidades, analisando exclusivamente as que atenderem ao critério de ter apenas *n-1* arestas, e adiciona essas arestas caso não formem ciclo e não ultrapassem o limite de conexões permitido.
4. Armazena o custo e o índice da árvore caso esse custo seja menor do que menor custo já encontrado.
5. Após testar todas as possibilidades, salva, em arquivo de texto, a árvore de menor custo obtida.

Resolução do Problema

- Solução Simples
- **Solução Aprimorada**

Resolução do Problema - Solução Aprimorada

A 2ª solução se baseia na estratégia de testar todas as possíveis árvores em ordem de custo crescente de acordo com o artigo de [Sörensen and Janssens 2005] e consiste na seguinte lógica:

1. Realiza a leitura do arquivo de entrada, armazenando o número de casas, o máximo de conexões permitidas por casa e as arestas representando o custo de conexão de duas casas.

Resolução do Problema - Solução Aprimorada

A 2ª solução se baseia na estratégia de testar todas as possíveis árvores em ordem de custo crescente de acordo com o artigo de [Sörensen and Janssens 2005] e consiste na seguinte lógica:

1. Realiza a leitura do arquivo de entrada, armazenando o número de casas, o máximo de conexões permitidas por casa e as arestas representando o custo de conexão de duas casas.
2. Roda o algoritmo Kruskal para obter a árvore geradora mínima (MST) e, caso atenda ao limite máximo de conexões por nó, armazena essa árvore em um arquivo de texto encerrando o processamento. Caso contrário, segue para o próximo passo.

Resolução do Problema - Solução Aprimorada

A 2ª solução se baseia na estratégia de testar todas as possíveis árvores em ordem de custo crescente de acordo com o artigo de [Sörensen and Janssens 2005] e consiste na seguinte lógica:

1. Realiza a leitura do arquivo de entrada, armazenando o número de casas, o máximo de conexões permitidas por casa e as arestas representando o custo de conexão de duas casas.
2. Roda o algoritmo Kruskal para obter a árvore geradora mínima (MST) e, caso atenda ao limite máximo de conexões por nó, armazena essa árvore em um arquivo de texto encerrando o processamento. Caso contrário, segue para o próximo passo.
3. Este último passo utiliza a estratégia do artigo, com o uso de partições, para calcular as MST's em ordem de custo crescente e interrompe o processamento quando acha uma MST que atende ao critério do número máximo de conexões por nó, salvando essa árvore em um arquivo de texto.

Estruturas de Dados Utilizadas

- Estruturas de dados simples
- Estruturas de dados avançadas

Estruturas de Dados Utilizadas

- **Estruturas de dados simples**
- Estruturas de dados avançadas

Estruturas de Dados Utilizadas - Simples

- **Lista.** Representada pela interface List e com a implementação de ArrayList. Esta estrutura, com essa implementação, foi escolhida, pois permite inserção de elementos mantendo ordem específica entre os elementos, também permite a ordenação dos elementos e o percorrimento da lista, busca por elemento específico e retorno de elementos em índices determinados.
- **Conjunto.** Representado pela interface Set e com a implementação de HashSet. Esta estrutura, com essa implementação, foi escolhida por realizar as operações de inserção, remoção e retorno em tempo constante (O(1)) e por utilizar pouca memória.

Estruturas de Dados Utilizadas

- Estruturas de dados simples
- **Estruturas de dados avançadas**

Estruturas de Dados Utilizadas - Avançadas

- **Conjuntos Disjuntos.** Esta estrutura de dados avançada foi desenvolvida para este projeto para realizar as operações de um Conjunto Disjunto: MakeSet, Union e FindSet. Com estas operações é possível conectar $n - 1$ nós, cada qual representando uma casa, sem formar ciclos, obtendo, assim, árvores.

