



PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Projeto e Análise de Algoritmos e Inteligência Artificial

Problema do Caixeiro Viajante

Caio Duarte Santos, Iander Marques, Luís Gustavo

Belo Horizonte
2022

Introdução:

O problema do caixeiro viajante, consiste em determinar a menor rota para percorrer uma série de cidades, sendo necessário visitar somente uma única vez cada uma das cidades. Este é um conhecido problema da computação pertencente à classe NP-hard, sendo classificado como um problema de otimização.

Para solucionar o problema apresentado acima, vamos propor algoritmos que resolvem o problema em questão dos mais variadas formas e paradigmas, entre eles estão a programação dinâmica, força bruta e guloso.

O principal objetivo deste trabalho, é apresentar a diferença entre as implementações propostas, contando com testes a fim de mostrar a diferença entre os custos, tempos e caminhos encontrados.

Implementação

- Programação Dinâmica

O algoritmo construído no paradigma de programação dinâmica para resolução do problema caixeiro viajante, foi construído utilizando a linguagem python.

Inicialmente, o algoritmo lê a entrada presente no arquivo de texto. Os dados de entrada são compostos por:

```
8 n
9 X1 Y1
10 X2 Y2
11 X3 Y3
12 X4 Y4
13
```

- **n**: número de cidades;
- **Xn**: coordenada da cidade no eixo X;
- **Yn**: coordenada da cidade no eixo Y;

Exemplo de dados de entrada:

```
1 4
2 1 2
3 3 4
4 5 6
5 7 11
6
```

Após a leitura das entradas, realizamos o cálculo das distâncias entre as cidades. Para isso, construímos uma função que percorre o array de coordenadas das cidades que retorna um array de distâncias de cada cidade para as outras.

```
def calculateCityDistances(cityArray):
    distancesArray = []

    for city in cityArray:
        distCity = []
        for targetCity in cityArray:
            distCityAndTarget = sqrt((city[0]-targetCity[0])**2) +
            ((city[1]-targetCity[1])**2)
            distCity.append(distCityAndTarget)

        distancesArray.append(distCity)

    return distancesArray
```

Para cada item percorrido no loop, é calculado a distância entre os pontos x e y de cada cidade.

- Array de distâncias retornado:

```
[[0.0, 6.0, 20.0, 87.0], [6.0, 0.0, 6.0, 53.0], [20.0, 6.0, 0.0, 27.0], [87.0, 53.0, 27.0, 0.0]]
```

Após o cálculo das distâncias, inicia-se a busca pelo caminho ótimo entre as cidades.

```
def DP_TSP(distances_array):
    n = len(distances_array)
    all_points_set = set(range(n))

    # memo keys: tuple(sorted_points_in_path, last_point_in_path)
    # memo values: tuple(cost_thus_far, next_to_last_point_in_path)
    memo = {(tuple([i]), i): tuple([0, None]) for i in range(n)}
    queue = [(tuple([i]), i) for i in range(n)]

    while queue:
        prev_visited, prev_last_point = queue.pop(0)
        prev_dist, _ = memo[(prev_visited, prev_last_point)]

        to_visit = all_points_set.difference(set(prev_visited))
        for new_last_point in to_visit:
            new_visited = tuple(sorted(list(prev_visited) + [new_last_point]))
            new_dist = prev_dist + distances_array[prev_last_point][new_last_point]

            if (new_visited, new_last_point) not in memo:
                memo[(new_visited, new_last_point)] = (new_dist, prev_last_point)
                queue += [(new_visited, new_last_point)]
            else:
                if new_dist < memo[(new_visited, new_last_point)][0]:
                    memo[(new_visited, new_last_point)] = (new_dist, prev_last_point)

    optimal_path, optimal_cost = retrace_optimal_path(memo, n)

    return optimal_path, optimal_cost
```

```
def retrace_optimal_path(memo: dict, n: int) -> [[int], float]:
    points_to_retrace = tuple(range(n))

    full_path_memo = dict((k, v) for k, v in memo.items() if k[0] == points_to_retrace)
    path_key = min(full_path_memo.keys(), key=lambda x: full_path_memo[x][0])

    last_point = path_key[1]
    optimal_cost, next_to_last_point = memo[path_key]

    optimal_path = [last_point]
    points_to_retrace = tuple(sorted(set(points_to_retrace).difference({last_point})))

    while next_to_last_point is not None:
        last_point = next_to_last_point
        path_key = (points_to_retrace, last_point)
        _, next_to_last_point = memo[path_key]

        optimal_path = [last_point] + optimal_path
        points_to_retrace = tuple(sorted(set(points_to_retrace).difference({last_point})))

    return optimal_path, optimal_cost
```

A busca pelo melhor caminho começa por qualquer primeira posição do array de distâncias, em seguida, expandimos cada estado adicionando todos os nós possíveis para fazer um caminho de tamanho 2 e assim por diante. Cada vez que visitamos uma solução parcial que já foi visitada anteriormente, mantemos apenas a melhor solução obtida.

