Pontifícia Universidade Católica de Minas Gerais

Algoritmos em Grafos

Trabalho Prático

Aluno(s): Jonathan Douglas Diego Tavares
Professores(as): Alexei Machado

Novembro
2020

1 Introdução

Na teoria de grafos, o problema do caminho mínimo consiste na minimização do custo de travessia de um grafo entre dois nós (ou vértices); custo este dado pela soma dos pesos de cada aresta percorrida.

O problema de encontrar o menor caminho em um grafo pode ser utilizado para resolver diversos problemas, distintos do problema que originou a sua concepção. Dependendo do significado do peso das arestas, é possível utilizar o algoritmo de Dijkstra e outras variações para resolver diversos outros problemas que envolvem minimização em contextos próximos ao do original.

O problema do Caixeiro Viajante (em inglês travelling salesman problem, travelling salesperson problem ou TSP) consiste na seguinte pergunta: "Dada uma lista de aeroportos e as distâncias entre cada par de aeroportos, é possível traçar uma rota que visita cada aeroporto e retorna ao aeroporto inicial?". Este é um problema NP difícil em Otimização Combinatória, importante em Pesquisas Operacionais e Ciência da Computação Teórica.

O problema foi formulado pela primeira vez em 1930 e é um dos problemas mais intensamente estudados em otimização. É usado como referência para muitos métodos de otimização. Embora o problema seja computacionalmente difícil, é conhecido um grande número de heurísticas e algoritmos exatos, para que algumas instâncias com dezenas de milhares de aeroportos possam ser resolvidas completamente e até problemas com milhões de aeroportos possam ser aproximados em uma pequena fração de 1%.

O TSP tem várias aplicações, mesmo em sua formulação mais pura, como planejamento, logística e fabricação de microchips. Ligeiramente modificado, ele aparece como um subproblema em muitas áreas, como o seqüenciamento de DNA. Nessas aplicações, o aeroporto conceitual representa, por exemplo, clientes, pontos de solda ou fragmentos de DNA, e a distância do conceito representa tempos ou custos de viagem, ou uma medida de similaridade entre fragmentos de DNA. O TSP também aparece na astronomia, pois os astrônomos que observam muitas fontes desejam minimizar o tempo gasto movendo o telescópio entre as fontes. Em muitos aplicativos, restrições adicionais, como recursos limitados ou janelas de tempo, podem ser impostas.

2 História

As origens do problema do caixeiro viajante não são claras. Um manual para vendedores ambulantes de 1832 menciona o problema e inclui exemplos de passeios pela Alemanha e Suíça, mas não contém tratamento matemático.

O problema do caixeiro viajante foi matematicamente formulado em 1800 pelo matemático irlandês W.R. Hamilton e pelo matemático britânico Thomas Kirkman. O jogo Icosian de Hamilton era um quebra-cabeça recreativo baseado na descoberta de um ciclo hamiltoniano. A forma geral do TSP parece ter sido estudada pela primeira vez por matemáticos durante a década de 1930 em Viena e em Harvard, principalmente por Karl Menger, que define o problema, considera o algoritmo óbvio de força bruta e observa que pode não haver resultado ótimo na heurística do vizinho mais próximo.

Foi considerado matematicamente pela primeira vez na década de 1930 por Merrill M. Flood, que procurava resolver um problema de roteamento de ônibus escolar. Hassler Whitney, da Universidade de Princeton, introduziu o nome problema de caixeiro viajante logo depois.

Nas décadas seguintes, o problema foi estudado por muitos pesquisadores de matemática, ciência da computação, química, física e outras ciências. Na década de 1960, no entanto, foi criada uma nova abordagem, que, em vez de buscar soluções ótimas, seria possível produzir uma solução cujo comprimento é comprovadamente limitado por um múltiplo do tamanho ideal e, desta forma, criar limites mais baixos para o problema; estes podem então ser usados com abordagens ramificadas e vinculadas. Um método para fazer isso era criar uma árvore geradora mínima do grafo e, em seguida, dobrar todas as suas arestas, o que produz o limite de que a duração de um caminho ideal é no máximo o dobro do peso de uma árvore geradora mínima.

Richard M. Karp mostrou em 1972 que o problema do ciclo hamiltoniano era NP completo, o que implica no TSP ser um problema NP difícil. Isso forneceu uma explicação matemática para a aparente dificuldade computacional de encontrar caminhos ideais.