Padrão de projeto MVC

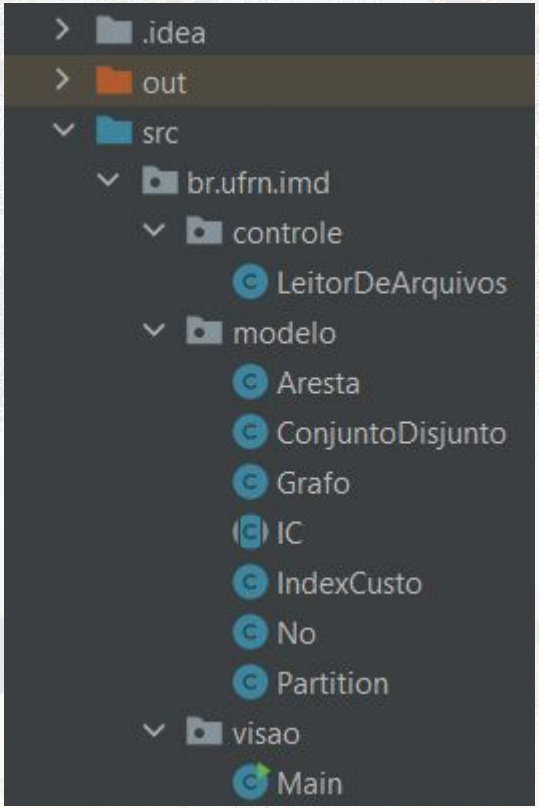
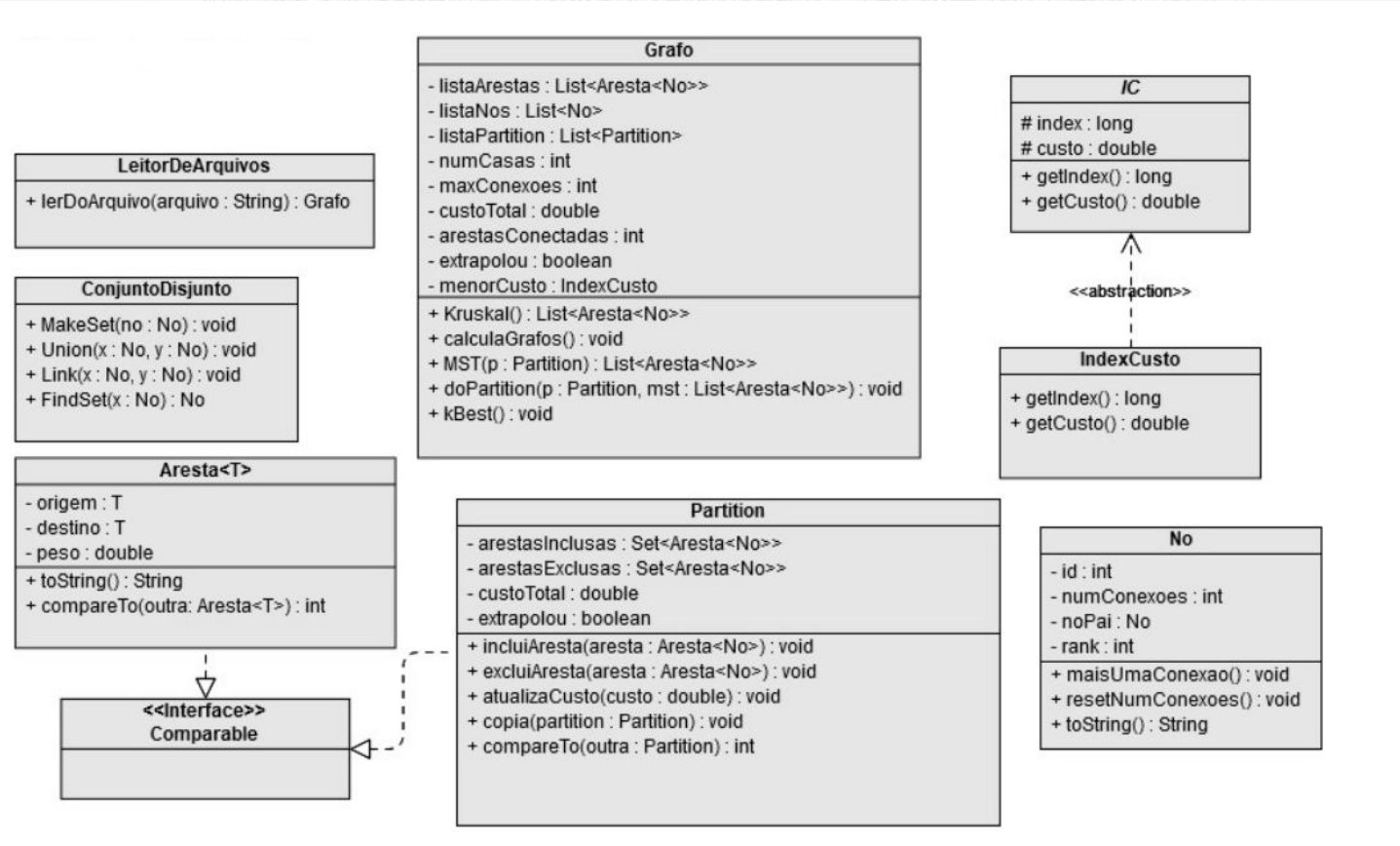


Diagrama de Classes



Tratamento de Exceções

```java
public static Grafo lerDoArquivo(String arquivo) {

    File file = new File(arquivo);
    Scanner scanner = null;
    try{
        scanner = new Scanner(file);
    } catch (IOException e){
        System.out.println("Arquivo invalido!");
    }

    String linha = scanner.nextLine();
    String[] elementosLinha = linha.split( regex: " ");

    if(elementosLinha.length != 2) {
        System.out.println("Arquivo com número inválido de argumentos.");
        return null;
    }

    List<Integer> lista = new ArrayList<>();

    for(int i = 0; i < elementosLinha.length; i++) {
        try{
            lista.add( Integer.parseInt(elementosLinha[i]));
        } catch (NumberFormatException e){
            System.out.println("Exceção: " + e.getMessage());
        }
    }
```

