UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

ESCOLA DE INFORMÁTICA APLICADA

Ordenação Topológica

Bruna Ruback Frauches

Emerson de Santana Emidio

Marcio Brasil Fernandes Loureiro

RIO DE JANEIRO, RJ - BRASIL

Dezembro de 2023

ÍNDICE

1 Introdução

1.1 Problema

Na ciência da computação, uma classificação topológica ou ordenação topológica de um gráfico direcionado é uma ordenação linear de seus vértices de modo que, para cada aresta direcionada *(u,v)* do vértice *u* ao vértice *v*, u vem antes de *v* na ordenação. Precisamente, uma ordenação topológica é uma travessia de gráfico em que cada nó *v* é visitado somente depois que todas as suas dependências são visitadas. Uma ordenação topológica é possível se e somente se o gráfico não tiver ciclos direcionados, ou seja, se for um gráfico acíclico direcionado.

Por exemplo, os vértices do gráfico podem representar tarefas a serem executadas e as bordas podem representar restrições de que uma tarefa deve ser executada antes de outra; nesse caso, uma ordenação topológica é apenas uma sequência válida para as tarefas. A classificação topológica tem muitas aplicações, especialmente em problemas de classificação, como o conjunto de arcos de feedback.

2 Ordenação Topológica

2.1 Abordagem

Para a resolução do trabalho, definimos as classes internas *Elo* e *EloSucessor*. A primeira é uma classe privada interna que irá armazenar um contador, uma referência para a lista de sucessores de cada elo e um ponteiro para o próximo elo da lista.

Além disso, temos uma chave que se responsabiliza por armazenar todas as chaves dos elos criados durante a execução do programa.

```
public class OrdenacaoTopologica
{
    private class Elo
    {
        public int chave;
        public int contador;
        public Elo prox;
        public EloSucessor listaSuc;

        public Elo()
        {
            prox = null;
            contador = 0;
            listaSuc = null;
        }

        public Elo(int chave, int contador, Elo prox, EloSucessor listaSuc)
        {
            this.chave = chave;
            this.contador = contador;
            this.prox = prox;
            this.listaSuc = listaSuc;
        }
    }
```

Classe Elo com seus atributos

Já a classe EloSucessor faz com que sejamos capazes de armazenar a lista de sucessores de cada elo. Conta com uma referência para o id de cada elo e um ponteiro para o próximo elo da lista de sucessores.

```
private class EloSucessor
{
    public Elo id;
    public EloSucessor prox;

    public EloSucessor()
    {
        id = null;
        prox = null;
    }

    public EloSucessor(Elo id, EloSucessor prox)
    {
        this.id = id;
        this.prox = prox;
    }
}
```

Classe Elosucessor com seus atributos

2.2 Construção do grafo

No método geraGrafo(), recebemos o número de vértices e a probabilidade por parâmetro, criamos um loop para adicionar vértices de 0 a n através da função adicionarVertice().

Após isso, construímos um for aninhado, em que a primeira parte recebe o elo corrente, iniciando em prim, e no for interno construímos uma aresta do elo corrente para o elo de chave i com a probabilidade definida através da variável valor aleatório, checando se é menor que a probabilidade.

Também há uma checagem para ver se a chave do elo corrente é diferente de i, para não gerar uma aresta, por exemplo, 1 → 1. Também não geramos arestas para 0, com isso, o 0 será garantidamente um elemento sem predecessor.

```java
//O(n^2)
public void geraGrafo(int n, double probabilidade) {
    Random random = new Random();
    // Criar vértices
    for (int i = 0; i < n; i++) {
        adicionarVertice(i);
    }

    // Adicionar arestas com probabilidade p
    for (Elo eloAtual = prim; eloAtual != null; eloAtual = eloAtual.prox) {
        for (int i = 0; i < n; i++) {
            Double valorAleatorio = random.nextDouble();
            if (i != eloAtual.chave && (valorAleatorio < probabilidade) && i!= 0) {
                adicionarAresta(eloAtual.chave, i);
            }
        }
    }

}
```

Método geraGrafo() - complexidade O(N^2)

2.2.1 Vértices

A criação dos vértices é a fase inicial para a solução do problema, visto que é a partir deles que criamos as arestas.

