Newton

Quem se prepara, não para.

Teoria de Grafos

RAFAEL DELFINO GONTIJO

SABRINA MARIA VAILANTE MEDEIROS

SAMUEL MARQUES ABREU PEREIRA

PROBLEMA DO CAIXEIRO VIAJANTE

Belo Horizonte - MG
2023

RAFAEL DELFINO GONTIJO

SABRINA MARIA VAILANTE MEDEIROS

SAMUEL MARQUES ABREU PEREIRA

PROBLEMA DO CAIXEIRO VIAJANTE

Versão aproximada para Problema do caixeiro Viajante apresentado ao Instituto Cultural Newton Paiva Ferreira, como exigência para a disciplina Teoria de Grafos.

Orientador(a): Prof(a).

Michelle Hanne

Belo Horizonte - MG
2023

Newton
Quem se prepara, não para.

SUMÁRIO

Introdução

o que é o problema do caixeiro Viajante?

O Problema do Caixeiro Viajante (PCV) é um problema clássico de otimização combinatória. Ele consiste em encontrar o menor caminho ou rota possível que um caixeiro viajante deve percorrer para visitar um conjunto de cidades uma única vez e retornar à cidade de origem.

Formalmente, dado um conjunto de cidades e as distâncias entre elas, o objetivo é encontrar a rota que minimize a distância total percorrida pelo caixeiro viajante, passando por todas as cidades exatamente uma vez e retornando à cidade de partida.

O PCV é considerado um problema NP-difícil, o que significa que não se conhece uma solução eficiente (polinomial) para resolvê-lo em todos os casos. À medida que o número de cidades aumenta, o número de possíveis rotas cresce exponencialmente, tornando a busca pela solução ótima computacionalmente inviável para instâncias grandes do problema.

Portanto, em muitos casos, são utilizadas heurísticas e algoritmos aproximados para encontrar soluções subótimas, mas razoavelmente boas, para o PCV. Essas soluções aproximadas não garantem a obtenção do valor ótimo, mas podem fornecer resultados aceitáveis em um tempo computacional razoável.

O PCV tem diversas aplicações práticas em áreas como logística, planejamento de rotas, transporte, telecomunicações e genética. A busca por soluções eficientes e aproximações melhores para o PCV continua sendo uma área ativa de pesquisa na ciência da computação.

O objetivo deste relatório é apresentar o algoritmo implementado para resolver o Problema do Caixeiro Viajante, utilizando uma heurística. Serão discutidos os detalhes do algoritmo, exemplos de sua aplicação, a escolha da linguagem de programação e a interação com o usuário.

Newton
Quem se prepara, não para.

Descrição do Algoritmo Implementado

O algoritmo implementado utiliza uma abordagem heurística conhecida como algoritmo guloso. Ele parte de uma cidade de origem selecionada pelo usuário e, em cada etapa, escolhe a cidade mais próxima não visitada para ser visitada em seguida. O processo continua até que todas as cidades tenham sido visitadas, e então o caixeiro retorna à cidade de origem, completando o ciclo.

Carregamento dos Dados

O algoritmo carrega as distâncias entre as cidades a partir de um arquivo CSV fornecido pelo usuário. As distâncias são armazenadas em uma matriz bidimensional, permitindo acesso rápido aos valores de distância entre as cidades.

distancias_minas_gerais.csv

Figura 1 - Arquivo .csv carregado no Código

Encontrar a Rota

O algoritmo começa selecionando a cidade de origem fornecida pelo usuário e adicionando-a à rota. Em seguida, ele itera sobre as cidades não visitadas e escolhe a cidade mais próxima para visitar em cada etapa. Essa escolha é

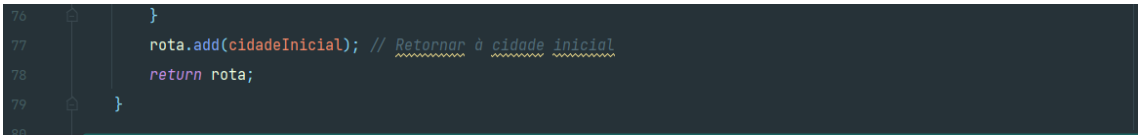
baseada nas distâncias armazenadas na matriz de distâncias. O processo continua até que todas as cidades tenham sido visitadas.

```
1 usage  new *
public List<Integer> encontrarRota(int cidadeInicial) {
    List<Integer> rota = new ArrayList<>();
    boolean[] visitadas = new boolean[NUM_CIDADES];
    rota.add(cidadeInicial);
    visitadas[cidadeInicial] = true;
    while (rota.size() < NUM_CIDADES) {
        int cidadeAtual = rota.get(rota.size() - 1);
        int proximaCidade = -1;
        int menorDistancia = Integer.MAX_VALUE;
        for (int cidade = 0; cidade < NUM_CIDADES; cidade++) {
            if (!visitadas[cidade] && distancias[cidadeAtual][cidade] < menorDistancia) {
                menorDistancia = distancias[cidadeAtual][cidade];
                proximaCidade = cidade;
            }
        }
        rota.add(proximaCidade);
        visitadas[proximaCidade] = true;
    }
    rota.add(cidadeInicial); // Retornar à cidade inicial
    return rota;
}
```

