



**INSTITUTO FEDERAL  
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA**  
Fluminense

MINISTÉRIO DA  
**EDUCAÇÃO**



Felipe Soares, Kevin Perdomo, Mateus Terra

## **Projeto e Análise de Algoritmos**

**Algoritmo de busca local iterada aplicado ao problema do  
caixeiro viajante simétrico**

21 de Setembro de 2023

Campos dos Goytacazes

# Sumário:

## **1.0 Manual de uso do programa**

## **2.0 O problema**

## **3.0 A metaheurística**

### 3.1 Características da metaheurística

## **4.0 Resolução do problema**

### 4.1 Adaptando a lógica

### 4.2 Parâmetros

### 4.3 Leitura dos dados

### 4.4 Primeira Geração

#### 4.4.1 função: gerar\_solucão\_inicial

#### 4.4.2 função calcular\_custo:

#### 4.4.3: iterações

#### 4.4.4: função: busca\_local

#### 4.4.5: função: perturbar

## **5.0 Análise dos resultados**

### 5.1 Exemplo 01: 4 cidades

### 5.2 Exemplo 02: 10 cidades

### 5.3 Exemplo 03: 15 cidades

## 1.0 Manual de uso do programa

Para utilizar o programa, basta utilizar um arquivo denominado “matriz\_distancias.txt” dentro de uma pasta chamada “output” no mesmo diretório do programa e executá-lo.

Note que o arquivo “matriz\_distancias.txt” deve conter na primeira linha o número de vértices e o número de arestas da matriz de custos da viagem entre as cidades. As demais linhas armazenam a matriz distância  $D$ , onde o elemento  $d_{ij}$  armazena a distância da cidade  $i$  para a cidade  $j$  ( $1 \leq i, j \leq n$ ).

## 2.0 O problema

Para esse trabalho, estaremos procurando uma solução para o problema do caixeiro viajante simétrico, porém para isso primeiro precisamos entender melhor o que é esse problema.

O problema do caixeiro viajante (PCV) é um problema de otimização combinatória que envolve encontrar a rota mais rápida possível para que um viajante possa passar visitando todas as cidades exatamente uma vez e voltando para a cidade de origem, tendo o menor custo possível com as viagens (dado que cada rota entre cidades tem um custo próprio).

O problema do caixeiro viajante simétrico (PCVS) por sua vez segue os mesmos princípios, com a mudança de que os custos para a viagem de uma cidade qualquer  $A$  para uma cidade qualquer  $B$  devem ser os mesmos da cidade  $B$  para a cidade  $A$ , ou seja, o valor da ida de uma cidade a outra deve ser o mesmo valor da volta. Isso faz com que a matriz de custos entre cidades seja uma matriz simétrica.

Assim, o nosso objetivo é que:

$$\sum_{i=0}^n \sum_{j=0, j \neq i}^n d_{ij} \cdot x_{ij} = \min, \text{ onde } x = \{0, 1\}$$

Para:  $n$  (número total de cidades),  $d$  (distância),  $x$  (se existem arestas)

Obedecendo ao princípio de que:

$$\sum_{i=0, i \neq j}^n x_{ij} = 1, \text{ onde } x_i \in \{0, 1\}$$

Para:  $n$  (número total de cidades),  $d$  (distância),  $x$  (se existem arestas)

### 3.0 A metaheurística

A Busca Local Iterada, também conhecida pelo termo em inglês "Iterated Local Search" (ILS), refere-se a uma modificação na solução atual encontrada previamente.

Os métodos de busca local podem ficar presos em mínimos locais, onde não há vizinhos melhores disponíveis. Projetar o algoritmo de perturbação para o ILS não é uma tarefa fácil.