O método adicionaVertice() recebe um valor de chave. Se o prim for nulo, cria um elo com essa chave e o coloca no início da lista. Se não, percorre até o elemento cujo próximo é nulo, armazena o último elemento na variável ant e faz o ant apontar pro novo elo, inserindo assim o novo elo no fim da lista.

```java
//O(N)
public Elo adicionarVertice(int chave) {
    Elo p, novoElo, ant;

    if(prim == null){
        novoElo = new Elo(chave, contador:0, prox:null, listaSuc:null);
        prim = novoElo;
    }else{
        novoElo = new Elo(chave, contador:0, prox:null, listaSuc:null);
        ant = percorrerLista();

        ant.prox = novoElo;
        n++;
    }
    return novoElo;
}
```

Método adiconarVertice() - complexidade O(N)

2.2.2 Arestas

A partir da criação dos vértices, prosseguimos para a criação das arestas juntamente com o estabelecimento da ordenação parcial do conjunto.

O método adicionarAresta() encontra os elos de origem e destino através do método encontrarElo(). Após isso, verifica-se a possibilidade de um ciclo. Por exemplo: ao montar a aresta 8 → 5, é checado se existe a aresta 5 → 8, isso é feito através do método verificarCiclo().

Se não existir ciclo, insiro o eloDestino no início da lista de sucessores do eloOrigem. Com isso, o novo sucessor passa a ser o primeiro elemento na lista de sucessores do eloOrigem.

```java
//O(N + M)
public void adicionarAresta(int origem, int destino) {
    Elo eloOrigem = encontrarElo(origem);
    Elo eloDestino = encontrarElo(destino);

    if (eloOrigem != null && eloDestino != null && !verificarCiclo(eloDestino, eloOrigem)) {
        EloSucessor novoSucessor = new EloSucessor(eloDestino, eloOrigem.listaSuc);
        eloOrigem.listaSuc = novoSucessor;
        eloDestino.contador++;
    }
}
```

Método adiconarAresta() - complexidade O(N+M)

O método verificarCiclo() recebe dois Elos como parâmetro e verifica se existe alguma aresta entre dois elos passados como parâmetro. Ele precede do método de adicionarAresta(), então, quando queremos criar uma aresta (por exemplo, de 8 → 5), precisamos verificar se existe alguma aresta de 5 → 8.

E é isso que o método verificarCiclo() se propõe. Caso exista alguma aresta entre os elos passados como parâmetro, ele retorna true. Caso contrário, retorna false;

```java
//O(S) onde S é o tamanho da lista de sucessores
private boolean verificarCiclo(Elo origem, Elo destino) {
    // Verifica se há um ciclo ao percorrer os sucessores de origem
    for (EloSucessor sucessor = origem.listaSuc; sucessor != null; sucessor = sucessor.prox) {
        if (sucessor.id == destino) {
            return true; // Ciclo encontrado
        }
    }
    return false; // Não há ciclo
}
```

Método adiconarAresta() - complexidade O(M) onde M é o tamanho da lista de sucessores

2.3 Mostrando os códigos

No método inicializar(), declaramos um vetor de inteiro com os valores dos vértices a serem rodados nos testes. Para cada valor do vértice, chamamos o método medirTempo(), que passará o valor corrente do vetor e na linha seguinte terá o *print* do valor corrente e do tempo médio.

```
public void incializar() {
    // Vetor de inteiros fornecido como argumento
    int[] vetorParametro = {10, 20, 30, 40, 50, 100, 200, 500, 1000, 5000, 10000, 20000, 30000, 50000, 100000};

    for (int valor : vetorParametro) {
        double tempoMedio = medirTempoMedio(valor);
        System.out.printf(format:"Para o valor %d: Tempo medio = %.6f segundos %n", valor, tempoMedio);
    }
}
```

Método *inicializar() - Complexidade O(1)*

No método medirTempoMedio() recebemos um valor por parâmetro, declaramos um vetor de tempos para receber os valores dos tempos para cada execução do algoritmo. Criamos uma variável para receber o tempo inicial, outra para receber o tempo final e rodamos o algoritmo.