- Caixeiro viajante, força bruta:

Construção do algoritmo usando a linguagem C# situação busca exaustiva, também conhecida como gerar e testar ou força bruta, usando a técnica de solução de problemas trivial, mas muito geral, que envolve a enumeração de todos os possíveis candidatos à solução e o exame de cada candidato para ver se ele satisfaz o enunciado do problema.

inicialmente o programa busca pelo o número de cidades que devem ser percorridas pelo algoritmo :

```
Console.Write("\n\n\t\tDigite o numero de cidades: ");
numCidades = int.Parse(Console.ReadLine());
caixeiro.montaGrafo(out grafo, numCidades);
```

armazenando os dados na variável **numCidades**, exemplo dados de entrada;

```
Digite o numero de cidades: 5
```

Gera os pesos dos arcos do grafo randomicamente e preenche a matriz grafo->M, que é indexada pelos nomes dos vértices (cidades);

```
void montaGrafo(out Grafo grafo, int numCidades)
{
    //int i, j;      // indexadores da matriz
    int custo;      // pesos dos arcos {i, j}

    Random randomizer = new Random();
    grafo = new Grafo();
    grafo.M = new int[numCidades, numCidades];

    for (int i = 0; i < numCidades; i++)
    {
        for (int j = 0; j < numCidades; j++)
        {
            custo = randomizer.Next(MaxCusto) + 1;
            if (i < j)
                grafo.M[i, j] = custo;
            else
            {
                if (i == j)
                    grafo.M[i, j] = 0;
                else
                    grafo.M[i, j] = grafo.M[j, i];
            }
        }
    }
}
```

Logo em seguida gera os possíveis caminhos entre a cidade zero e todas as outras envolvidas na rota da viagem do caixeiro e escolhe a melhor rota entre todas.

Chamando a função **permuta** que gera os possíveis caminhos entre a cidade zero e as outras (N-1) envolvidas na busca, armazenando-os no vetor permutação, um por vez, e a cada permutação gerada, chama a função melhor Caminho que escolhe o caminho (a permuta) de menor custo.

Faço também a instância da **CaixeiroOtimo caixeiro = new CaixeiroOtimo;** para verificar se a permutação passada como parâmetro tem custo melhor que o custo já obtido. Caso positivo, então monta a rota correspondente à permutação como sendo a melhor rota e armazena no vetor melhor Rota, retornando também o custo total da melhor rota.

```
void permuta(int[] permutacao, Grafo grafo, Rota[] melhorRota, ref int melhorCusto, int controle, int k)
{
    int i;
    permutacao[k] = ++controle;
    if (controle == (melhorRota.Length - 1)) /* se gerou um caminho então verifica se ele é melhor */
        melhorCaminho(grafo, melhorRota, ref melhorCusto, permutacao);
    else
        for (i = 1; i < melhorRota.Length; i++)
            if (permutacao[i] == 0)
                permuta(permutacao, grafo, melhorRota, ref melhorCusto, controle, i);
    controle--;
    permutacao[k] = 0;
}
```

```

void imprimeMelhorCaminho(int custo, Rota[] melhorRota)
{
    int i; /* indexa o vetor que contem a rota */
    Console.WriteLine("\n\nCUSTO MINIMO PARA A VIAGEM DO CAIXEIRO: " + custo);
    Console.WriteLine("\n\nMELHOR CAMINHO PARA A VIAGEM DO CAIXEIRO:");
    Console.WriteLine("\n\n          DE          PARA          CUSTO ");
    for (i = 0; i < melhorRota.Length; i++)
    {
        Console.Write("          " + melhorRota[i].cidade1 + "          " + melhorRota[i].cidade2 + "          " + melhorRota[i].custo + "\n");
    }
    Console.WriteLine("\n");
}

```

Imprimindo o melhor caminho para a viagem do caixeiro, bem como o custo total da viagem;

por fim mostra o tempo de execução do algoritmo

```

void imprimeTempo(Stopwatch tempo)
{
    Console.WriteLine("TEMPO DE EXECUÇÃO: ");
    Console.WriteLine(tempo.Elapsed.Hours+" horas "+tempo.Elapsed.Minutes+" minutos "+tempo.Elapsed.Seconds+" segundos "+tempo.Elapsed.Milliseconds+" milisegundos");
} /* fim Imprime tempo */

```

execução da main saída do programa:

```

Cidades e custos:
  0 1 2 3 4
0  0 38 30 36 24
1  38 0 20 14 9
2  30 20 0 41 42
3  36 14 41 0 20
4  24 9 42 20 0
CUSTO MINIMO PARA A VIAGEM DO CAIXEIRO: 108

```

MELHOR CAMINHO PARA A VIAGEM DO CAIXEIRO:DE	PARA	CUSTO
0	2	30
2	1	20
1	3	14
3	4	20
4	0	24

Acredito que o pior caso é o algoritmo tem que comparar todas as n cidades antes de se mover.

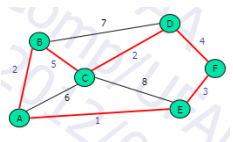


imagem ilustrativa,custo = 17

compilador com terminal linux para teste,lembrando que o arquivo está todo comentado para demais dúvidas a vir sobre o programa.

<https://www.programiz.com/csharp-programming/online-compiler/> (Csharp).

O nome do arquivo que será enviado junto há documentação(forçaBruta.cs.)