O objetivo principal é evitar ficar preso no mesmo mínimo local, e para alcançar isso, a perturbação precisa ser suficientemente forte para

*"O tempo é mais escasso que a memória" Perdomo, 2023*

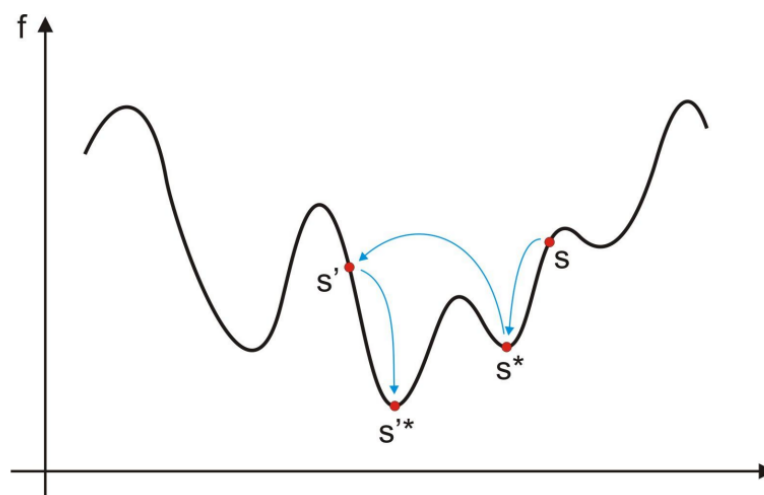
permitir que a busca local explore diferentes soluções, porém fraca o suficiente para evitar uma reinicialização aleatória.

### 3.1 Características da metaheurística

A abordagem iterativa da ILS envolve realizar uma perturbação na solução atual, levando a uma solução intermediária que será utilizada como novo ponto de partida para aprimorar o método. Além disso, um critério de aceitação adicional é empregado para determinar qual solução será mantida e dar continuidade ao processo.

De maneira mais ampla, o ILS pode ser aplicado quando dispomos de um método de otimização local. Ele demonstra maior eficiência se comparado com a busca com recomeços, onde são escolhidos aleatoriamente diversos pontos do espaço de soluções, e o algoritmo de busca local é aplicado a cada um deles.

Figura 1 - A Busca Local Iterada buscando uma solução a partir de um ótimo sub-ótimo mínimo local.



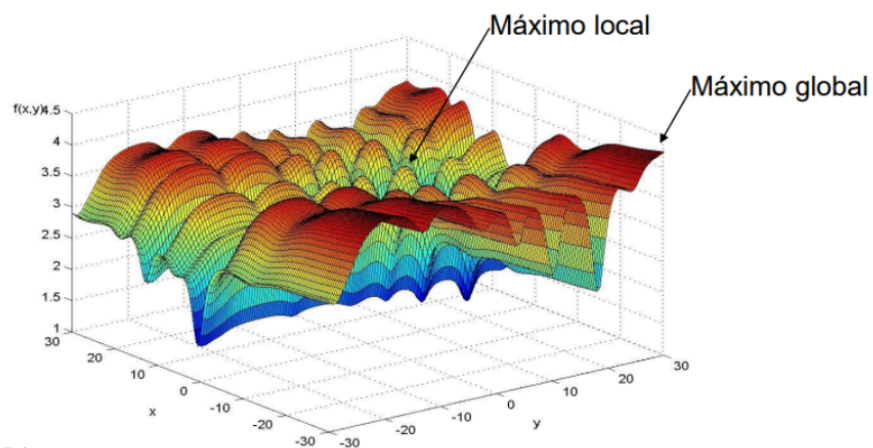
*BUSCA LOCAL ITERADA - Francisco A. M. Gomes*

## 4.0 Resolução do problema

### 4.1 Adaptando a lógica

Para resolvermos o problema do caixeiro viajante simétrico é necessário primeiro adaptarmos a lógica da metaheurística à lógica necessária para o nosso problema.

