Centro Federal de Educação Tecnológica de Minas Gerais

Laboratório e Teoria de Algoritmos e Estrutura de Dados 2

Problema do Caixeiro Viajante

Aluno(s): Jonathan Douglas Diego Tavares
Professores(as): Thiago de Souza Rodrigues
Amadeu Almeida Coco

Dezembro
2019

1 Introdução

O problema do Caixeiro Viajante (em inglês travelling salesman problem, travelling salesperson problem ou TSP) consiste na seguinte pergunta: "Dada uma lista de cidades e as distâncias entre cada par de cidades, é possível traçar uma rota que visita cada cidade e retorna a cidade 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 cidades possam ser resolvidas completamente e até problemas com milhões de cidades 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, a cidade 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 cidades, dada por um problema de layout de microchips, atualmente a maior

instância TSPLIB resolvida. Para muitos outros casos com milhões de cidades, é possível encontrar soluções que garantem estar entre 2 e 3% de um caminho ideal.

3 Complexidade

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 cidades, basta fazer um raciocínio combinatório simples e clássico. Por exemplo, no caso de n = 4 cidades, a primeira e última posição são fixas, de modo que elas não afetam o cálculo; na segunda posição podemos colocar qualquer uma das 3 cidades 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 uma cidade; 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 cidades, 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 Aplicações

4.1 Perfuração de placas de circuito impresso

Uma aplicação direta do TSP está no problema de perfuração de placas de circuito impresso (PCBs). Para conectar um condutor em uma camada com um condutor em outra camada, ou para posicionar os pinos dos circuitos integrados, é necessário fazer furos através do borda. Os furos podem ter tamanhos diferentes. Para fazer dois furos de diâmetros diferentes consecutivamente, a cabeça da máquina deve se mover para uma caixa de ferramentas e alterar a perfuração equipamento.

Este problema de perfuração pode ser visto como uma série de TSPs, um para cada furo diâmetro, onde as 'cidades' são a posição inicial e o conjunto de todos os furos que podem ser perfurados com uma e a mesma broca. A 'distância' entre duas cidades é dada pelo tempo que leva para mova a cabeça de perfuração de uma posição para outra. O objetivo é minimizar o tempo de viagem para a cabeça da máquina.

4.2 Fiação do computador

Um caso de aplicação especial é a conexão de componentes em um placa de computador. Os módulos estão localizados em uma placa de computador e um determinado subconjunto de pinos deve estar conectado. Ao contrário do caso usual em que uma conexão em árvore Steiner é desejada, aqui o requisito é que não mais que dois fios sejam conectados a cada pino. Portanto, temos o problema de encontrar um caminho hamiltoniano mais curto com pontos de início e termino não especificados. Uma situação semelhante ocorre para a chamada fiação testbus. Para testar placa fabricada é preciso

realizar uma conexão que entra na placa em algum ponto especificado, executa através de todos os módulos e termina em algum ponto especificado. Para cada módulo também há um ponto de entrada e saída especificado para esta fiação de teste. Esse problema também equivale a resolver um problema de caminho hamiltoniano com a diferença de que as distâncias não são simétricas e que o ponto inicial e final são especificados.

4.3 Plotagem de máscara na produção de PCB

Para a produção de cada camada de uma placa de circuito impresso, bem como para camadas de dispositivos semicondutores, uma máscara fotográfica deve ser produzida. No nosso caso, placas impressas de circuito são feitas por um dispositivo de plotagem mecânica. O plotter move uma lente sobre um placa de vidro revestido fotossensível. O obturador pode ser aberto ou fechado para expor partes do prato. Existem diferentes aberturas disponíveis para poder gerar diferentes estruturas no quadro. Dois tipos de estruturas devem ser considerados. Uma linha é exposta na placa movendo obturador fechado para um ponto final da linha, e então abre-se o obturador e o move para o outro ponto final da linha. Então o obturador é fechado. Uma estrutura do tipo ponto é gerada movendo-se (com abertura apropriada) para a posição deste ponto então abrindo o obturador apenas para fazer um flash curto e o fecha novamente. A modelagem exata do problema de controle da plotadora leva a um problema mais complicado que o TSP e também mais complicado que o problema do carteiro rural.