Figura 2 Encontrar Rota

Retornar à Cidade de Origem

Após visitar todas as cidades, o caixeiro retorna à cidade de origem, adicionando-a novamente à rota. Dessa forma, a rota forma um ciclo fechado, representando um percurso completo pelas cidades visitadas.



```
        }
        rota.add(cidadeInicial); // Retornar à cidade inicial
        return rota;
    }
```

Figura 3 Retornar a cidade de origem

2.4. Cálculo da Distância Total

Após obter a rota completa, o algoritmo calcula a distância total percorrida. Para isso, percorre a rota e soma as distâncias entre cada par de cidades adjacentes, considerando a matriz de distâncias.

```java
public int calcularDistanciaTotal(List<Integer> rota) {
    int distanciaTotal = 0;
    for (int i = 0; i < rota.size() - 1; i++) {
        int cidadeA = rota.get(i);
        int cidadeB = rota.get(i + 1);
        distanciaTotal += distancias[cidadeA][cidadeB];
    }
    return distanciaTotal;
}
```

Figura 4 Calculo da distância Total

2.4. Carregar Cidades

Com o intuito de auxiliar o usuário a escolher a cidade de origem, foi implementado o método carregaCidades Ao executar o programa, ele exibirá a lista de cidades com seus índices. Essa informação permitirá ao usuário fornece o índice da cidade de origem e das outras cidades a serem visitadas.

Aqui, a lista listaCidades é criada a partir dos elementos do indicesCidades e é ordenada com base nos valores dos índices. Em seguida, o loop percorre essa lista ordenada e exibe o índice e o nome de cada cidade.

Ao executar o programa, as cidades serão exibidas em ordem numérica crescente de acordo com os índices. Isso facilitará a seleção da cidade de origem e das cidades a serem visitadas, se necessário.

```java
1 usage  new *
public void carregarCidades() {
    List<Map.Entry<String, Integer>> listaCidades = new ArrayList<>(indicesCidades.entrySet());
    Collections.sort(listaCidades, Comparator.comparingInt(Map.Entry::getValue));

    for (Map.Entry<String, Integer> entrada : listaCidades) {
        String nomeCidade = entrada.getKey();
        int indiceCidade = entrada.getValue();
        System.out.println(indiceCidade + ", " + nomeCidade);
    }
}
```

Figura 5 Carregas a lista de Cidades

```
0, Belo Horizonte
1, Contagem
2, Uberlândia
3, Juiz de Fora
4, Ribeirão das Neves
5, Betim
6, Montes Claros
7, Uberaba
8, Governador Valadares
9, Santa Luzia
10, Ipatinga
11, Sete Lagoas
12, Divinópolis
13, Poços de Caldas
14, Ibirité
```

Figura 6 Lista de Cidades

Exemplos

Aqui estão alguns exemplos de execução do algoritmo implementado:

Exemplo 1:

```
 Qual o indice da cidade de origem? 0
Rota:
Cidade Belo Horizonte
Cidade Contagem
Cidade Betim
Cidade Ibirité
Cidade Ribeirão das Neves
Cidade Santa Luzia
Cidade Sete Lagoas
Cidade Divinópolis
Cidade Juiz de Fora
Cidade Ipatinga
Cidade Governador Valadares
Cidade Montes Claros
Cidade Uberlândia
Cidade Uberaba
Cidade Poços de Caldas
Cidade Belo Horizonte
Distância total percorrida: 3319 km
```

Figura 8 Rota a partir da cidade 0 - Belo Horizonte

1

Exemplo 2

```
 Qual o indice da cidade de origem? 8
Rota:
Cidade Governador Valadares
Cidade Ipatinga
Cidade Santa Luzia
Cidade Belo Horizonte
Cidade Contagem
Cidade Betim
Cidade Ibirité
Cidade Ribeirão das Neves
Cidade Sete Lagoas
Cidade Divinópolis
Cidade Juiz de Fora
Cidade Poços de Caldas
Cidade Uberaba
Cidade Uberlândia
Cidade Montes Claros
Cidade Governador Valadares
Distância total percorrida: 3067 km
```

Figura 7 Rota a partir da Cidade 8 - Governador Valadares

Interface com o Usuário

A interação com o usuário é realizada por meio da linha de comando. O programa primeiramente mostra a lista de cidades que estão pre cadastradas no no arquivo CSV carregado e posteriormente a isso, solicita a cidade de origem. As cidades visitadas são determinadas pelo algoritmo guloso. Ao final, o programa exibe a rota encontrada e a distância total percorrida.