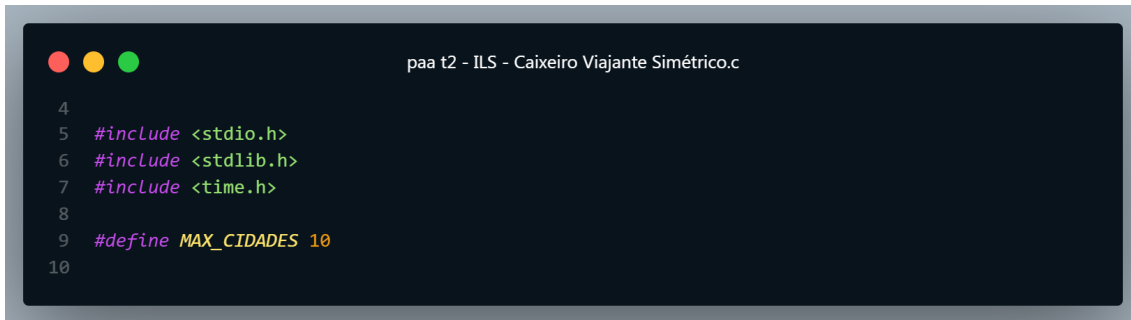
No ILS os resultados são aprimorados por meio de buscas feitas na vizinhança dos resultados atuais, essa vizinhança faz parte de um espaço maior que consideramos nosso espaço de possibilidades. No nosso cenário ele é composto por todas as possíveis rotas que o caixeiro pode tomar para sua viagem, independente dos custos. Estes custos representam uma terceira dimensão no espaço e serão a unidade que nosso algoritmo se baseará para saber se as combinações estão obtendo resultados melhores ou não.



### 4.2 Parâmetros

Inicialmente trabalhamos com uma quantidade máxima de cidades que podemos usar para obter nossos resultados. É uma variável de segurança

para impedir que valores maiores que o limite físico da máquina sejam inseridos.



```
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <time.h>
8
9 #define MAX_CIDADES 10
10
```

### 4.3 Leitura dos dados

Tendo os parâmetros definidos e a nossa lógica em mente, agora começaremos a leitura dos dados. Os dados se encontram em um arquivo “matriz\_distancias.txt” dentro de uma pasta chamada “output” no nosso diretório. Dentro deles estão os principais dados que variam do problema, eles são respectivamente:

|            |   |
|------------|---|
| 4 6        | <- Número de vértices   Número de arestas                   |
| 0 70 10 30 | <- Matriz de custos da viagem entre a cidade i e a cidade j |
| 70 0 50 80 |   |
| 10 50 0 60 |   |
| 30 80 60 0 |   |

*(valores unicamente ilustrativos)*

Para leitura dos dados utilizamos as funções de leitura de arquivos normais com uma verificação de erro caso o programa não consiga ler o arquivo. Após isso, lemos as 2 primeiras linhas, que possuem os dados referentes a quantidade de vértices e arestas, respectivamente. E terminamos a leitura com um loop que vai armazenar os valores e pesos de cada item em dois respectivos vetores.

```
paa t2 - ILS - Caixeiro Viajante Simétrico.c

73 int main() {
74     srand(time(NULL));
75
76     int num_cidades;
77     int num_arestas;
78
79     FILE *arquivo = fopen("matriz_distancias.txt", "r");
80     if (arquivo == NULL) {
81         printf("Erro ao abrir o arquivo.\n");
82         return 1;
83     }
84
85     fscanf(arquivo, "%d %d", &num_cidades, &num_arestas);
86
87     int matriz_distancias[MAX_CIDADES][MAX_CIDADES];
88
89     // Preenchendo a matriz de distancias com os dados do arquivo txt;
90     for (int i = 0; i < num_cidades; i++) {
91         for (int j = 0; j < num_cidades; j++) {
92             fscanf(arquivo, "%d", &matriz_distancias[i][j]);
93         }
94     }
95     fclose(arquivo);
```

## 4.4 Primeira Geração

```
paa t2 - ILS - Caixeiro Viajante Simétrico.c

97 int rota[MAX_CIDADES];
98 gerar_solucao_inicial(rota, num_cidades);
```

Para criação do primeiro vetor de resultados, criamos um resultado aleatório para que possamos começar de um ponto qualquer do nosso espaço de resultados. Esse resultado aleatório será gerado por meio da função “*gerar\_solucao\_inicial*”.