4.4 Roteamento de veículos

Suponha que uma cidade possui n caixas postais que devem ser esvaziadas todos os dias em um certo período de tempo, por exemplo 1 hora. O problema é encontrar o número mínimo de veículos para fazer isto e o menor tempo necessário para fazer as coletas usando este número de veículos. Esse problema é solucionável como um TSP se não houver restrições de tempo e capacidade e se o número de caminhões for fixo (por exemplo, m).

4.5 Problema de agendamento de impressão

Uma das maiores e principais aplicações do mTSP surge no agendamento de uma impressora para um periódico com várias edições. Aqui, existem cinco pares de cilindros entre os quais os rolos de papel e os dois lados de uma página são impressos simultaneamente. Existem três tipos de formulários, ou seja, formulários de 4, 6 e 8 páginas, usados para imprimir as edições. O problema de agendamento consiste em decidir qual formulário estará em qual execução e o tamanho de cada execução. No vocabulário mTSP, os custos de troca de placas são os custos entre cidades.

5 Implementação

A implementação foi feita através da construção de um projeto no NetBeans. Foram implementadas 9 classes que estão contidas dentro de um único pacote. É possível visualizar quais são as classes implementadas na figura abaixo.

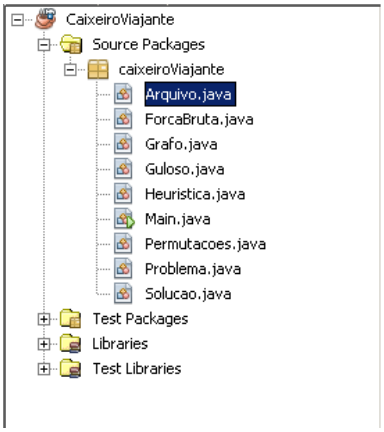

Figura 2: Classes no projeto CaixeiroViajante

5.1 Classe Main

A classe Main é a classe principal, responsável por realizar a chamada dos algoritmos que irão resolver o TSP. O método main contém objetos das classes Problema, Solucao, Grafo, Guloso, ForcaBruta e Heuristica. Há também variáveis para o controle de tempo de execução. A imagem abaixo mostra como é feita a chamada para a execução da solução de um dos algoritmos criados para solucionar o TSP.

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

solucao = guloso.getSolucao(cidadeInicial);

tempoFinal = System.currentTimeMillis();

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

System.out.println("Solução Heurística Gulosa(Vizinho mais próximo) - Cidade inicial (" + cidadeInicial + "):
        + solucao.getDistanciaTotal());

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

Figura 3: Chamada de solução do TSP (método Guloso)

Inicialmente é capturado o tempo atual do sistema na variável tempoInicial. É feita a chamada do método getSolucao (na imagem através de guloso.getSolucao(cidadeInicial)) 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 resoluçaõ do TSP que foram implementados.

A escolha do uso de instâncias providas via arquivo ou instâncias aleatórias também é feita no main através da habilitação dos trechos de códigos indicados no código fonte através de comentários.

5.2 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 um único método que se chama getMatrizFromFile. Este método atua 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. Há 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.

5.3 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. Seu conjunto de atributos é:

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

    //Atributos
    private final int numeroVertices; //aka número de cidades
    private int[][] matrizAdjacencia;
```

Figura 4: Conjunto de atributos da classe Grafo

5.4 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.

5.5 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:

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

    //Atributos
    private int distanciaTotal; //distância obtida do caminho solução
    private ArrayList<Integer> caminho; //caminho solução
```

Figura 5: Conjunto de atributos da classe Solucao

5.6 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:

```java
/**
 * 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 6: 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.

5.7 Classe ForcaBruta

A classe ForcaBruta implementa o algoritmo de força bruta para resolução do problema TSP. Seu funcionamento é através da geração de todas as possibilidades de caminhos existentes partindo de uma cidade inicial e na posterior verificação daquela possibilidade que possui o menor custo. O código fonte implementado pode ser visto na imagem abaixo:

```
private void calculaMenorCaminho(int cidadeInicial){
    //gera todas as permutações
    caminhos = permutacoes.getPermutacoes(geraPermutacaoInicial(cidadeInicial));
    //imprime(caminhos);
    //completa os caminhos de forma a ser possível sair da cidade inicial e voltar para a mesma
    ArrayList<int[]> caminhosCompletos = completarCaminhos(cidadeInicial, caminhos);