Nos anos 90, Applegate, Bixby, Chvátal e Cook desenvolveram o programa Concorde, que tem sido usado em muitas soluções de registros recentes. Gerhard Reinelt publicou o TSPLIB em 1991, uma coleção de instâncias de referência de dificuldade variável, que tem sido usada por muitos grupos de pesquisa para comparar resultados. Em 2006, Cook e outros computaram um caminho ideal por uma instância de 85.900 aeroportos, dada por um problema de layout de microchips, atualmente a maior instância TSPLIB resolvida. Para muitos outros casos com milhões de aeroportos, é possível encontrar soluções que garantem estar entre 2 e 3% de um caminho ideal.

3 Complexidade

O algoritmo de Dijkstra, concebido pelo cientista da computação holandês Edsger Dijkstra em 1956 e publicado em 1959, soluciona o problema do caminho mais curto num grafo dirigido ou não dirigido com arestas de peso não negativo, em tempo computacional O(m + n log n) onde m é o número de arestas e n é o número de vértices. O algoritmo que serve para resolver o mesmo problema em um grafo com pesos negativos é o algoritmo de Bellman-Ford, que possui maior tempo de execução que o Dijkstra.

O algoritmo considera um conjunto S de menores caminhos, iniciado com um vértice inicial I. A cada passo do algoritmo busca-se nas adjacências dos vértices pertencentes a S aquele vértice com menor distância relativa a I e adiciona-o a S e, então, repetindo os passos até que todos os vértices alcançáveis por I estejam em S. Arestas que ligam vértices já pertencentes a S são desconsideradas.

O problema do caixeiro é um clássico exemplo de problema de otimização combinatória. A primeira coisa que podemos pensar para resolver esse tipo de problema é o reduzir a um problema de enumeração: achamos todas as rotas possíveis e, usando um computador, calculamos o comprimento de cada uma delas e então vemos qual a menor. Se acharmos todas as rotas estaremos fazendo uma contagem, daí podermos dizer que estamos reduzindo o problema de otimização a um de enumeração.

Para acharmos o número R(n) de rotas para o caso de n aeroportos, basta fazer um raciocínio combinatório simples e clássico. Por exemplo, no caso de n = 4 aeroportos, a primeira e última posição são fixas, de modo que elas não afetam o cálculo; na segunda posição podemos colocar qualquer uma das 3 aeroportos restantes B, C e D, e uma vez escolhida uma delas, podemos colocar qualquer uma das 2 restantes na terceira posição; na quarta posição não teríamos nenhuma escolha, pois sobrou apenas um aeroporto; consequentemente, o número de rotas é 3 x 2 x 1 = 6, resultado que fora obtido antes contando diretamente a lista de rotas acima.

De modo semelhante, para o caso de n aeroportos, como a primeira é fixa, o número total de escolhas que podemos fazer é:

$$(n-1) \times (n-2) \times \ldots \times 2 \times 1 = (n-1)!$$

Figura 1: Escolhas com n rotas

Logo podemos dizer que a complexidade do problema partindo desta abordagem é de O(n!).

4 Implementação

A implementação foi feita através da construção de um projeto no Eclipse. Foram implementadas 17 classes que estão contidas dentro de 4 pacotes, sendo eles "UI", "Modelagem", "DAO"e "Algoritmos". É possível visualizar quais são as classes implementadas na figura abaixo.

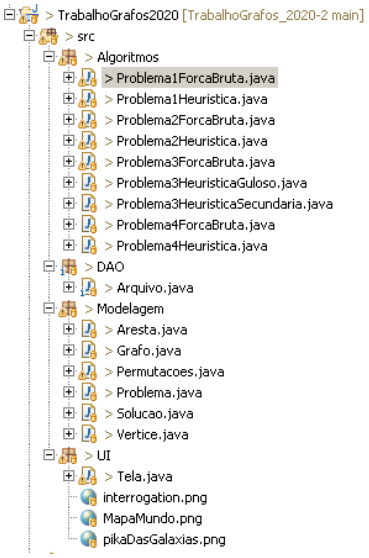

Figura 2: Classes no projeto TrabalhoGrafos2020

4.1 Classe Tela

A classe Tela é a classe principal, responsável por realizar a chamada dos algoritmos que irão resolver os problemas propostos. O método main é responsável por inicializar a UI, que permitirá ao usuário interagir e executar os algoritmos sobre os grafos que desejar. Além do método main, há um conjunto de outros métodos que tem por objetivo realizar as funcionalidades dos componentes presentes na interface e também realizar a chamada dos métodos que solucionarão os problemas disponíveis. Foram resolvidos os 3 primeiros problemas utilizando uma versão em força bruta e outra baseada em heurística construtiva ("Vizinho mais próximo"). As imagens abaixo mostram como está a interface do usuário e como é realizada, em código, a chamada para a execução da solução de um dos algoritmos criados para solucionar os problemas. Os testes foram baseados em instâncias disponíveis no TSPLIB.