#### 4.4.1 função: *gerar\_solucão\_inicial*

```
paa t2 - ILS - Caixeiro Viajante Simétrico.c

21 void gerar_solucão_inicial(int rota[], int num_cidades) {
22     for (int i = 0; i < num_cidades; i++) {
23         rota[i] = i;
24     }
25
26     for (int i = num_cidades - 1; i >= 0; i--) {
27         int j = rand() % (i + 1);
28         int temp = rota[i];
29         rota[i] = rota[j];
30         rota[j] = temp;
31     }
32 }
```

essa função é composta por 2 partes principais:

```
paa t2 - ILS - Caixeiro Viajante Simétrico.c

22 for (int i = 0; i < num_cidades; i++) {
23     rota[i] = i;
24 }
```

A primeira parte que inicia o vetor “*rota*” com dados organizados de 0 a ‘*num\_cidades*’.

```
paa t2 - ILS - Caixeiro Viajante Simétrico.c

26 for (int i = num_cidades - 1; i >= 0; i--) {
27     int j = rand() % (i + 1);
28     int temp = rota[i];
29     rota[i] = rota[j];
30     rota[j] = temp;
31 }
```

*“O tempo é mais escasso que a memória” Perdomo, 2023*

A segunda parte fica responsável pela aleatoriedade dos números. Ela utiliza da função “*rand()*” com limites entre os valores 0 e *i*. Como pela natureza da função “*rand()*” não é possível que 2 casas do vetor ‘rotas’ recebam o mesmo número.

#### 4.4.2 função *calcular\_custo*:

```
paa t2 - ILS - Caixeiro Viajante Simétrico.c

12 int calcular_custo(int rota[], int num_cidades, int matriz_distancias[][MAX_CIDADES])
13 {
14     int custo = 0;
15     for (int i = 0; i < num_cidades - 1; i++) {
16         custo += matriz_distancias[rota[i]][rota[i + 1]];
17     }
18     custo += matriz_distancias[rota[num_cidades - 1]][rota[0]];
19     return custo;
20 }
```

A função ‘*calcular\_custo*’ possui uma nova variável que será o nosso contador de custo total da rota. Essa contagem é feita por meio de um loop que percorre todo o vetor, e soma o valor necessário para viagem da cidade *i* até a cidade ‘*i + 1*’ (esses valores estão na matriz ‘*matriz\_distancias*’).

#### 4.4.3: iterações

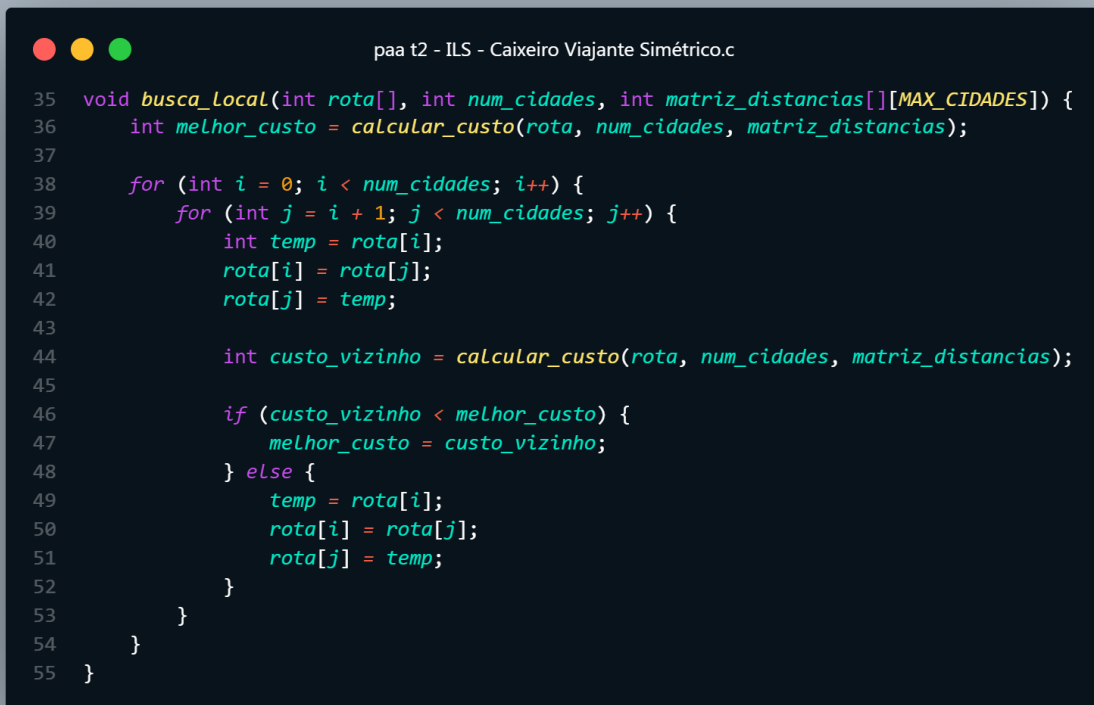
```
paa t2 - ILS - Caixeiro Viajante Simétrico.c

101 int num_iteracoes = 100;
102
103 for (int iteracao = 0; iteracao < num_iteracoes; iteracao++) {
104     busca_local(rota, num_cidades, matriz_distancias);
105     int custo_atual = calcular_custo(rota, num_cidades, matriz_distancias);
106     if (custo_atual < melhor_custo) {
107         melhor_custo = custo_atual;
108     }
109
110     perturbar(rota, num_cidades);
111 }
```