Classe abstrata e Herança

```java
package br.ufrn.imd.modelo;

public abstract class IC {

    2 usages
    protected long index;
    2 usages
    protected double custo;

    2 usages  1 implementation
    public abstract long getIndex();

    4 usages  1 implementation
    public abstract double getCusto();
}
```

```java
package br.ufrn.imd.modelo;

public class IndexCusto extends IC {

    3 usages
    public IndexCusto(long index, double custo) {
        this.index = index;
        this.custo = custo;
    }

    2 usages
    public long getIndex() { return index; }

    4 usages
    public double getCusto() { return custo; }
}
```

Interface e Polimorfismo Dinâmico

```java
package br.ufrn.imd.modelo;

public class Aresta<T> implements Comparable<Aresta<T>>{

    3 usages
    private T origem;
    3 usages
    private T destino;
    5 usages
    private double peso;

    1 usage
    public Aresta(T origem, T destino, double peso) {
        this.origem = origem;
        this.destino = destino;
        this.peso = peso;
    }

    5 usages
    public T getOrigem() { return origem; }

    5 usages
    public T getDestino() {
        return destino;
    }

    4 usages
    public double getPeso() { return peso; }

    4 usages
    public double getPeso() { return peso; }

    @Override
    public String toString() { return origem + " ---- " + peso + " ---- " + destino; }

    @Override
    public int compareTo(Aresta<T> outra) { return Double.compare(this.peso, outra.peso); }
}
```

Polimorfismo Estático

```java
public class Partition implements Comparable<Partition>{

    10 usages
    private Set<Aresta<No>> arestasInclusas;
    8 usages
    private Set<Aresta<No>> arestasExclusas;

    11 usages
    private double custoTotal;

    8 usages
    private boolean extrapolou;

    4 usages
    public Partition() {
        arestasInclusas = new HashSet<>();
        arestasExclusas = new HashSet<>();
        custoTotal = 0;
        extrapolou = false;
    }

    public Partition(double custoTotal, boolean extrapolou) {
        arestasInclusas = new HashSet<>();
        arestasExclusas = new HashSet<>();
        this.custoTotal = custoTotal;
        this.extrapolou = extrapolou;
    }
```

```java
public Partition(Partition partition) {
    this.arestasInclusas = new HashSet<>(partition.getArestasInclusas());
    this.arestasExclusas = new HashSet<>(partition.getArestasExclusas());
    this.custoTotal = partition.getCustoTotal();
    this.extrapolou = partition.isExtrapolado();
}

public Partition(Set<Aresta<No>> arestasInclusas,
                 Set<Aresta<No>> arestasExclusas, double custoTotal, boolean extrapolou) {
    this.arestasInclusas = new HashSet<>(arestasInclusas);
    this.arestasExclusas = new HashSet<>(arestasExclusas);
    this.custoTotal = custoTotal;
    this.extrapolou = extrapolou;
}
```

Complexidade das Soluções

- A primeira estratégia possui complexidade assintótica $O\left(2^{n^2} + n \cdot C_{\frac{n(n-1)}{2},(n-1)}\right)$.

Onde n é o número de casas.

Já a segunda solução possui complexidade assintótica dada por $\boldsymbol{O(N.|E|log|E| + N^2)}$.

Onde $\boldsymbol{N}$ é o número de árvores geradoras do grafo representando todas as conexões das casas e $\boldsymbol{|E|}$ é o número de arestas desse grafo.

Comparação de tempos (processador de 4 GHz)

n	A	Possibilidades	tempo (s)			Combinações	tempo (s)		
5	10	1.024	0,000000256			210	0,0000000525		
6	15	32.768	0,000008192			3003	0,00000075075		
7	21	2.097.152	0,000524288			54.264	0,000013566		
8	28	268.435.456	0,067108864			1.184.040	0,00029601		
9	36	68.719.476.736	17,18			3,03E+07	0,007565085		
10	45	3,52E+13	8.796,09	2,44	horas	8,86E+08	0,2215407838		
11	55	3,60E+16	9.007.199,25	104,25	dias	2,92E+10	7,312162358		
12	66	7,38E+19	1,84E+10	584,94	anos	1,07E+12	268,520699	4,48	min
13	78	3,02E+23	7,56E+13	2,40E+06	anos	4,34E+13	10.857,74	3,02	horas
14	91	2,48E+27	6,19E+17	1,96E+10	anos	1,92E+15	479.320,75	5,55	dias
15	105	4,06E+31	1,01E+22	3,22E+14	anos	9,17E+16	22.937.154,38	265,48	dias
16	120	1,33E+36	3,32E+26	1,05E+19	anos	4,73E+18	1,18E+09	37,50	anos
17	136	8,71E+40	2,18E+31	6,91E+23	anos	2,61E+20	6,54E+10	2.072,47	anos
18	153	1,14E+46	2,85E+36	9,05E+28	anos	1,54E+22	3,85E+12	122.208,88	anos

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). Introduction to Algorithms. The MIT Press, 3th edition.

Sörensen, K. and Janssens, G. K. (2005). An algorithm to generate all spanning trees of a graph in order of increasing cost. SciELO.