- Algoritmo Guloso

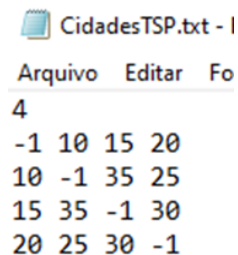
Para implementar o problema do caixeiro viajante com uma abordagem gulosa foi utilizado a linguagem Java. Em que foi criada uma lista com os índices das cidades da matriz de entrada, com as distâncias entre elas. É feita busca na matriz de adjacência dada `tsp[][]` para toda a cidade e se o custo de alcançar qualquer cidade da cidade atual for menor que o custo atual, atualize o custo. Então, gera o ciclo de caminho mínimo e retorna o custo mínimo.

O programa lê de um arquivo as distâncias das cidades e preenche uma matriz do tamanho informado no arquivo com esses valores. Utilizando a função `readFile()`, criada para fazer essa conversão do arquivo para a matriz dentro do programa.

```
String caminho = "src/CidadesTSP.txt";

int[][] tspMatrix = readFile(caminho);
```

Como podemos ver nesse exemplo de arquivo de entrada abaixo, que representa 4 cidades e o custo de seus caminhos.



```
CidadesTSP.txt - I
Arquivo  Editar  Fo
4
-1 10 15 20
10 -1 35 25
15 35 -1 30
20 25 30 -1
```

A função que vai percorrer os caminhos e encontrar o melhor é a `findMinRoute()`, que foi implementada da seguinte forma:

```
static void findMinRoute(int[][] tsp) throws IOException {
    int sum = 0;
    int counter = 0;
    int j = 0, i = 0;
    int min = Integer.MAX_VALUE;
    List<Integer> visitedRouteList
        = new ArrayList<>();

    visitedRouteList.add(0);
    int[] route = new int[tsp.length];
```

Primeiramente, ele já adiciona a primeira cidade na `visitedRouteList`, que vai ser responsável por manter quais nós foram visitados naquele caminho que está sendo analisado. Nisso é inicializada também a variável `route` que é um vetor que vai armazenar o caminho.

```

while (i < tsp.length && j < tsp[i].length) {

    if (counter >= tsp[i].length - 1) {
        break;
    }
    //Se o caminho nao foi visitado e o custo for menor, atualiza o custo
    if (j != i && !(visitedRouteList.contains(j))) {
        if (tsp[i][j] < min) {
            min = tsp[i][j];
            route[counter] = j + 1;
        }
    }
    j++;
    // verifica todos os caminhos a partir do indice i
    if (j == tsp[i].length) {
        sum += min;
        min = Integer.MAX_VALUE;
        visitedRouteList.add(route[counter] - 1);
        j = 0;
        i = route[counter] - 1;
        counter++;
    }
}

// Atualiza a cidade final no array
i = route[counter - 1] - 1;

for (j = 0; j < tsp.length; j++) {

    if ((i != j) && tsp[i][j] < min) {
        min = tsp[i][j];
        route[counter] = j + 1;
    }
}

sum += min;
System.out.print("Custo minimo: ");
System.out.println(sum);

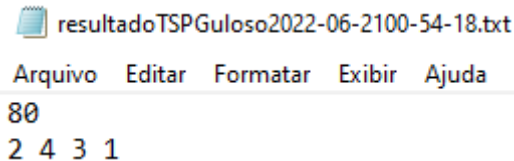
```

Aqui é percorrido os caminhos a partir de determinada cidade e são comparados os caminhos possíveis a partir dela, em que se o custo de ir para qualquer cidade através da cidade atual for menor que o custo atual, é atualizado o custo e o caminho. A complexidade do algoritmo é $O(N^2 \log_2 N)$.

Depois disso é escrito no arquivo de saída o custo mínimo encontrado e o caminho percorrido.

```
sum += min;  
System.out.print("Custo mínimo: ");  
System.out.println(sum);  
  
writeFile(sum,route);
```

Exemplo de arquivo de saída:



resultadoTSPGuloso2022-06-2100-54-18.txt

Arquivo Editar Formatar Exibir Ajuda

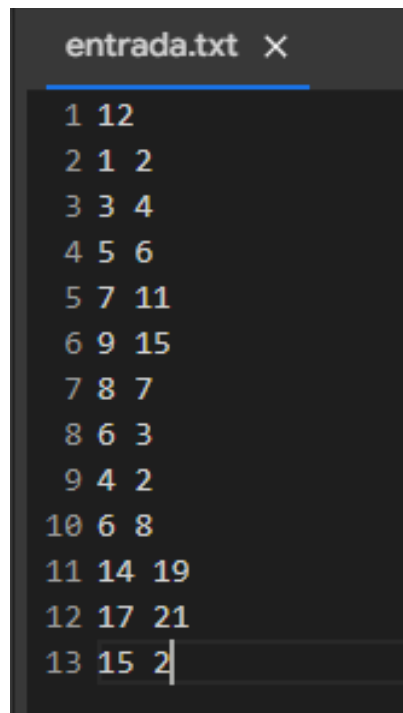
80

2 4 3 1

Testes

- Programação Dinâmica

Para realizar os testes na solução de programação dinâmica, vamos estabelecer a seguinte entrada para a aplicação:



entrada.txt X

1 12

2 1 2

3 3 4

4 5 6

5 7 11

6 9 15

7 8 7

8 6 3

9 4 2

10 6 8

11 14 19

12 17 21

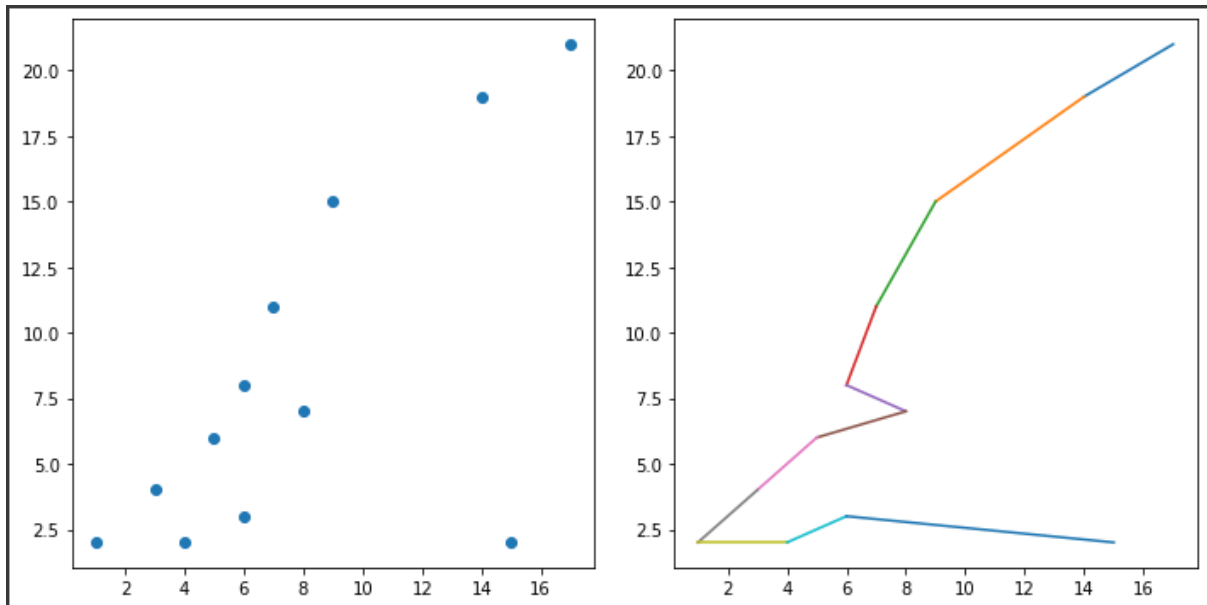
13 15 2

A entrada acima é composta por 12 cidades, contendo os pares de coordenadas X,Y. Cada linha abaixo do valor "N" (12), representa as coordenadas de cada cidade em sequência.

Para este conjunto de entradas, obtivemos os resultados abaixo com o seguinte caminhamento:

```
Encontrou o caminho ideal em 0.262 segundos.  
Custo ótimo: 91.0, caminho ótimo: [10, 9, 4, 3, 8, 5, 2, 1, 0, 7, 6, 11]
```

Caminhamento plotado em gráfico:



- Força Bruta

Para realizar os testes no caixeiro viajante solução força bruta, vamos estabelecer a seguinte entrada para a aplicação:

Número de cidades testadas: 12.

onde o tempo de execução foi de:

0 horas 0 minutos 17 segundos 282 milisegundos

Pode ser testado com um número como o 17 onde o algoritmo irá demorar muito tempo para ser executado.

Para este conjunto de entradas, obtivemos os resultados abaixo com o seguinte caminhamento:

Digite o numero de cidades: 12

2

Cidades e custos:

```
  0 1 2 3 4 5 6 7 8 9 10 11
0  0 31 32 1 32 18 34 3 40 26 40 6
1  31 0 3 35 21 48 1 12 42 49 19 12
2  32 3 0 44 2 20 45 22 16 33 9 44
3  1 35 44 0 32 32 17 23 40 23 28 15
4  32 21 2 32 0 21 12 7 13 21 49 5
5  18 48 20 32 21 0 40 47 22 10 36 45
6  34 1 45 17 12 40 0 31 50 20 1 18
7  3 12 22 23 7 47 31 0 28 42 21 42
8  40 42 16 40 13 22 50 28 0 44 14 34
9  26 49 33 23 21 10 20 42 44 0 40 3
10 40 19 9 28 49 36 1 21 14 40 0 37
11 6 12 44 15 5 45 18 42 34 3 37 0
```

CUSTO MINIMO PARA A VIAGEM DO CAIXEIRO: 82

MELHOR CAMINHO PARA A VIAGEM DO CAIXEIRO:

CUSTO MINIMO PARA A VIAGEM DO CAIXEIRO: 82

MELHOR CAMINHO PARA A VIAGEM DO CAIXEIRO:

	DE	PARA	CUSTO
	0	7	3
	7	4	7
	4	2	2
	2	1	3
	1	6	1
	6	10	1
	10	8	14
	8	5	22
	5	9	10
	9	11	3
	11	3	15
	3	0	1

TEMPO DE EXECU??O:

0 horas 0 minutos 17 segundos 282 milisegundos

Algoritmo guloso

Para testarmos o desempenho do algoritmo, iremos utilizar como entrada uma matriz de caminhos entre 12 cidades:

1	12
2	-1 43 5 39 36 20 15 22 8 22 42 22
3	36 -1 27 23 30 14 38 24 25 43 5 27
4	45 16 -1 13 25 23 45 23 42 8 18 27
5	44 21 11 -1 44 11 13 20 43 30 9 28
6	38 2 11 2 -1 11 26 9 21 8 1 43
7	30 40 1 45 42 -1 23 26 1 12 36 41
8	40 2 9 15 23 36 -1 14 37 11 22 23
9	15 3 14 16 32 33 18 -1 39 9 35 15
10	25 20 38 36 44 3 35 27 -1 33 22 42
11	31 8 19 11 23 28 4 13 44 -1 5 11
12	4 3 31 24 13 17 45 39 17 43 -1 15
13	27 34 24 31 27 16 18 25 44 29 34 -1