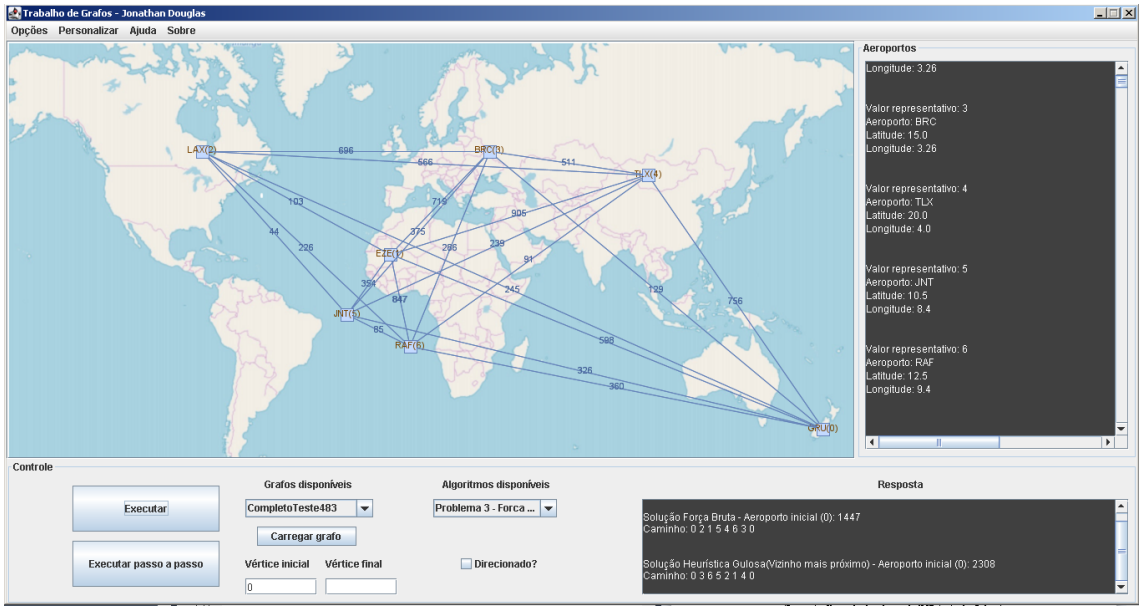

Figura 3: User Interface demonstrando um teste em grafo completo para o problema do Caixeiro Viajante(viagem ao redor do mundo)

```
/*
 * ************************************ Solução Heurística Gulosa Vizinho mais
 * próximo
 *
 * Comentar trecho de código caso não vá utilizar o algoritmo
 ************************************
 */
tempoInicial = System.currentTimeMillis();

solucao = guloso.getSolucao(aeroportoInicial);

tempoFinal = System.currentTimeMillis();

// habilitar a linha abaixo caso deseje ver o caminho o obtido pela solução
solucao.mostrarCaminho();

txtResposta.setText(
        txtResposta.getText() + "\n\nSolução Heurística Gulosa(Vizinho mais próximo) - Aeroporto inicial ("
                + aeroportoInicial + "): " + solucao.getPrecoTotal() + "\n" + solucao.mostrarCaminho());

System.out.printf("Tempo total de execução: %.3f ms%n", (tempoFinal - tempoInicial) / 1000d);
System.out.println("");
```

Figura 4: Chamada de solução do TSP (método Guloso)

O teste de tempo é feito capturando-se o tempo atual do sistema na variável tempoInicial. É feita a chamada do método getSolucao (na imagem através de "guloso.getSolucao(aeroportoInicial))"que devolve um objeto do tipo Solucao para a variável solucao. Esta variável contém os dados gerados pela solução do TSP (na imagem a solução é obtida pelo algoritmo guloso). Uma vez obtido os dados da solução do TSP, obtêm-se o tempo atual do sistema na variável tempoFinal e é impresso o resultado obtido na solução e o tempo total de execução. Este padrão de implementação segue para os demais métodos de resolução do TSP que foram implementados.

A escolha do uso de instâncias providas via arquivo ou instâncias aleatórias também é feita através da interface, sendo possível gerar grafos diversos, através da geração de matrizes de adjacência que simulam grafos de vários tipos. Tudo isso é possível através da geração de matrizes de adjacência aleatórias, sendo somente necessário fornecer os aeroportos antes que se possa dar início a geração das matrizes com um único clique.