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

    //percorre o conjunto de caminhos comparando suas distâncias totais
    //o menor caminho será aquele cuja distância total tenha o menor valor dentre o conjunto verificado
    for (int i = 1; i < caminhosCompletos.size(); i++){
        if (getDistanciaCaminho(caminhosCompletos.get(i)) < getDistanciaCaminho(menorCaminho)){
            menorCaminho = caminhosCompletos.get(i);
        }
    }

    //passa os vértices obtidos do menor caminho para um ArrayList que será
    //passado para a solução
    for (int i = 0; i < menorCaminho.length; i++){
        menorCaminhoList.add(menorCaminho[i]);
    }

    //seta o melhor caminho obtido na solução
    solucao.setCaminho(menorCaminhoList);
    //solucao.mostrarCaminho();
    //seta a distância total obtida no melhor caminho
    solucao.setDistanciaTotal(getDistanciaCaminho(menorCaminho));
}
```

Figura 7: Método calculaMenorCaminho para o algoritmo de força bruta

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.

5.8 Classe Guloso

A classe Guloso implementa o algoritmo de heurística do vizinho mais próximo (heurística construtiva) para resolução do problema TSP. Partindo de uma cidade inicial, a ideia é adicionar ao caminho solução aquela cidade cuja distância seja a menor a partir da cidade atual. Repete-se este passo até que a quantidade de cidades do caminho solução seja equivalente a quantidade de cidades do problema inicial. Ao fim adiciona-se o custo do retorno para a cidade inicial e obtêm-se a solução final. O código fonte implementado pode ser visto nas imagens abaixo:

```
private void calculaMenorCaminho(int cidadeInicial) {
    ArrayList<Integer> caminhoTemp = new ArrayList<>(); //carregará o melhor caminho obtido
    caminhoTemp.add(cidadeInicial); //adiciona a cidade inicial ao início do percurso

    //utiliza o conceito de cidade atual (vértice atual) e
    //cidade mais próxima(vértice cuja aresta tem menor peso a partir do vértice atual)
    int cidadeAtual = cidadeInicial;
    int cidadeMaisProxima;
    int distanciaTotal = 0; //distancia total obtida no percurso
    int verticeDeMenorAresta; //vértice cujo peso é o menor a patir do vértice atual
    int nCidades = problema.getGrafo().numVertices(); //obtém a quantidade de cidades do problema

    //enquanto o caminho não houver a mesma quantidade de cidades do problema inicial
    while (caminhoTemp.size() < nCidades) {
        //obtém a lista de adjacência do vértice atual (cidade atual)
        ArrayList<Integer> listaAdj = problema.getGrafo().listaDeAdjacencia(cidadeAtual);

        //retira todas as cidades já percorridas da lista de adjacência
        for (Integer i : caminhoTemp) {
            listaAdj.remove(i);
        }

        //obtém a cidade mais próxima da cidade atual
        verticeDeMenorAresta = getVerticeComArestaDeMenorPeso(cidadeAtual, listaAdj);
        cidadeMaisProxima = verticeDeMenorAresta;

        //System.out.println("DistanciaEscolhida: " + problema.getDistancia(cidadeAtual, cidadeMaisProxima));
        //incrementa a distância total
        distanciaTotal = distanciaTotal + problema.getDistancia(cidadeAtual, cidadeMaisProxima);
```

Figura 8: Método calculaMenorCaminho para o algoritmo de heurística gulosa

```
        //adiciona a cidade mais próxima ao percurso
        caminhoTemp.add(cidadeMaisProxima);

        //seta a cidade atual como a cidade mais próxima obtida na solução
        cidadeAtual = cidadeMaisProxima;
    }

    //adiciona o custo da volta da última cidade para a cidade inicial
    distanciaTotal = distanciaTotal + problema.getDistancia(cidadeAtual, caminhoTemp.get(0));
    //adiciona a volta para a cidade inicial
    caminhoTemp.add(cidadeInicial);