Foi adicionado ao programa as seguintes linhas para monitorar o tempo gasto para a execução do método findMinRoute():

```
Instant startTime = Instant.now();

findMinRoute(tspMatrix);

Instant endTime = Instant.now();
Duration totalTime = Duration.between(startTime, endTime);

System.out.println("\n"+totalTime);
```

Resultados obtidos:

- Arquivo de saída:

TSPGuloso.java	resultadoTSPGuloso2022-06-2101-19-59.txt
1	109
2	3 10 7 2 11 5 4 6 9 8 12 6

- Tempo Gasto:

PT0.034907S

O programa gastou aproximadamente 0,035s para encontrar o melhor caminho entre as 12 cidades apresentadas.

Anexos

Segue abaixo, os algoritmos construídos para resolução do problema do caixeiro viajante.

- Programação Dinâmica

```
import time
import numpy as np
from math import sqrt
np.random.seed(42)

import matplotlib.pyplot as plt
%matplotlib inline
```

```
def DP_TSP(distances_array):
    n = len(distances_array)
    all_points_set = set(range(n))

    memo = {(tuple([i]), i): tuple([0, None]) for i in range(n)}
    queue = [(tuple([i]), i) for i in range(n)]

    while queue:
        prev_visited, prev_last_point = queue.pop(0)
        prev_dist, _ = memo[(prev_visited, prev_last_point)]

        to_visit = all_points_set.difference(set(prev_visited))
        for new_last_point in to_visit:
            new_visited = tuple(sorted(list(prev_visited) + [new_last_point]))
            new_dist = prev_dist + distances_array[prev_last_point][new_last_point]

            if (new_visited, new_last_point) not in memo:
                memo[(new_visited, new_last_point)] = (new_dist, prev_last_point)
                queue += [(new_visited, new_last_point)]
            else:
                if new_dist < memo[(new_visited, new_last_point)][0]:
                    memo[(new_visited, new_last_point)] = (new_dist, prev_last_point)

    optimal_path, optimal_cost = retrace_optimal_path(memo, n)

    return optimal_path, optimal_cost
```

```

def retrace_optimal_path(memo: dict, n: int) -> [[int], float]:
    points_to_retrace = tuple(range(n))

    full_path_memo = dict((k, v) for k, v in memo.items() if k[0] == points_to_retrace)
    path_key = min(full_path_memo.keys(), key=lambda x: full_path_memo[x][0])

    last_point = path_key[1]
    optimal_cost, next_to_last_point = memo[path_key]

    optimal_path = [last_point]
    points_to_retrace = tuple(sorted(set(points_to_retrace).difference({last_point})))

    while next_to_last_point is not None:
        last_point = next_to_last_point
        path_key = (points_to_retrace, last_point)
        _, next_to_last_point = memo[path_key]

        optimal_path = [last_point] + optimal_path
        points_to_retrace = tuple(sorted(set(points_to_retrace).difference({last_point})))

    return optimal_path, optimal_cost

```

```

def calculateCityDistances(cityArray):
    distancesArray = []

    for city in cityArray:
        distCity = []
        for targetCity in cityArray:
            distCityAndTarget = sqrt((city[0]-targetCity[0])**2) + ((city[1]-targetCity[1])**2)
            distCity.append(distCityAndTarget)

        distancesArray.append(distCity)

    return distancesArray

```

```
def readInputFile(fileName):
    with open(fileName) as f:
        array = []
        n = int(next(f))

        z = 0
        for line in f:
            a = []
            i = 0
            for x in line.split() :
                a.append(int(x))
                i = i+1
            array.append(a)
            z = z+1

        return np.array(array)
```

```
def plot_route(X, optimal_path):
    for p1, p2 in zip(optimal_path[:-1], optimal_path[1:]):
        plt.plot([X[p1, 0], X[p2, 0]], [X[p1, 1], X[p2, 1]]);
```

```
X = readInputFile('entrada.txt')
distances_array = calculateCityDistances(X)

t = time.time()
optimal_path, optimal_cost = DP_TSP(distances_array)
runtime = round(time.time() - t, 3)

print(f"Encontrou o caminho ideal em {runtime} segundos.")
print(f"Custo ótimo: {round(optimal_cost, 3)}, caminho ótimo: {optimal_path}")
```