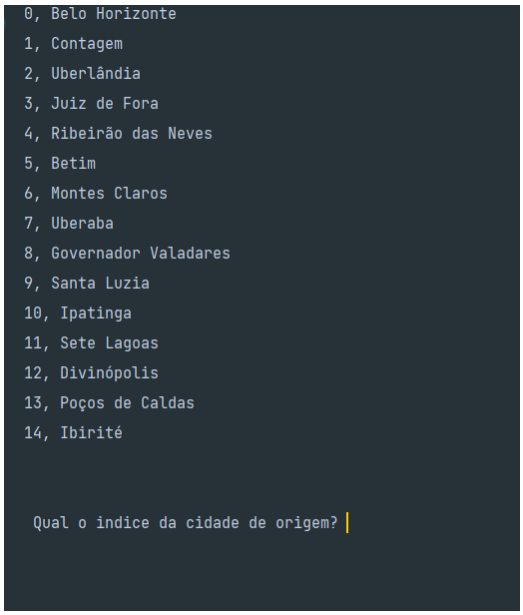


Figura 9 - 10 Interação com o usuário - Mostra a Lista de Cidades e pergunta a Cidade Origem

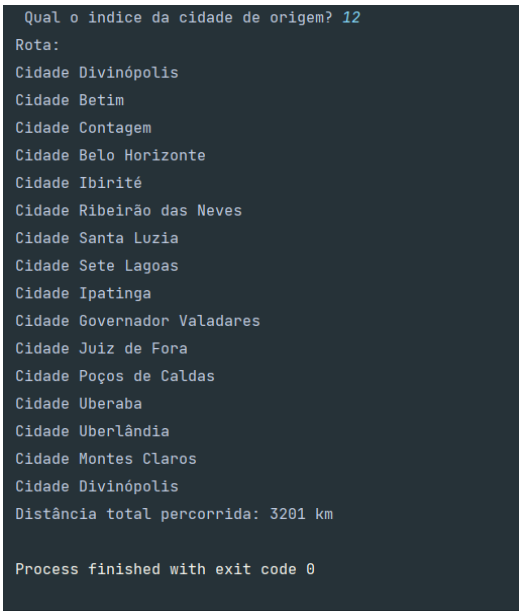


Figura Mostra a Rota a partir da cidade escolhida

Código completo

A implementação foi realizada em Java devido às suas vantagens, como ser uma linguagem de propósito geral, amplamente utilizada e com uma rica biblioteca padrão. Além disso, Java é conhecida por sua portabilidade, o que permite que o algoritmo seja executado em diferentes plataformas.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.*;

public class ProblemaCaixeiroViajante {
    private static final String CAMINHO_ARQUIVO =
"C:\\Users\\sabri\\OneDrive\\Documentos\\Projetos\\ProblemaDoCaixeiroViajante\\src\\distancias_minas_gerais.csv";
    private static final int NUM_CIDADES = 15;

    private Map<String, Integer> indicesCidades;
    private int[][] distancias;

    public ProblemaCaixeiroViajante() {
        indicesCidades = new HashMap<>();
        distancias = new int[NUM_CIDADES][NUM_CIDADES];
    }
    public void carregarDistancias(String caminhoArquivo) {
        try {
            BufferedReader leitor = new BufferedReader(new FileReader(caminhoArquivo));
            String linha = leitor.readLine(); // Ignorar a primeira linha (cabeçalho)

            int indiceCidade = 0;
            while ((linha = leitor.readLine()) != null) {
                String[] partes = linha.split(",");
                String nomeCidade = partes[0].replaceAll("\"", ""); // Remover as aspas duplas
                indicesCidades.put(nomeCidade, indiceCidade);

                String[] distanciasStr = Arrays.copyOfRange(partes, 1, partes.length);
                for (int i = 0; i < distanciasStr.length; i++) {
                    int distancia = Integer.parseInt(distanciasStr[i]);
                    distancias[indiceCidade][i] = distancia;
                    distancias[i][indiceCidade] = distancia; // Considere que a distância é bidirecional
                }
                indiceCidade++;
            }

            leitor.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public String getNomeCidade(int indiceCidade) {
        for (Map.Entry<String, Integer> entrada : indicesCidades.entrySet()) {
```