Após a criação do nosso resultado inicial, devemos fazer as buscas para encontrarmos resultados melhores que o resultado atual. Para isso começamos com a criação de uma variável que nos dirá quantas iterações faremos no nosso algoritmo, ou seja, quantas vezes vamos buscar por um resultado melhor que o resultado atual.

Com um loop que seguirá até que o todas as iterações sejam completadas, são realizadas as funções ‘*busca\_local*’, ‘*calcular\_custo*’ e ‘*perturbar*’ para cada iteração, acompanhadas de um a condicional que é responsável por comparar o custo do resultado atual com o custo do resultado obtido pela função ‘*busca\_local*’, se o resultado for melhor que o atual ele será o novo resultado atual do programa.

#### 4.4.4: função: *busca\_local*



```
paa t2 - ILS - Caixeiro Viajante Simétrico.c
35 void busca_local(int rota[], int num_cidades, int matriz_distancias[][MAX_CIDADES]) {
36     int melhor_custo = calcular_custo(rota, num_cidades, matriz_distancias);
37
38     for (int i = 0; i < num_cidades; i++) {
39         for (int j = i + 1; j < num_cidades; j++) {
40             int temp = rota[i];
41             rota[i] = rota[j];
42             rota[j] = temp;
43
44             int custo_vizinho = calcular_custo(rota, num_cidades, matriz_distancias);
45
46             if (custo_vizinho < melhor_custo) {
47                 melhor_custo = custo_vizinho;
48             } else {
49                 temp = rota[i];
50                 rota[i] = rota[j];
51                 rota[j] = temp;
52             }
53         }
54     }
55 }
```

A função '*busca\_local*' é a parte que fará o papel de busca de vizinhança necessária para o algoritmo ILS. Para isso inicia-se uma variável de melhor custo com a rota atual (no caso da primeira busca a rota atual será a rota gerada aleatoriamente).

A função começa calculando o custo da solução atual usando a função '*calcular\_custo*' e armazena esse custo na variável '*melhor\_custo*'. Esta variável manterá o custo da melhor solução encontrada durante a busca local.

Em seguida, a função entra em dois loops aninhados para explorar todas as possíveis trocas de cidades na rota. O primeiro loop itera sobre todas as cidades, e o segundo loop itera sobre as cidades subsequentes (começando da próxima após a atual).

Dentro desses loops, a função realiza o seguinte:

Troca as cidades na posição  $i$  e  $j$  na rota. Isso simula a troca de duas cidades na rota, o que pode potencialmente melhorar o custo da solução.