- Força Bruta

```
C# forcaBruta.cs
1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Text;
5
6  namespace Caixeiro
7  {
8
9      /* Um campo do vetor que contera a melhor rota */
10     class Rota
11     {
12         public int cidade1, cidade2, custo;
13     }
14
15     //-----
16
17     /* matriz que contera' o grafo completo */
18     class Grafo
19     {
20         public int[,] M; /** o grafo esperado*/
21     }
22
23     //-----
24
25     class CaixeiroOtimo
26     {
27         const int MaxCusto = 50;    // custo ou distância máxima entre duas cidades
28         //-----
29
30         /*
31         * Verifica-se a permutação passada como parâmetro tem custo melhor que o custo
32         * já obtido. Caso positivo, então monta a rota correspondente à permutação como
33         * sendo a melhor rota (e armazena no vetor melhorRota, retornando também o custo
34         * total da melhor rota
35         */
36         void melhorCaminho(Grafo grafo, Rota[] melhorRota, ref int melhorCusto, int[] permutacao)
37         {
38             int j, k;                /* contadores: auxiliam a montagem das rotas */
39             int cid1, cid2;          /* cidades da melhor rota */
40             int custo;               /* custo total da melhor rota */
41             int[] proxDaRota;         /* vetor que armazena a sequência de cidades que estão
42                                     em uma rota, tal que um índice indica uma cidade e
43                                     o conteúdo deste índice, a próxima cidade da rota */
44
45             proxDaRota = new int[melhorRota.Length];
46             /* monta uma rota com a permutacao */
47             cid1 = 0;                /* a primeira cidade é a cidade 0 */
48             cid2 = permutacao[1];
49             custo = grafo.M[cid1, cid2];
50             proxDaRota[cid1] = cid2;
51
52             for (j = 2; j < melhorRota.Length; j++)
53             {
54                 cid1 = cid2;
55                 cid2 = permutacao[j];
56                 custo += grafo.M[cid1, cid2]; /* calcula o custo parcial da rota */
57                 proxDaRota[cid1] = cid2;      /* armazena a rota fornecida pela permutacao */
58             }
59
60             proxDaRota[cid2] = 0;           /* completa o ciclo da viagem */
61         }
62     }
63 }
```



```

C# forçaBrutal.cs
60     proxDaRota[cid2] = 0;          /* completa o ciclo da viagem */
61     custo += grafo.M[cid2, 0];    /* custo total desta rota */
62
63     if (custo < melhorCusto)      /* procura pelo melhor (menor) custo */
64     {
65         melhorCusto = custo;
66         cid2 = 0;
67         for (k = 0; k < melhorRota.Length; k++) /* guarda a melhor rota */
68         {
69             cid1 = cid2;
70             cid2 = proxDaRota[cid1];
71             melhorRota[k].cidade1 = cid1;
72             melhorRota[k].cidade2 = cid2;
73             melhorRota[k].custo = grafo.M[cid1, cid2];
74         }
75     }
76 } /* fim melhorCaminho */
77
78 //-----
79 /* Gera os possíveis caminhos entre a cidade zero e as outras (N-1) envolvidas
80    na busca, armazenando-os no vetor permutacao, um por vez, e a cada permutacao
81    gerada, chama a funcao melhorCaminho que escolhe o caminho (a permutacao) de
82    menor custo.
83    * CÓDIGO ADAPTADO DE "Algorithms in C" (Robert Sedgewick), página 624.
84    */
85 void permuta(int[] permutacao, Grafo grafo, Rota[] melhorRota, ref int melhorCusto, int controle, int
86 {
87     int i;
88     permutacao[k] = ++controle;
89     if (controle == (melhorRota.Length - 1)) /* se gerou um caminho então verifica se ele é melhor */
90         melhorCaminho(grafo, melhorRota, ref melhorCusto, permutacao);

```

```

C# forçaBrutal.cs
89     if (controle == (melhorRota.Length - 1)) /* se gerou um caminho então verifica se ele é melhor */
90         melhorCaminho(grafo, melhorRota, ref melhorCusto, permutacao);
91     else
92         for (i = 1; i < melhorRota.Length; i++)
93             if (permutacao[i] == 0)
94                 permuta(permutacao, grafo, melhorRota, ref melhorCusto, controle, i);
95     controle--;
96     permutacao[k] = 0;
97 } /* fim permuta */
98
99 //-----
100
101 /* Gera os pesos dos arcos do grafo randomicamente e preenche
102    a matriz grafo->M, que é indexada pelos nomes dos vertices (cidades)
103    */
104 void montaGrafo(out Grafo grafo, int numCidades)
105 {
106     //int i, j;      // indexadores da matriz
107     int custo;      // pesos dos arcos {i, j}
108
109     Random randomizer = new Random();
110     grafo = new Grafo();
111     grafo.M = new int[numCidades, numCidades];
112
113     for (int i = 0; i < numCidades; i++)
114     {
115         for (int j = 0; j < numCidades; j++)
116         {
117             custo = randomizer.Next(MaxCusto) + 1;
118             if (i < j)
119                 grafo.M[i, j] = custo;

```

```

118         if (i < j)
119             grafo.M[i, j] = custo;
120         else
121             if (i == j)
122                 grafo.M[i, j] = 0;
123             else
124                 grafo.M[i, j] = grafo.M[j, i];
125     }
126 }
127
128 /* Para conferência: no formatting at all!!! */
129 Console.WriteLine("\nCidades e custos:\n");
130 for (int i = 0; i < numCidades; i++) // preenche a matriz grafo -> M
131     Console.Write(i + " ");
132 Console.WriteLine();
133 for (int i = 0; i < numCidades; i++)
134 { // preenche a matriz grafo -> M
135     Console.Write(i + " ");
136     for (int j = 0; j < numCidades; j++)
137     {
138         Console.Write(" " + grafo.M[i, j]);
139     }
140     Console.WriteLine();
141 }
142 } // fim montaGrafo