Após isso, armazenamos na posição i do vetor de tempos. Repetimos o loop para a quantidade de vezes que se deseja rodar o algoritmo e ao final chamamos o método calcularMedia() e convertemos o tempo médio para segundos. No calcularMedia(), recebemos o vetor de tempos, declaramos uma variável soma para receber o valor total dos tempos. No loop, incrementamos a variável soma. Ao finalizar o loop, dividimos o total da soma pelo tamanho do vetor.

```java
public double medirTempoMedio(int valor) {
    // Roda a função 10 vezes e calcula o tempo médio
    long[] tempos = new long[4];
    for (int i = 0; i < 4; i++) {
        long inicio = System.currentTimeMillis();
        geraGrafo(valor, probabilidade:0.13);
        long fim = System.currentTimeMillis();
        tempos[i] = fim - inicio;
    }

    double tempoMedio = calcularMedia(tempos);
    return tempoMedio / 1000.0; // Converte para segundos
}

private static double calcularMedia(long[] tempos) {
    long soma = 0;
    for (long tempo : tempos) {
        soma += tempo;
    }
    return (double) soma / tempos.length;
}
```

Métodos medirTempoMedio() e calcularMedia() - complexidade O(1) e O(N)

No método encontrarElo(), passamos uma chave como parâmetro e iteramos sobre a lista até encontrar o elo que tem a chave que passamos por parâmetro.

```java
//O(n)
public Elo encontrarElo(int chave) {
    for (Elo eloAtual = prim; eloAtual != null; eloAtual = eloAtual.prox) {
        if (eloAtual.chave == chave) {
            return eloAtual;
        }
    }
    return null;
}
```

Método encontrarElo() - complexidade O(N)

No método listaSemPredecessores() criamos um elo auxiliar p para receber o prim e o apontamos para null. Enquanto o p for diferente de null, cada vez que encontrarmos um Elo que possui contador igual a 0, inserimos isso na lista. Fazemos isso até percorrer todos os elementos da lista e com isso criamos uma lista com os elementos sem predecessor.

```
//O(n)
public void listaSemPredecessores(){
    Elo p = new Elo(prim.chave, contador:0, prim.prox, prim.listaSuc);
    Elo antigoPrim;
    prim = null;
    Elo elementoCorrente;

    while(p!= null){
        elementoCorrente = p;
        p = elementoCorrente.prox;
        if(elementoCorrente.contador == 0){
            Elo novoElo = new Elo(elementoCorrente.chave, contador:0, elementoCorrente, elementoCorrente.listaSuc);
            antigoPrim = prim;
            prim = novoElo;
            prim.prox = antigoPrim;
        }
    }
}
```

Método listaSemPredecessor() - complexidade O(N)

No método imprimirElementos(), iteramos sobre a lista de elementos com 0 predecessor montada anteriormente, decrementamos o n, imprimimos a chave e entramos no loop interno.

O loop interno itera sobre a lista de sucessores do elemento corrente, diminui 1 do contador do elemento da lista de sucessores.

Se chegar a 0, remove o elemento em questão da lista de sucessores daquele elemento e insere este mesmo elemento no final da lista de elementos com 0 predecessor.

```java
//O(n^2)
public void imprimirElementos(){

    System.out.println(x:"Elementos sem predecessores");
    Elo aux;
    for(aux = prim; aux != null; aux=aux.prox){
        n--;
        System.out.print(aux.chave + ", ");
        for (EloSucessor sucessor = aux.listaSuc; sucessor != null; sucessor = sucessor.prox) {
            sucessor.id.contador--;
            if(sucessor.id.contador== 0){
                if(aux.listaSuc == sucessor){
                    Elo ult = percorrerLista();
                    ult.prox = sucessor.id;
                    aux.listaSuc = sucessor.prox;
                    sucessor.id.prox = null;

                }else{
                    EloSucessor antListSuc;
                    antListSuc = percorrerListaSucessores(aux.listaSuc, sucessor.id.chave);
                    antListSuc.prox = sucessor.prox;
                    Elo ult = percorrerLista();
                    ult.prox = sucessor.id;
                    sucessor.id.prox = null;

                }
            }
        }
        prim = aux.prox;
    }
}
```