4.2 Classe Arquivo

A classe Arquivo é a classe responsável por processar e obter os dados contidos nos arquivos .tsp (instâncias de teste) fornecidos junto a especificação do trabalho. A classe possui três métodos que se chamam getGrafoNaoOrientado, getGrafoOrientado e getMatrizFromFile. Estes métodos atuam abrindo o arquivo e através de uma lógica implementada identifica os dados relevantes como a dimensão da matriz de adjacência, a partir de onde se iniciam os dados da matriz e os próprios dados da matriz. O método getMatrizFromFile, especializado em abrir arquivos do TSPLIB, contém duas formas de preencher a matriz de acordo com os arquivos fornecidos, que é através do preenchimento da diagonal superior ou inferior e posteriormente é feito um espelhamento.

4.3 Classe Grafo

A classe Grafo implementa um grafo no formato de matriz de adjacência. Ela possui métodos que permitem setar os dados do grafo e os recuperar uma vez que a matriz de adjacência esteja pronta. Vale ressaltar que a matriz foi construída tendo cada posição representando uma "Aresta", pois uma aresta pode assumir diversos tipos de significado. Seu conjunto de atributos é:

```
/**
 * Classe Grafo
 *
 * @author Jonathan Douglas Diego Tavares
 */
public class Grafo {

    // Atributos
    private String nomeGrafo;
    private final int numeroVertices; // aka numero de aeroportos
    private Aresta[][] matrizAdjacencia;

    /**
     * MÃ©todo construtor, responsÃ¡vel por inicializar o nÃºmero de vÃ©rtices e a
     * matriz de adjacÃªncia
     *
     * @param vertices quantidade de vÃ©rtices do grafo
     */
    public Grafo(int vertices) {
        numeroVertices = vertices;
        matrizAdjacencia = new Aresta[numeroVertices][numeroVertices];
        inicializaMatrizAdj();
    }
```

Figura 5: Conjunto de atributos da classe Grafo e seu método construtor

4.4 Classe Vertice

A classe Vertice implementa a representação do vértice(aeroporto) na modelagem escolhida. É possível visualizar abaixo todos os seus atributos, bem como a inicialização dos mesmos.

```
public class Vertice {

    // Atributos
    private String labelVertice;
    private int valorRepresentativo;
    private int grau;
    private double latitude;
    private double longitude;

    public Vertice(int valorRepresentativo, double latitude, double longitude) {
        this.labelVertice = "NoLabel";
        this.valorRepresentativo = valorRepresentativo;
        this.grau = 0;
        this.latitude = latitude;
        this.longitude = longitude;
    }

    public Vertice(String labelVertice, int valorRepresentativo, double latitude, double longitude) {
        this.labelVertice = labelVertice;
        this.valorRepresentativo = valorRepresentativo;
        this.grau = 0;
        this.latitude = latitude;
        this.longitude = longitude;
    }
```

Figura 6: Conjunto de atributos da classe Vertice e seu método construtor

4.5 Classe Aresta

A classe Aresta implementa a representação das arestas(trajetos) na modelagem escolhida. É possível visualizar abaixo todos os seus atributos, bem como a inicialização dos mesmos.

```
public class Aresta {
    private Vertice v1;
    private Vertice v2;
    private int altitude;
    private double distancia;
    private int preco;

    public Aresta(Vertice v1, Vertice v2) {
        this.v1 = v1;
        this.v2 = v2;
        this.altitude = 10000;
        this.distancia = -1;
        this.preco = -1;
    }

    public Aresta(Vertice v1, Vertice v2, int altitude, double distancia, int preco) {
        this.v1 = v1;
        this.v2 = v2;
        this.altitude = altitude;
        this.distancia = distancia;
        this.preco = preco;
    }
```

Figura 7: Conjunto de atributos da classe Aresta e seu método construtor

4.6 Classe Problema

A classe Problema implementa a ideia de ter a instância do problema como um objeto que pode fornecer informações sobre a instância. Possui como atributo apenas uma variável do tipo Grafo, sendo capaz de retornar está variável e fornecer dados sobre a o valor do peso de uma determinada aresta do grafo.