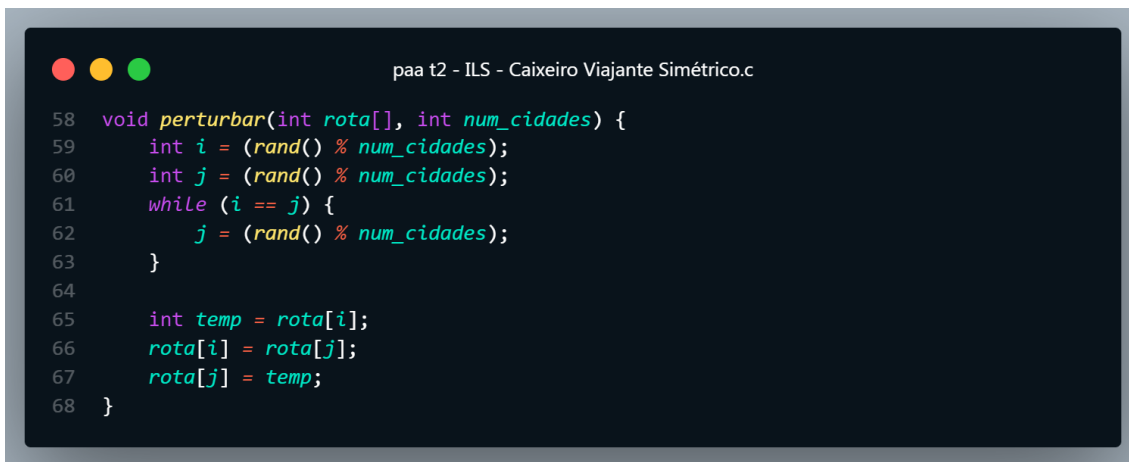
Calcula o custo da solução resultante após a troca usando a função '*calcular\_custo*' e armazena esse custo na variável '*custo\_vizinho*'.

Compara o '*custo\_vizinho*' com o '*melhor\_custo*'. Se '*custo\_vizinho*' for menor (ou seja, a troca melhorou a solução), então atualiza '*melhor\_custo*' com o valor de '*custo\_vizinho*'.

Caso contrário, se a troca não melhorar o custo, a função reverte a troca, restaurando a rota à sua forma original.

Essencialmente, esta função explora todas as possíveis trocas de cidades na rota atual e mantém a troca se ela resultar em uma melhoria no custo da solução. Isso é uma estratégia de busca local para encontrar uma solução ótima ou próxima à ótima para o (PCVS) a partir de uma solução inicial.

#### 4.4.5: função: *perturbar*



```
paa t2 - ILS - Caixeiro Viajante Simétrico.c

58 void perturbar(int rota[], int num_cidades) {
59     int i = (rand() % num_cidades);
60     int j = (rand() % num_cidades);
61     while (i == j) {
62         j = (rand() % num_cidades);
63     }
64
65     int temp = rota[i];
66     rota[i] = rota[j];
67     rota[j] = temp;
68 }
```

Dentro da função, a primeira coisa que é feita é escolher duas posições aleatórias  $i$  e  $j$  na rota atual. Essas posições representam duas cidades na rota que serão trocadas. O código para escolher valores aleatórios é “`rand() % num_cidades`”, que gera índices aleatórios dentro do intervalo de 0 a ‘`num_cidades - 1`’.

No entanto, a função verifica se  $i$  e  $j$  são iguais e, caso sejam, gera um novo valor para  $j$  até que  $i$  e  $j$  sejam diferentes. Isso garante que duas cidades diferentes sejam escolhidas para troca.

Em seguida, a função executa a troca real das duas cidades na posição  $i$  e  $j$  na rota. Isso perturba a solução atual, criando uma nova configuração da rota.

Essa função é o ponto chave no algoritmo ILS (Iterated Local Search) para introduzir diversificação na busca, permitindo que o algoritmo explore diferentes regiões do espaço de solução, o que pode ajudar a escapar de mínimos locais e potencialmente encontrar uma solução melhor para o (PCVS). A perturbação aleatória introduz um componente estocástico na busca, tornando-a mais exploratória.

## 5.0 Análise dos resultados

Após todas as iterações do ILS, o programa imprime o melhor custo encontrado. Este programa representa uma implementação básica do algoritmo ILS para resolver o (PCVS). Ele começa com uma solução inicial aleatória, realiza buscas locais em cada iteração e introduz perturbações para explorar diferentes soluções. O resultado final é o melhor custo encontrado após as iterações especificadas. Note que os resultados podem variar de uma execução para outra devido à aleatoriedade envolvida, porém o melhor resultado que o algoritmo encontra ao fim de cada busca local será sempre um valor menor que o anterior.