Método imprimirElementos() - complexidade O(N^2)

O método percorrerLista() percorre a lista inteira até chegar no último elemento e o retorna.

```java
public Elo percorrerLista() {
    Elo eloAtual;

    for (eloAtual = prim; eloAtual.prox != null; eloAtual = eloAtual.prox);

    return eloAtual;
}
```

Método percorrerLista() - complexidade O(N)

No método percorrerListaSucessores(), passamos como parâmetro uma referência para o primeiro elemento de uma lista de sucessores e um elemento inteiro. O objetivo é armazenar o elemento anterior ao Elo que possui a chave elem. Esse é um método auxiliar para realizar a remoção de um elemento da lista de sucessores de outro elo.

```
//O(m) onde m é o tamanho da lista de sucessores
public EloSucessor percorrerListaSucessores(EloSucessor inicio, int elem){
    EloSucessor ant = null;
    EloSucessor p;

    for(p = inicio; p.id.chave != elem; p=p.prox){
        ant = p;
    }

    return ant;
}
```

Método percorrerListaSucessores() - Complexidade O(M) onde m é o tamanho da lista de Sucessores

O método realizaLeitura() é o método inicial para a realização da leitura do arquivo, que recebe a string da entrada como parâmetro e acusa erro caso não consiga encontrar esse arquivo.

```
public void realizaLeitura(String nomeEntrada)
{
    try {
        this.lerArquivo(nomeEntrada);
    } catch (Exception err) {
        System.out.println("Erro ao ler o arquivo.");
        err.printStackTrace();
    }
}
```

Método realizaLeitura() - Complexidade O(M), onde M é o número total de linhas

O método lerArquivo() cria um scanner que recebe o arquivo como parâmetro e vai lendo linha a linha, chamando o método lerLinha(), enquanto houver algo para ser lido.

```
private void lerArquivo(String nomeArquivo) throws Exception {
    File file = new File(nomeArquivo);
    Scanner scanner = new Scanner(file);

    while (scanner.hasNextLine()) {
        this.lerLinha(scanner);
    }

    scanner.close();
}
```

Método lerArquivo() - Complexidade O(M), onde M é o número total de linhas

O método lerLinha() realiza a leitura de cada linha do arquivo, a partir de um vetor de elementos. Recebe a chave do predecessor e do sucessor e, a partir disso, inicia o processo de construção das arestas.

```
private void lerLinha(Scanner scanner) {
    String data = scanner.nextLine();
    String[] elementos = data.split( regex: "\\s*<\\s*");

    int predecessorChave = Integer.parseInt(elementos[0]);
    int sucessorChave = Integer.parseInt(elementos[1]);

    Elo predecessor = this.encontrarElo(predecessorChave);
    if (predecessor == null) predecessor = this.adicionarVertice(predecessorChave);

    Elo sucessor = this.encontrarElo(sucessorChave);
    if (sucessor == null) sucessor = this.adicionarVertice(sucessorChave);

    this.adicionarAresta(predecessor.chave, sucessor.chave);
}
```

Método lerLinha() - Complexidade O(N+M), onde M é o número total de linhas e N é o número de elementos no grafo

O método debug() realiza a impressão das Arestas criadas pelo programa seguindo o padrão de formatação de *print* pedido no enunciado.

```
//O(n*m)
private void debug()
{
    for (Elo p = this.prim; p != null; p = p.prox) {
        System.out.print(p.chave + " predecessores: " + p.contador + ", sucessores: ");
        for (EloSucessor s = p.listaSuc; s != null; s = s.prox) {
            System.out.print(s.id.chave + " -> ");
        }
        System.out.println("NULL");
    }
}
```