```
        if (entrada.getValue() == indiceCidade) {
            return entrada.getKey();
        }
    }
    return null;
}
public List<Integer> encontrarRota(int cidadeInicial) {
    List<Integer> rota = new ArrayList<>();
    boolean[] visitadas = new boolean[NUM_CIDADES];
    rota.add(cidadeInicial);
    visitadas[cidadeInicial] = true;
    while (rota.size() < NUM_CIDADES) {
        int cidadeAtual = rota.get(rota.size() - 1);
        int proximaCidade = -1;
        int menorDistancia = Integer.MAX_VALUE;
        for (int cidade = 0; cidade < NUM_CIDADES; cidade++) {
            if (!visitadas[cidade] && distancias[cidadeAtual][cidade] < menorDistancia) {
                menorDistancia = distancias[cidadeAtual][cidade];
                proximaCidade = cidade;
            }
        }
        rota.add(proximaCidade);
        visitadas[proximaCidade] = true;
    }
    rota.add(cidadeInicial); // Retornar à cidade inicial
    return rota;
}
public int calcularDistanciaTotal(List<Integer> rota) {
    int distanciaTotal = 0;
    for (int i = 0; i < rota.size() - 1; i++) {
        int cidadeA = rota.get(i);
        int cidadeB = rota.get(i + 1);
        distanciaTotal += distancias[cidadeA][cidadeB];
    }
    return distanciaTotal;
}
public void carregarCidades() {
    List<Map.Entry<String, Integer>> listaCidades = new ArrayList<>(indicesCidades.entrySet());
    Collections.sort(listaCidades, Comparator.comparingInt(Map.Entry::getValue));

    for (Map.Entry<String, Integer> entrada : listaCidades) {
        String nomeCidade = entrada.getKey();
        int indiceCidade = entrada.getValue();
        System.out.println(indiceCidade + ", " + nomeCidade);
    }
}
public static void main(String[] args) {
    ProblemaCaixeiroViajante pcv = new ProblemaCaixeiroViajante();
    pcv.carregarDistancias(CAMINHO_ARQUIVO);
    pcv.carregarCidades();

    Scanner scanner = new Scanner(System.in);

    System.out.print("\n \n Qual o indice da cidade de origem? ");
    int cidadeOrigem = scanner.nextInt();

    List<Integer> rota = pcv.encontrarRota(cidadeOrigem);
    int distanciaTotal = pcv.calcularDistanciaTotal(rota);
```

```
        System.out.println("Rota:");
        for (int indiceCidade : rota) {
            String nomeCidade = pcv.getNomeCidade(indiceCidade);
            System.out.println("Cidade " + nomeCidade);
        }
        System.out.println("Distância total percorrida: " + distanciaTotal + " km");

        scanner.close();
    }
}
```

Conclusão

O algoritmo implementado para resolver o Problema do Caixeiro Viajante utilizando uma heurística demonstrou ser eficiente na obtenção de soluções aproximadas. Ele utiliza o algoritmo guloso para selecionar as cidades mais próximas a cada etapa, formando uma rota que percorre todas as cidades e retorna à cidade de origem. A escolha da linguagem Java e a interface com o usuário por linha de comando garantem uma execução eficiente e uma interação simples com o programa.

A heurística do vizinho mais próximo oferece uma abordagem eficiente para resolver o Problema do Caixeiro Viajante em um tempo razoável, especialmente para instâncias menores. Embora essa solução não garanta a rota ótima, ela fornece uma solução aproximada que geralmente está muito próxima da ótima.

No entanto, é importante destacar que o Problema do Caixeiro Viajante é um problema NP-difícil, o que significa que encontrar a solução ótima para instâncias maiores é impraticável em um tempo viável. Nesses casos, a abordagem heurística é uma alternativa viável para obter uma solução satisfatória em um tempo razoável.

No aspecto da interface com o usuário, embora o programa atualmente solicite apenas a cidade de origem, ele pode ser facilmente expandido para permitir a seleção de um conjunto personalizado de cidades a serem visitadas. Essa flexibilidade proporciona ao usuário a capacidade de explorar diferentes rotas e personalizar a experiência de acordo com suas necessidades.

Em resumo, o algoritmo implementado para o Problema do Caixeiro Viajante utilizando uma heurística de vizinho mais próximo demonstrou ser uma solução eficiente e viável. Ele oferece uma solução aproximada que pode ser obtida em um tempo razoável e apresenta uma interface simples com o usuário. Embora não garanta a solução ótima, o algoritmo fornece resultados satisfatórios para a maioria das instâncias do problema.