    //seta o melhor percurso obtido
    solucao.setCaminho(caminhoTemp);
    //solucao.mostrarCaminho();
    //seta a distância total obtida para o melhor percurso
    solucao.setDistanciaTotal(distanciaTotal);
}
```

Figura 9: Método calculaMenorCaminho para o algoritmo de heurística gulosa

O método tem início adicionando ao caminho a cidade inicial. Em seguida, marca-se a cidade atual como a cidade inicial e inicia-se um laço while que somente encerrará quando o caminho solução contiver a mesma quantidade de cidades do problema. A cada iteração do laço while é obtida a lista de adjacência da cidade atual e desta são removidas as cidades já presentes no caminho solução para que

não haja o problema de retorno. No próximo passo se obtém a cidade cuja distância é a menor a partir da cidade atual, adiciona esta cidade ao caminho solução e diz que esta cidade agora é a cidade atual.

Uma vez terminado o laço while, adiciona-se a volta para a cidade inicial e incrementa o custo da volta para a cidade inicial a partir da última cidade colocada no caminho solução. 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.

5.9 Classe Heuristica

A classe Heuristica implementa o algoritmo de heurística da inserção mais barata (heurística construtiva) que consiste em construir uma rota passo a passo, partindo de rota inicial envolvendo 3 cidades (obtidas por um método qualquer) e adicionar a cada passo, a cidade k (ainda não visitada) entre a ligação (i, j) de cidades já visitadas, cujo custo de inserção seja o mais barato. O código fonte implementado pode ser visto nas imagens abaixo:

```
private void calculaMenorCaminho(int cidadeInicial) {
    //Consideramos que para esta heurística funcionar o mínimo de cidades
    //deve ser 3, pois para o funcionamento da mesma é admitido um caminho
    //inicial entre 3 cidades
    if (problema.getGrafo().numVertices() < 3) {
        System.out.println("Para este algoritmo o número mínimo de cidades é de 3.");
        return;
    }

    ArrayList<Integer> caminho = new ArrayList<>(); //representa o melhor caminho
    ArrayList<Integer> naoVisitados = new ArrayList<>(); //conjunto de cidades não visitadas
    ArrayList< Pair<Integer[], Integer>> tabelaTemp = new ArrayList<>(); //tabela de custos

    //gera o caminho inicial contendo 3 cidades, incluindo a cidade inicial
    geraCaminhoInicial(caminho, cidadeInicial);

    //atualiza o conjunto de cidades não visitadas
    atualizaNaoVisitados(caminho, naoVisitados);

    //enquanto a quantidade de cidades no caminho não for igual a quantidade de cidades do problema inicial
    while (caminho.size() < problema.getGrafo().numVertices()) {
        //percorre o conjunto de cidades visitadas, ou seja, cidades que já estão no caminho
        for (Integer visitado : caminho) {
            //percorre o conjunto de cidades não visitadas, ou seja, cidades que ainda não estão no caminho
            for (Integer naoVisitado : naoVisitados) {
                //percorre o conjunto de cidades visitadas
                for (Integer segVisitado : caminho) {
                    //valor que determina a garantia de que somente o caminho já existente será
                    //verificado para o calculo do custo na tabela
                    int valorDesejado = (caminho.indexOf(visitado) == caminho.size() - 1) ? caminho.get(caminho.indexOf(visitado)) :
                            caminho.get(caminho.indexOf(visitado) + 1);
```

Figura 10: Método calculaMenorCaminho para o algoritmo de heurística inserção mais barata

```
                    //se a cidade da iteração atual for diferente da cidade da primeira iteração
                    //e existe um caminho entre a cidade da iteração atual e cidade da primeira iteração
                    //no ArrayList que representa o melhor caminho
                    if (segVisitado != visitado && segVisitado == valorDesejado) {
                        //calcula o custo da inserção do vértice ainda não visitado
                        //Custo da inserção = dik + dkj - dij
                        //i -> cidade da primeira iteração
                        //k -> cidade ainda não visitada
                        //j -> cidade da iteração atual que é obrigatoriamente diferente da cidade da primeira iteração
                        int custo = problema.getDistancia(visitado, naoVisitado) + problema.getDistancia(naoVisitado, segVisitado)
                                - problema.getDistancia(visitado, segVisitado);
                        //armazena o custo atual e o percurso (i->k->j)
                        Integer[] caminhoTemp = new Integer[3];
                        caminhoTemp[0] = visitado;
                        caminhoTemp[1] = naoVisitado;
                        caminhoTemp[2] = segVisitado;
                        //adiciona na tabela de custos
                        tabelaTemp.add(new Pair(caminhoTemp, custo));
                    }
                }
            }
        }