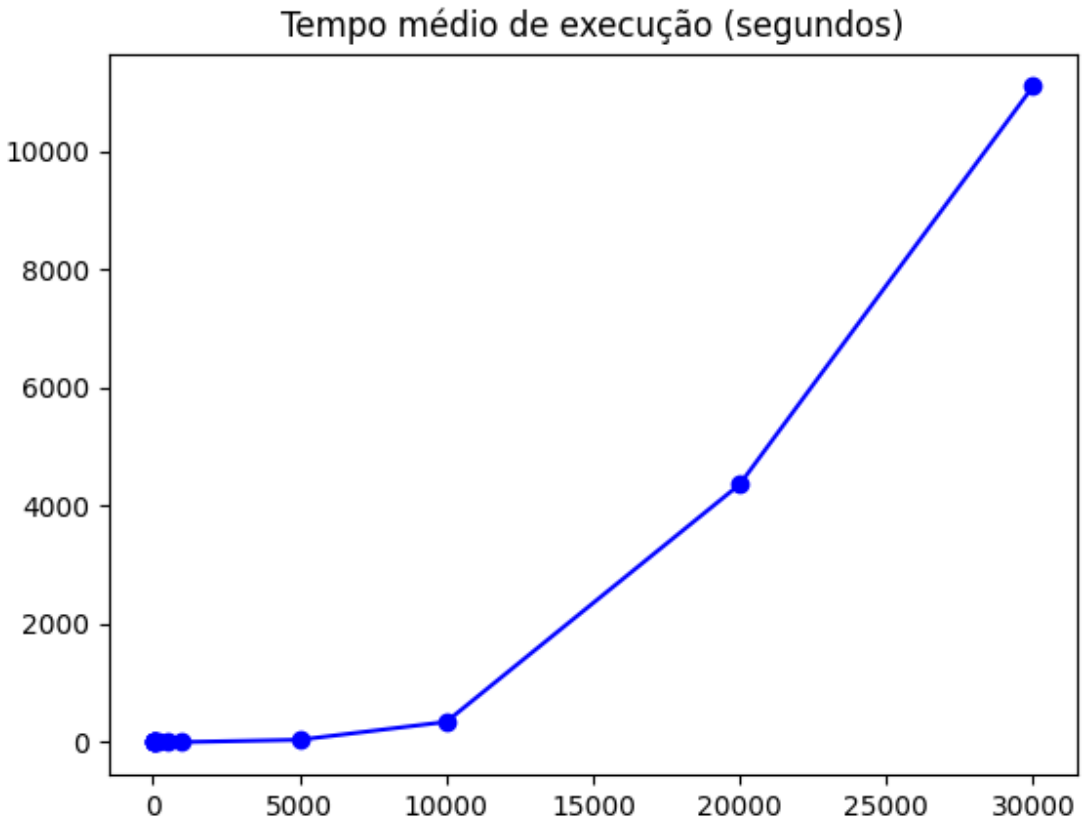
*Método debug() - Complexidade O(M*N) onde M é o tamanho da lista de sucessores e N da lista de principal*

3 Tempos de execução

As medições de tempo foram feitas com o System.currentTimeMillis().

Ambos os integrantes realizaram testes em seus respectivos computadores. A diferença é notável: em um deles, havia um gargalo no início da execução, com o teste de cinco entradas; no outro, esse gargalo não existia, mas o tempo de execução foi maior.