```

```

147     na rota da viagem do caixeiro e escolhe a melhor rota entre todas.
148     */
149     void geraEscolheCaminhos(ref int[] permutacao, Grafo grafo, Rota[] melhorRota, out int melhorCusto)
150     {
151         int controle = -1;
152         melhorCusto = int.MaxValue;
153
154         for (int i = 0; i < melhorRota.Length; i++)
155             melhorRota[i] = new Rota();
156
157         /* Gera os caminhos possíveis e escolhe o melhor, chamando a função recursiva
158            permuta */
159         permuta(permutacao, grafo, melhorRota, ref melhorCusto, controle, 1);
160     } /* fim GeraEscolheCaminhos */
161
162     //-----
163     /*
164     Imprime o melhor caminho para a viagem do caixeiro, bem como o custo total da
165     viagem.
166     */
167     void imprimeMelhorCaminho(int custo, Rota[] melhorRota)
168     {
169         int i; /* indexa o vetor que contém a rota */
170         Console.WriteLine("\nCUSTO MINIMO PARA A VIAGEM DO CAIXEIRO: " + custo);
171         Console.WriteLine("\nMELHOR CAMINHO PARA A VIAGEM DO CAIXEIRO:");
172         Console.WriteLine("\n          DE          PARA          CUSTO ");
173         for (i = 0; i < melhorRota.Length; i++)
174         {
175             Console.Write("          " + melhorRota[i].cidade1 + "          " + melhorRota[i].
176             Console.WriteLine("\n");
177

```

```

177         Console.WriteLine("\n");
178     } /* fim ImprimeMelhorCaminho */
179
180     //-----
181
182     void imprimeTempo(Stopwatch tempo)
183     {
184         Console.WriteLine("TEMPO DE EXECUÇÃO: ");
185         Console.WriteLine(tempo.Elapsed.Hours+" horas "+tempo.Elapsed.Minutes+" minutos "+tempo.Elapsed.Seconds+" segundos");
186     } /* fim Imprime tempo */
187
188     //-----
189
190
191     /* Coordena as partes do programa */
192     static void Main(string[] args)
193     {
194         int[] permutacao; /* vetor com uma possivel rota de viagem */
195         Rota[] melhorRota; /* contera' a melhor rota da viagem */
196         int numCidades, /* numero de vertices (cidades) do grafo */
197             melhorCusto; /* custo da viagem pelo grafo (pelas cidades) */
198
199         CaixaOtimizado caixaOtimizado = new CaixaOtimizado();
200         Grafo grafo; // = new Grafo(); /* matriz de adjacencia com o grafo */
201
202         Console.WriteLine("\n\n\t\tDigite o numero de cidades: ");
203         numCidades = int.Parse(Console.ReadLine());
204         caixaOtimizado.montaGrafo(out grafo, numCidades);
205
206         permutacao = new int[numCidades];
207         melhorRota = new Rota[numCidades];

```

```

200         Grafo grafo; // = new Grafo(); /* matriz de adjacencia com o grafo */
201
202         Console.WriteLine("\n\n\t\tDigite o numero de cidades: ");
203         numCidades = int.Parse(Console.ReadLine());
204         caixaOtimizado.montaGrafo(out grafo, numCidades);
205
206         permutacao = new int[numCidades];
207         melhorRota = new Rota[numCidades];
208
209         Stopwatch stopwatch = new Stopwatch();
210         stopwatch.Start(); // Inicia a contagem do tempo
211         caixaOtimizado.geraEscolheCaminhos(ref permutacao, grafo, melhorRota, out melhorCusto);
212         stopwatch.Stop(); // Encerra a contagem do tempo
213         caixaOtimizado.imprimeMelhorCaminho(melhorCusto, melhorRota);
214         caixaOtimizado.imprimeTempo(stopwatch);
215
216         Console.ReadKey(true);
217     } /* fim main */
218 }
219 }

```

- Algoritmo Guloso

```
1  import java.io.*;
2  import java.time.Duration;
3  import java.time.Instant;
4  import java.time.LocalDateTime;
5  import java.time.format.DateTimeFormatter;
6  import java.util.ArrayList;
7  import java.util.List;
8
9  public class TSPGuloso {
10
11     public static void main(String[] args) throws IOException {
12
13         String caminho = "src/CidadesTSP.txt";
14
15         int[][] tspMatrix = readFile(caminho);
16
17         Instant startTime = Instant.now();
18
19         findMinRoute(tspMatrix);
20
21         Instant endTime = Instant.now();
22         Duration totalTime = Duration.between(startTime, endTime);
23
24         System.out.println("\n"+totalTime);
25     }
26
27     @ public static int[][] createMatrix(int n){
28         int[][] matrix = new int[n][n];
29
30         for(int i = 0; i < n; i++){
31             for(int j = 0; j < n; j++){
32                 matrix[i][j]=-1;
33             }
34         }
35         return matrix;
36     }
```

```

45 @ public static int[][] readFile(String path){
46     String linha = "";
47     Reader r = null;
48     try {
49         r = new FileReader(path);
50         int c;
51         while ((c = r.read()) != -1) {
52             linha += (char) c;
53         }
54         String[] result = linha.split( regex: "\\n");
55
56         int tam = Integer.parseInt(result[0].trim());
57
58         int[][] matrix;
59         matrix = createMatrix(tam);
60
61         for(int i = 1; i< result.length;i++){
62             String columns[] = result[i].split( regex: " ");
63
64             for(int j=0;j<columns.length;j++){
65                 matrix[i-1][j] = Integer.valueOf(columns[j].trim());
66             }
67         }
68         //printMatrix(matrix);
69         return matrix;
70     } catch (FileNotFoundException ex) {
71         System.out.println(path + " nao existe.");
72     } catch (IOException ex) {
73         System.out.println("Erro de leitura de arquivo.");
74     } finally {
75         try {
76             if (r != null) {
77                 r.close();
78             }
79         } catch (IOException ex) {
80             System.out.println("Erro ao fechar o arquivo " + path);
81         }
82     }
83     return null;
84 }