Para analisarmos os resultados utilizamos 3 exemplos diferentes, sendo um obtido no texto explicativo do trabalho e dois exemplos gerados por um algoritmo gerador de matrizes aleatórias.

### 5.1 Exemplo 01: 4 cidades



\*matriz\_distancias.txt - Bloco de Notas

Arquivo Editar Formatar Exibir Ajuda

4 6

00 07 01 03

07 00 05 08

01 05 00 06


03 08 06 00

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TER
PS C:\Users\Kevin\Desktop\Caixeiro Vi
Iterada\output'
PS C:\Users\Kevin\Desktop\Caixeiro Vi
Melhor custo encontrado: 17
PS C:\Users\Kevin\Desktop\Caixeiro Vi
```

Observe que o melhor resultado é encontrado rapidamente pelo algoritmo. Isso se dá pelo fato da quantidade de resultados ser pequena e a quantidade de iterações feitas pelo algoritmo ser alta. Logo o resultado encontrado após a primeira geração de resultados (que é aleatória) tem grandes chances de já ser o melhor resultado ou um resultado que se aproxime do melhor.

## 5.2 Exemplo 02: 10 cidades

Para um exemplo mais abrangente, que nos permite ver melhor a evolução do melhor resultado do algoritmo, vamos aumentar o número de cidades.

 matriz\_distancias.txt - Bloco de Notas


| Arquivo | Editar | Formatar | Exibir | Ajuda             |
|---------|--------|----------|--------|-------------------|
| 10      | 45     |          |        |                   |
| 00      | 08     | 10       | 09     | 01 06 03 02 07 06 |
| 08      | 00     | 02       | 04     | 06 02 03 07 08 06 |
| 10      | 02     | 00       | 09     | 08 08 04 05 09 10 |
| 09      | 04     | 09       | 00     | 08 10 06 02 02 10 |
| 01      | 06     | 08       | 08     | 00 04 03 03 02 04 |
| 06      | 02     | 08       | 10     | 04 00 06 10 07 10 |
| 03      | 03     | 04       | 06     | 03 06 00 02 06 06 |
| 02      | 07     | 05       | 02     | 03 10 02 00 09 08 |
| 07      | 08     | 09       | 02     | 02 07 06 09 00 05 |
| 06      | 06     | 10       | 10     | 04 10 06 08 05 00 |

```
PS C:\Users\Kevin\Desktop\Caixei  
melhor custo: 32  
melhor custo: 29  
melhor custo: 28  
melhor custo: 26  
melhor custo: 25  
melhor custo: 24  
melhor custo: 23  
Melhor custo encontrado: 23  
PS C:\Users\Kevin\Desktop\Caixei
```

Dessa vez os resultados foram mais abrangentes, com uma quantidade maior de melhores resultados encontrados, o que nos mostra que o algoritmo teve uma evolução maior desse resultado antes de chegar no melhor resultado.



## 5.3 Exemplo 03: 15 cidades

 matriz\_distancias.txt - Bloco de Notas

Arquivo Editar Formatar Exibir Ajuda

15 105

|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 00 | 10 | 02 | 10 | 03 | 02 | 09 | 06 | 04 | 03 | 05 | 04 | 09 | 03 | 01 |
| 10 | 00 | 09 | 09 | 09 | 08 | 09 | 01 | 04 | 02 | 03 | 05 | 10 | 01 | 08 |
| 02 | 09 | 00 | 03 | 02 | 06 | 04 | 05 | 04 | 06 | 05 | 02 | 03 | 02 | 10 |
| 10 | 09 | 03 | 00 | 07 | 08 | 02 | 08 | 01 | 06 | 03 | 09 | 01 | 01 | 06 |
| 03 | 09 | 02 | 07 | 00 | 02 | 02 | 04 | 07 | 05 | 06 | 07 | 09 | 08 | 05 |
| 02 | 08 | 06 | 08 | 02 | 00 | 06 | 06 | 03 | 04 | 01 | 07 | 07 | 05 | 03 |
| 09 | 09 | 04 | 02 | 02 | 06 | 00 | 04 | 01 | 03 | 06 | 09 | 10 | 05 | 06 |
| 06 | 01 | 05 | 08 | 04 | 06 | 04 | 00 | 07 | 06 | 06 | 07 | 03 | 02 | 09 |
| 04 | 04 | 04 | 01 | 07 | 03 | 01 | 07 | 00 | 07 | 05 | 02 | 05 | 10 | 02 |
| 03 | 02 | 06 | 06 | 05 | 04 | 03 | 06 | 07 | 00 | 02 | 07 | 04 | 09 | 07 |
| 05 | 03 | 05 | 03 | 06 | 01 | 06 | 06 | 05 | 02 | 00 | 01 | 03 | 06 | 06 |
| 04 | 05 | 02 | 09 | 07 | 07 | 09 | 07 | 02 | 07 | 01 | 00 | 03 | 08 | 06 |
| 09 | 10 | 03 | 01 | 09 | 07 | 10 | 03 | 05 | 04 | 03 | 03 | 00 | 04 | 07 |
| 03 | 01 | 02 | 01 | 08 | 05 | 05 | 02 | 10 | 09 | 06 | 08 | 04 | 00 | 09 |
| 01 | 08 | 10 | 06 | 05 | 03 | 06 | 09 | 02 | 07 | 06 | 06 | 07 | 09 | 00 |