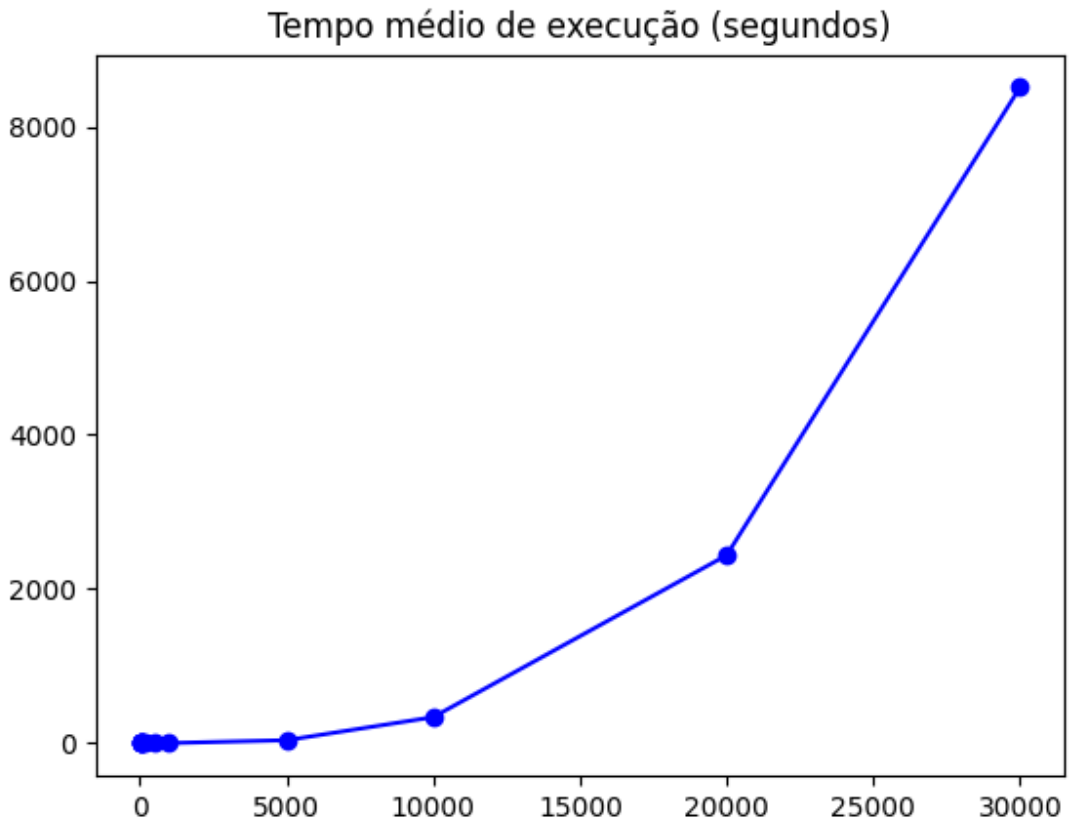
Todas as impressões da codificação foram removidas para que o tempo fosse comparado. A média aritmética final para comparação foi feita com currentTimeMillis(), tornando claro que o tempo por vezes era pequeno demais, próximo de zero.



Eixo y: Tempo em segundos

Eixo x: Tamanho da entrada

Acima, o gráfico do tempo médio de execução - computador de Emerson



Eixo y: Tempo em segundos

Eixo x: Tamanho da entrada

Acima, o gráfico do tempo médio de execução - computador de Bruna

Observação: a conversão para notação científica foi automática, realizada pela biblioteca matplotlib do Python. Verificamos que o ângulo entre as retas e o eixo x fica cada vez maior, isso implica que com o aumento do tamanho da entrada, o tempo de execução aumenta em uma proporção bem maior, e o gráfico vai tendendo a esse comportamento cada vez mais

Configurações

- Computador de Emerson

Nome do dispositivo: DESKTOP-D5I6BEG

Processador: Intel(R) Core(TM) i5-9600K CPU @ 3.70GHz 3.70 GHz

RAM instalada: 32,0 GB (utilizável: 31,9 GB)

Tipo de sistema: Sistema operacional de 64 bits, processador baseado em x64

- Computador de Bruna

Nome do dispositivo: MacBook Pro

Processador: 2,6GHz Intel Core i7 6-Core

RAM instalada: 16GB 2667 MHz DDR4

Tipo de sistema: Sistema operacional de 64 bits, processador baseado em x64

4 Problemas enfrentados

Tivemos bastante dificuldade para tentar realizar o processo de ordenação linear do projeto. Até a parte da criação da lista de elementos com 0 predecessor não tivemos problemas, porém, após isso, não conseguimos pensar em uma solução para conseguir resolver o problema pois não conseguimos identificar a causa real de não estar funcionando da ordenação topológica e com isso afetou a finalização correta do trabalho.

5 Referência bibliográficas

https://acervolima.com/como-criar-um-grafico-aleatorio-usando-a-geracao-de-borda-aleatoria-em-java/#google_vignette

https://acervolima.com/modelo-erdos-renyl-para-gerar-graficos-aleatorios/