        //obtém da tabela de custos o percurso cujo custo de inserção é o menor
        Pair<Integer[], Integer> melhorOpcao = getMelhorOpcao(tabelaTemp);
        //faz a inserção da cidade cujo custo de inserção é o menor
        insereMelhorOpcaoNoCaminho(melhorOpcao, caminho);
        //reinicializa a tabela de custos e o conjunto de cidades não visitadas
        tabelaTemp = new ArrayList<>();
        naoVisitados = new ArrayList<>();
```

Figura 11: Método calculaMenorCaminho para o algoritmo de heurística inserção mais barata

```
            //atualiza o conjunto de cidades não visitadas
            atualizaNaoVisitados(caminho, naoVisitados);
        }

        //fecha o caminho adicionando a volta para a cidade inicial
        caminho.add(cidadeInicial);
        //seta o caminho obtido como solução
        solucao.setCaminho(caminho);
        //converte ArrayList<Integer> em um vetor de inteiros
        Integer caminhoFinal[] = Arrays.asList(caminho.toArray()).toArray(new Integer[0]);
        //seta a distância total obtida do caminho solução
        solucao.setDistanciaTotal(getDistanciaCaminho(caminhoFinal));
    }
```

Figura 12: Método calculaMenorCaminho para o algoritmo de heurística inserção mais barata

É necessário para este método uma instância que contenha pelo menos 3 cidades para que possa ser executado. O método tem início gerando o caminho inicial a partir da inserção da cidade inicial no caminho solução e mais duas outras cidades escolhidas aleatoriamente. Inicia-se então um laço while que somente encerrará quando o caminho solução tiver a mesma quantidade de cidades do problema. Segue-se então uma sequência de 3 laços for responsáveis por criar a cada iteração do while uma tabela de custos que dirá qual cidade ainda não presente no caminho solução deve ser inserida. O custo é calculado da seguinte forma:

$$Custo = d(i,k) + d(k,j) - d(i,j)$$

Figura 13: Custo

Nesta expressão, i e j são cidades já presentes no caminho solução em que i precede j e k é uma cidade ainda não presente no caminho solução. A cidade k que apresentar o menor custo de inserção será inserida entre i e j no caminho solução. Atualiza-se então o conjunto de cidades ainda visitadas(cidades ainda não presentes no caminho solução) e continua-se a execução do laço while até que sua condição de parada seja satisfeita.

Terminado o laço adiciona-se a volta a cidade inicial a partir da última cidade presente no caminho solução e calcula-se a distância total obtida. 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.

6 Experimentos e análise de resultados

6.1 Configurações do Computador

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

6.2 Força Bruta

Figura 14: Gráfico Força Bruta

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

6.3 Heurísticas

Considerando o limite viável para execução de 5 minutos, a heurística gulosa (vizinho mais próximo) foi capaz de atingir o limite viável de 13000 cidades. 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 cidades. 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 15: Tabela Guloso

Na tabela acima é possível visualizar os tempos obtidos para cada instância e abaixo dos tempos o valor obtido para distância 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 cidades 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 16: 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 cidades em que a distância foi melhor.

7 Conclusão

Foi comprovado que o método de força bruta é inviável a medida que o número de instâncias cresce, tendo a quantidade de 11 cidades 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 um dos problemas mais importantes da Computação, que é o TSP. 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 teóricas e práticas.

Referências

[1] Wikipédia. Travelling salesman problem. Visitado em 11/19. Disponível em https://en.wikipedia.org/wiki/Travelling_salesman_problem.

[2] IntechOpen. Traveling Salesman Problem: an Overview of Applications, Formulations, and Solution Approaches. Visitado em 11/19. Disponível em https://www.intechopen.com/books/traveling-salesman-problem-theory-and-applications/traveling-salesman-problem-an-overview-of-applications-formulations-and-solution-approaches