4.7 Classe Solucao

A classe Solucao implementa a ideia de ter a solução como um objeto capaz de ser construído com os dados obtidos da execução dos algoritmos de resolução do TSP. Sendo assim, um objeto desta classe é capaz de retornar dados acerca da solução obtida como distância total da solução e o caminho. Seu conjunto de atributos é:

```
public class Solucao {

    // Atributos
    private double distanciaTotal; // distÃ¢ncia obtida do caminho soluÃ§Ã£o
    private int altitudeTotal;
    private int precoTotal;
    private ArrayList<Integer> caminho; // caminho soluÃ§Ã£o

    /**
     * Construtor
     */
    public Solucao() {
        this.distanciaTotal = 0;
        this.altitudeTotal = 0;
        this.precoTotal = 0;
        caminho = new ArrayList<>();
    }
```

Figura 8: Conjunto de atributos da classe Solucao

4.8 Classe Permutacoes

A classe Permutacoes implementa a operação responsável por gerar as permutações que serão utilizadas na resolução do TSP através do algoritmo de força bruta. Possui um método principal responsável por gerar as permutações. Na imagem imagem abaixo é possível ver a implementação deste método:

```
/**
 * Faz recursivamente as permutações a partir de uma permutação inicial
 * @param vet permutação inicial
 * @param n tamanho do vetor permutado atual
 */
private void permuta(int []vet, int n) {

    if (n==vet.length) {
        imprime();
    } else {

        for (int i=0; i < vet.length; i++) {

            boolean achou = false;

            for (int j = 0; j < n; j++) {

                if (permutacaoAtual[j]==vet[i]) achou = true;
            }

            if (!achou) {

                permutacaoAtual[n] = vet[i];

                permuta(vet,n+1);
            }

        }

    }
}
```

Figura 9: Método permuta

O método é implementado explorando a ideia da árvore de recursão. O número de permutações é dado por n!, em que n é a quantidade de elementos a serem permutados. Logo há n! caminhos que podem ser percorridos gerando n! combinações possíveis.

Visualizando cada combinação como um conjunto de gavetas que podem ser preenchidas, a variável n representa a gaveta atual e também a quantidade de elementos já preenchidos. O método executa um laço for que varrerá o vetor "vet" do primeiro ao último elemento. Um dado objeto será posto na primeira gaveta (que consiste na posição 0 de "permutacaoAtual"), e após preencher a primeira gaveta, é feita uma chamada recursiva para "permuta", mas agora passando para o nível n=1 (segunda gaveta). O laço for se encarrega de colocar na segunda gaveta um elemento diferente daquele que está na primeira e, novamente, faz a chamada para "permuta", dessa vez para tratar o nível n igual a 2 (terceira gaveta) que deverá conter um valor diferente do presente na primeira e na segunda gaveta. O método segue sua execução até que a última gaveta seja preenchida (caso que ocorre quando n equivale a vet.length). Nesse caso, a base da recursão é atingida e possível armazenar a permutação atual para posterior uso.

4.9 Classes de resolução dos problemas - Algoritmos

4.9.1 Força bruta

As classes que resolvem os problemas utilizando uma abordagem de força bruta estão nomeadas da seguinte forma "ProblemaXForcaBruta", em que X é o número dado ao problema resolvido. A numeração dos problemas segue a ordem presente na especificação. Em geral os algoritmos implementados utilizam a técnica de força bruta para resolução do problemas. Seu funcionamento é através da geração de todas as possibilidades de caminhos existentes partindo de um aeroporto inicial e na posterior verificação daquela possibilidade que possui o menor custo. Um exemplo de código fonte implementado pode ser visto na imagem abaixo:

```
/**
 * Calcula o menor caminho comparando a distÃ¢ncia total obtida para cada
 * permutaÃ§Ã£o gerada
 *
 * @param aeroportoInicial
 */
private void calculaMenorCaminho(int aeroportoInicial) {
    // gera todas as permutaÃ§Ãµes
    caminhos = permutacoes.getPermutacoes(geraPermutacaoInicial(aeroportoInicial));
    //imprime(caminhos);
    // completa os caminhos de forma a ser possÃ-vel sair da aeroporto inicial e
    // voltar
    // para a mesma
    // ArrayList<int[]> caminhosDesejados = removerPermIndesejada(caminhos);
    ArrayList<int[]> caminhosCompletos = completarCaminhos(aeroportoInicial, caminhos);

    int menorCaminho[] = caminhosCompletos.get(0);
    ArrayList<Integer> menorCaminhoList = new ArrayList<>();
    // imprime(caminhosCompletos);

    // percorre o conjunto de caminhos comparando suas distÃ¢ncias totais
    // o menor caminho serÃ¡ aquele cuja distÃ¢ncia total tenha o menor valor dentre
    // o conjunto verificado
    for (int i = 1; i < caminhosCompletos.size(); i++) {
        if (getPrecoCaminho(caminhosCompletos.get(i)) < getPrecoCaminho(menorCaminho)) {
            menorCaminho = caminhosCompletos.get(i);
        }
    }

    // passa os vÃ©rtices obtidos do menor caminho para um ArrayList que serÃ¡
    // passado para a soluÃ§Ã£o
    for (int i = 0; i < menorCaminho.length; i++) {
        menorCaminhoList.add(menorCaminho[i]);
    }

    // seta o melhor caminho obtido na soluÃ§Ã£o
    solucao.setCaminho(menorCaminhoList);
    // solucao.mostrarCaminho();
    // seta a distÃ¢ncia total obtida no melhor caminho
    solucao.setPrecoTotal(getPrecoCaminho(menorCaminho));
}
```