```

```

85 @ static void findMinRoute(int[][] tsp) throws IOException {
86     int sum = 0;
87     int counter = 0;
88     int j = 0, i = 0;
89     int min = Integer.MAX_VALUE;
90     List<Integer> visitedRouteList
91         = new ArrayList<>();
92
93     visitedRouteList.add(0);
94     int[] route = new int[tsp.length];
95
96
97     while (i < tsp.length && j < tsp[i].length) {
98
99         if (counter >= tsp[i].length - 1) {
100             break;
101         }
102         //Se o caminho nao foi visitado e o custo for menor, atualiza o custo
103         if (j != i && !(visitedRouteList.contains(j))) {
104             if (tsp[i][j] < min) {
105                 min = tsp[i][j];
106                 route[counter] = j + 1;
107             }
108         }
109         j++;
110         // verifica todos os caminhos a partir do indice i
111         if (j == tsp[i].length) {
112             sum += min;
113             min = Integer.MAX_VALUE;
114             visitedRouteList.add(route[counter] - 1);
115             j = 0;
116             i = route[counter] - 1;
117             counter++;
118         }
119     }
120     // Atualiza a cidade final no array
121     i = route[counter - 1] - 1;
122
123     for (j = 0; j < tsp.length; j++) {
124
125         if ((i != j) && tsp[i][j] < min) {
126             min = tsp[i][j];
127             route[counter] = j + 1;
128         }
129     }
130     sum += min;
131     System.out.print("Custo minimo: ");
132     System.out.println(sum);
133
134     writeFile(sum,route);
135 }

```

```

136 @
137 public static void writeFile(int cost,int[] route) throws IOException {
138     String routeString = "";
139     DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-ddHH-mm-ss");
140
141     OutputStream os = new FileOutputStream( name: "resultadoTSPGuloso"+dtf.format(LocalDateTime.now())+".txt");
142     Writer wr = new OutputStreamWriter(os); // criação de um escritor
143     BufferedWriter br = new BufferedWriter(wr); // adiciono a um escritor de buffer
144
145     br.write(String.valueOf(cost));
146     br.newLine();
147     for (int i=0;i<route.length;i++){
148         routeString+= route[i] + " ";
149     }
150
151     br.write(routeString);
152     br.close();
153
154 }
155 }

```

Conclusão:

Em sùmula, concluímos que em nossos experimentos, o algoritmo que obteve a melhor abordagem para a resolução do problema, foi o algoritmo guloso. Por meio desta abordagem, obtivemos sempre a melhor escolha em cada ponto de decisão, sem provas de corretude e falhas para obter a melhor solução.

Utilizando a programação dinâmica, o consumo de tempo do algoritmo é, em geral, proporcional ao tamanho da tabela e necessita de grande espaço de memória. Contudo, este método se mostrou eficiente ao observarmos o tempo para a resolução do problema.

Ademais, implementamos a solução do problema do caixeiro viajante através da abordagem de força bruta, sendo este algoritmo aquele que compara todas as possibilidades de resposta para determinado conjunto de entradas e devolve a melhor solução obtida. Dessa forma, para muitos problemas reais, o número de possibilidades são de ordem extremamente elevada, tornando inviável a implementação de tal método.

Referências:

Algoritmos e Programação de Computadores

<https://www.ic.unicamp.br/~zanoni/teaching/mc102/2013-2s/aulas/aula21.pdf>

AULA 07 - Projeto e Análise de Algoritmos - Força Bruta - Backtracking

https://www.youtube.com/watch?v=elyagipwO8w&ab_channel=DaniloEler

Força Bruta e Backtracking - LPC I 2021

https://www.youtube.com/watch?v=EXOAZU19O8E&ab_channel=Programa%C3%A7%C3%A3oCompetitivaUNESP

AULA 07 - Projeto e Análise de Algoritmos - Força Bruta - Branch-and-Bound

https://www.youtube.com/watch?v=mPfWcKDuhac&ab_channel=DaniloEler

Algoritmos/Reconhecimento de padrões/Algoritmo de Força Bruta

https://pt.wikibooks.org/wiki/Algoritmos/Reconhecimento_de_padr%C3%B5es/Algoritmo_de_For%C3%A7a_Bruta

Travelling Salesman Problem via the Greedy Algorithm

<https://medium.com/ivymobility-developers/algorithm-a168afcd3611>

Algoritmos gulosos e Problema das Tarefas Compatíveis

<https://www.youtube.com/watch?v=PCMCGPknMwk>

Aula 07 - Algoritmos Gulosos

<https://www.youtube.com/watch?v=M7PvVQplxt0>

Travelling Salesman Problem

<https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/>

Traveling Salesman Problem – Dynamic Programming Approach

<https://www.baeldung.com/cs/tsp-dynamic-programming>