```
PS C:\Users\Kevin\Desktop\Caixeiro
melhor custo: 29
melhor custo: 28
melhor custo: 27
Melhor custo encontrado: 27
PS C:\Users\Kevin\Desktop\Caixeiro
```

No exemplo de 15 cidades ele nos mostra que é possível encontrar o resultado de forma mais rápida, tudo depende do resultado gerado na primeira geração aleatória e como ele vai chegar perto do resultado.

Vale lembrar que mesmo levando menos resultados até chegar ao melhor encontrado ele ainda vai fazer todas as iterações definidas no algoritmo.

## 6.0 Referências

**“Resolução Do Problema Do Caixeiro Viajante Simétrico Com o Emprego de Planos-de-Corte Geométricos.”**. Maculan, Nelson Filho. 2002.

<https://cos.ufrj.br/index.php/pt-BR/publicacoes-pesquisa/details/15/572> . Acesso: 21/09/2023

**“Uma Análise experimental de Abordagens Heurísticas Aplicadas ao Problema do Caixeiro viajante”**. Prestes, Álvaro Nunes. 2006.

<https://repositorio.ufrn.br/bitstream/123456789/17962/1/AlvaroNP.pdf>. Acesso: 21/09/2023.

**“Busca Local Iterada”**. Abreu, Kelly Rodrigues. 2007.

[https://www.inf.ufpr.br/aurora/disciplinas/topicosia2/downloads/trabalhos/Apresentacao\\_ILS.pdf](https://www.inf.ufpr.br/aurora/disciplinas/topicosia2/downloads/trabalhos/Apresentacao_ILS.pdf). Acesso: 21/09/2023.

**“BUSCA LOCAL ITERADA (ILS – ITERATED LOCAL SEARCH)”**. Gomes, Francisco A. M. 2009.

<https://www.ime.unicamp.br/~chico/mt852/slidesils.pdf>. Acesso: 21/09/2023

**“Introdução às Metaheurísticas”**. Carvalho, Rafael Lima. Almeida, Tiago da Silva. Rocha, Marcelo Lisboa. 2020.

<https://repositorio.uft.edu.br/bitstream/11612/2486/1/Introdu%C3%A7%C3%A3o%20%C3%A0s%20Metaheur%C3%ADsticas.pdf>. Acesso: 21/09/2023.

**“ABORDAGENS HEURÍSTICAS APLICADAS AO THIEF ORIENTEERING PROBLEM”**. Faêda, Leonardo Moreira. 2022.

<https://www.locus.ufv.br/bitstream/123456789/30794/1/texto%20completo.pdf>. Acesso: 21/09/2023.

**“Heurísticas para o Problema da Partição Cromática de Custo Mínimo”**. Freire, Philippe Leal dos Santos. 2018.

<https://www.ic.uff.br/wp-content/uploads/2021/09/872.pdf>. Acesso: 21/09/2023.