Figura 10: Método calculaMenorCaminho para o algoritmo de força bruta no problema da volta ao redor do mundo

O método tem início na obtenção das permutações (possibilidades de caminhos). Admite-se que o melhor caminho é o primeiro do conjunto de caminhos e então inicia-se um laço for que irá verificar se este caminho é de fato o melhor ou há algum outro cujo custo seja menor, portanto melhor.

Obtido o melhor caminho, é setado na solução o valor da distância obtida neste caminho e o caminho em si. O resultado é retornado no formato de um objeto do tipo Solucao.

4.9.2 Heurística

A heurística utilizada em todos os casos implementa uma técnica construtiva, baseada em uma estratégia gulosa que é o vizinho mais próximo. Partindo de um aeroporto inicial, a ideia é adicionar ao caminho solução aquele aeroporto cuja distância seja a menor a partir do aeroporto atual. Repete-se este passo até que a quantidade de aeroportos do caminho solução seja equivalente a quantidade de aeroportos do problema inicial. Ao fim adiciona-se o custo do retorno para o aeroporto inicial e obtêm-se a solução final. Um dos códigos fonte implementado pode ser visto nas imagens abaixo:

```java
/**
 * Calcula o melhor percurso partindo de uma determinada aeroporto inicial
 *
 * @param aeroportoInicial
 */
private void calculaMenorCaminho(int aeroportoInicial) {
    ArrayList<Integer> caminhoTemp = new ArrayList<>(); // carregarÃ¡ o melhor caminho obtido
    caminhoTemp.add(aeroportoInicial); // adiciona a aeroporto inicial ao inÃ-cio do percurso

    // utiliza o conceito de aeroporto atual (vÃ©rtice atual) e
    // aeroporto mais prÃ³xima(vÃ©rtice cuja aresta tem menor peso a partir do
    // vÃ©rtice
    // atual)
    int aeroportoAtual = aeroportoInicial;
    int aeroportoMaisProximo;
    int precoTotal = 0; // distancia total obtida no percurso
    int verticeDeMenorAresta; // vÃ©rtice cujo peso Ã© o menor a patir do vÃ©rtice atual
    int nAeroportos = problema.getGrafo().numVertices(); // obtÃ©m a quantidade de aeroportos do problema

    // enquanto o caminho nÃ£o houver a mesma quantidade de aeroportos do problema
    // inicial
    while (caminhoTemp.size() < nAeroportos) {
        // obtÃ©m a lista de adjacÃªncia do vÃ©rtice atual (aeroporto atual)
        ArrayList<Integer> listaAdj = problema.getGrafo().listaDeAdjacencia(aeroportoAtual);

        // retira todas as aeroportos jÃ¡ percorridas da lista de adjacÃªncia
        for (Integer i : caminhoTemp) {
            listaAdj.remove(i);
        }

        // obtÃ©m a aeroporto mais prÃ³xima da aeroporto atual
        verticeDeMenorAresta = getVerticeComArestaDeMenorPeso(aeroportoAtual, listaAdj);
        aeroportoMaisProximo = verticeDeMenorAresta;

        // System.out.println("DistanciaEscolhida: " +
        // problema.getDistancia(cidadeAtual, cidadeMaisProxima));
        // incrementa a distÃ¢ncia total
        precoTotal = precoTotal + problema.getPreco(aeroportoAtual, aeroportoMaisProximo);
```

Figura 11: Método calculaMenorCaminho para o algoritmo de heurística gulosa na resolução da volta ao redor do mundo

```java
        // adiciona a aeroporto mais prÃ³xima ao percurso
        caminhoTemp.add(aeroportoMaisProximo);

        // seta a aeroporto atual como a cidade mais prÃ³xima obtida na soluÃ§Ã£o
        aeroportoAtual = aeroportoMaisProximo;
    }

    // adiciona o custo da volta da Ãºltima aeroporto para o aeroporto inicial
    precoTotal = precoTotal + problema.getPreco(aeroportoAtual, caminhoTemp.get(0));
    // adiciona a volta para o aeroporto inicial
    caminhoTemp.add(aeroportoInicial);

    // seta o melhor percurso obtido
    solucao.setCaminho(caminhoTemp);
    // solucao.mostrarCaminho();
    // seta a distÃ¢ncia total obtida para o melhor percurso
    solucao.setPrecoTotal(precoTotal);
}
```

Figura 12: Método calculaMenorCaminho para o algoritmo de heurística gulosa na resolução da volta ao redor do mundo

O método tem início adicionando ao caminho o aeroporto inicial. Em seguida, marca-se o aeroporto atual como o aeroporto inicial e inicia-se um laço while que somente encerrará quando o caminho solução contiver a mesma quantidade de aeroportos do problema. A cada iteração do laço while é obtida a lista

de adjacência do aeroporto atual e desta são removidos os aeroportos já presentes no caminho solução para que não haja o problema de retorno. No próximo passo se obtém o aeroporto cuja distância é a menor a partir do atual, então é feita a adição deste aeroporto ao caminho solução e diz que este agora é o atual.

Uma vez terminado o laço while, adiciona-se a volta para o aeroporto inicial e incrementa o custo da volta para o aeroporto inicial a partir do último inserido no caminho solução. Obtido o melhor caminho, é setado na solução o valor obtida para o problema em questão. O resultado é retornado no formato de um objeto do tipo Solucao.

5 Experimentos e análise de resultados

5.1 Configurações do Computador

Windows 7 Ultimate 64 bits (6.1, Compilação 7601) (8GB RAM / Processador AMD FX-8320e em 3455.84 MHz

5.2 Força Bruta - em média dos tempos dos problemas analisados

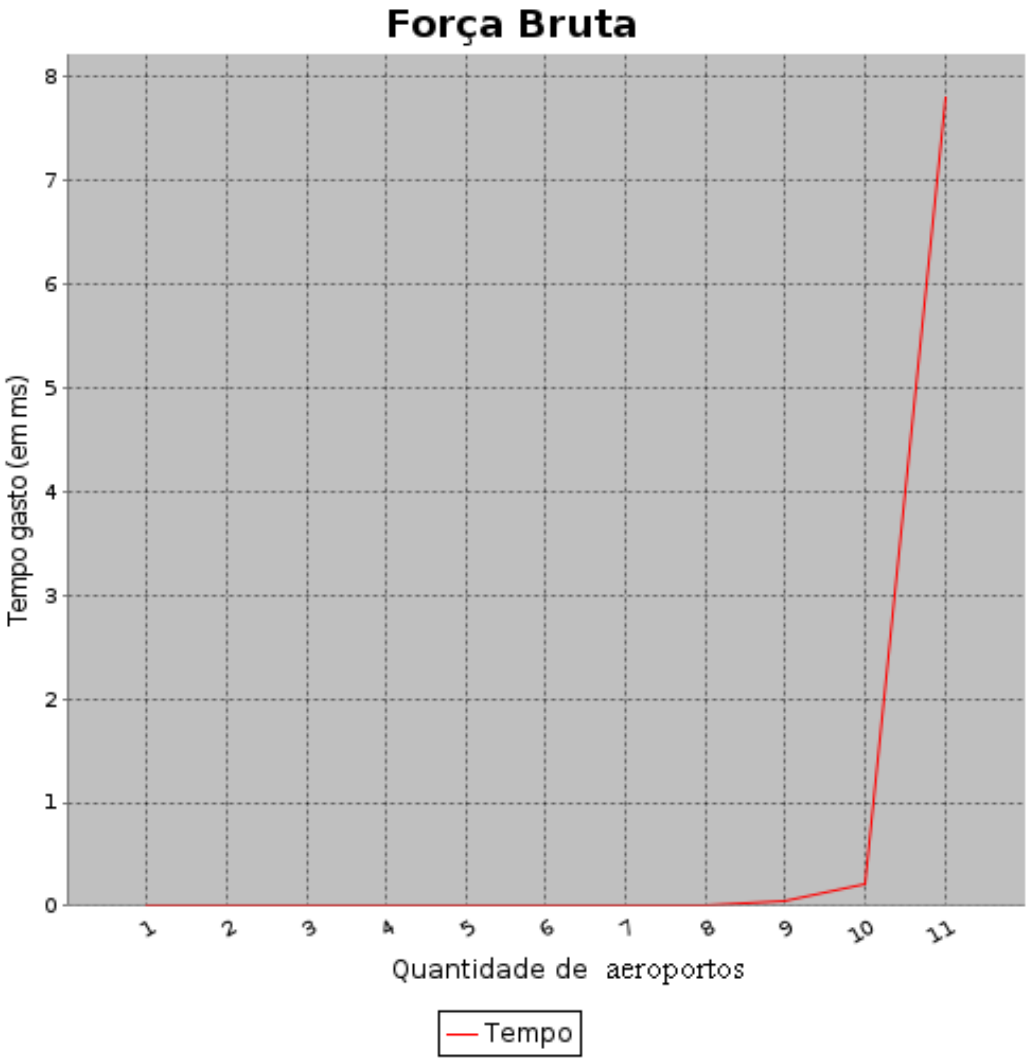

Figura 13: Gráfico Força Bruta

Os testes foram realizados utilizando-se instâncias geradas aleatoriamente com número de aeroportos crescente de 1 até 11. É possível visualizar no gráfico que a medida em que o número de aeroportos cresce de 1 até 10, o crescimento do tempo gasto é ínfimo (cerca de 0,200ms de 1 aeroporto para 10

aeroportos). Já para 11 aeroportos o tempo gasto sobe para mais de 7 segundos, o que significa que houve um crescimento de pelo menos 39 vezes em relação ao tempo gasto para 10 aeroportos. O teste para mais de 11 aeroportos é inviável.

5.3 Heurísticas - em média dos tempos dos problemas analisados

Considerando o limite viável para execução de 5 minutos, a heurística gulosa (vizinho mais próximo) foi capaz de atingir o limite viável de 13000 aeroportos. Vale lembrar que a heurística gulosa é da ordem de $O(n^2)$. Já a outra heurística implementada (inserção mais barata) foi capaz de atingir o limite viável entre 300 e 400 aeroportos. Vale lembrar que esta é da ordem de $O(n^3)$.

Os testes com as instâncias .tsp fornecidas somente foram possíveis de serem realizadas com a heurística gulosa. Segue abaixo os resultados obtidos:

Método	pa561	si535	si1032
Vizinho mais próximo	0,088 ms	0,087 ms	0,306 ms
	3422	50144	94571

Figura 14: Tabela Guloso

Na tabela acima é possível visualizar os tempos obtidos para cada instância e abaixo dos tempos o valor obtido para tarifa total do caminho gerado.

Segundo o TSPLIB, os melhores resultados obtidos para cada instância foram:

1. pa561, 2763
2. si535, 48450
3. si1032, 92650

Como o limite viável para o inserção mais barata foi muito abaixo da quantidade de aeroportos dos arquivos .tsp, para poder comparar as duas heurísticas foram utilizadas instâncias menores, dentro do viável para a heurística da inserção mais barata. Segue abaixo a tabela que compara os dois métodos com geração de instâncias randômicas:

Método	100	200	300	400
Vizinho mais próximo	0,007 ms	0,026 ms	0,036 ms	0,055 ms
Inserção mais barata	0,772 ms	19,676 ms	162,999 ms	793,341 ms
Distâncias				
Vizinho mais próximo	2202	4473	3360	2844
Inserção mais barata	3753	2598	5424	5792

Figura 15: Tabela Guloso e Inserção

Comparando os dois métodos vemos que o inserção mais barata obtém resultados piores em termos de tempo e distância para praticamente todos os casos, exceto para 200 aeroportos em que a tarifa foi melhor.

6 Conclusão

Foi comprovado que o método de força bruta é inviável a medida que o número de instâncias cresce, tendo a quantidade de 11 aeroportos como limite viável no testes realizados.

A heurística gulosa apresentou resultados satisfatórios nos testes realizados com as instâncias .tsp fornecidas. Pode-se ver que os resultados obtidos foram próximos dos melhores valores conhecidos para as instâncias, segundo o TSPLIB.

A heurística da inserção mais barata conseguiu apresentar resultados satisfatórios para instâncias pequenas, mas a medida que a instâncias crescem os resultados pioram muito.

Foi muito bom realizar este trabalho, ele permitiu a obtenção de um aprendizado mais profundo sobre dois dos problemas mais importantes da Computação, que são o TSP e o Shortest Path. Desenvolver os algoritmos para solucionar o problema levaram a necessidade de trabalhar com o modelo de grafo para representação do problema aprimorando ainda mais os conceitos vistos nas aulas.

Referências

[1] Wikipédia. Travelling salesman problem. Visitado em 11/20. Disponível em https://en.wikipedia.org/wiki/Travelling_salesman_problem.

[2] Wikipédia. Shortest path problem. Visitado em 11/20. Disponível em https://en.wikipedia.org/wiki/Shortest_path_problem

[3] Wikipédia. Dijkstra's algorithm. Visitado em 11/20. Disponível em https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

[4] IntechOpen. Traveling Salesman Problem: an Overview of Applications, Formulations, and Solution Approaches. Visitado em 11/20. Disponível em https://www.intechopen.com/books/traveling-salesman-problem-theory-and-applications/traveling-salesman-problem-an-overview-of-applications-formulations-and-solution-approaches