SERVIÇO PÚBLICO FEDERAL · MINISTÉRIO DA EDUCAÇÃO

UNIVERSIDADE FEDERAL DE VIÇOSA · UFV

CAMPUS FLORESTAL

**Trabalho 2 - AEDS 1**

**O problema do caixeiro viajante assimétrico.**

Aléxia Karoline Augusta Germano Silva [5373]

Ana Luísa Moreira Rodrigues [5389]

Lucas da Costa Moreira [5377]

Florestal - MG

2023

**Sumário**

# Sumário

[Sumário 2](#_Toc149693588)

[1. Introdução 3](#_Toc149693589)

[2. Organização 3](#_Toc149693590)

[3. Desenvolvimento 4](#_Toc149693591)

[4. Compilação e Execução 7](#_Toc149693592)

[5. Resultados 7](#_Toc149693593)

[6.Conclusão 12](#_Toc149693596)

# **1. Introdução**

O trabalho consiste na resolução do problema do caixeiro viajante assimétrico na linguagem C, tentando obter o melhor (mais curto) caminho entre N cidades, aplicando o conceito ensinado em sala de aula de Complexidade Computacional.

A abordagem utilizada para o programa foi o teste de força bruta em todas as permutações possíveis com a quantidade de cidades menos 1 (N-1)!, já que sabemos que uma das cidades será a inicial e a final. O algoritmo de permutação utilizado foi o algoritmo de geração de permutações por trocas, que utiliza de recursividade para obter todas as permutações possíveis de um conjunto de números.

Um TAD (Tipo Abstrato de Dados), nomeado de Matriz, foi utilizado para organizar o código e facilitar o uso de funções por meio dele, passando como argumento para elas apenas o próprio TAD a ser modificado.

# **2. Organização**

A figura exibe as pastas que separam e organizam o projeto. Na pasta TP2\_ANALUISA-5389\_ALEXIA-5373\_LUCASMOREIRA-5377/ encontra-se a implementação do projeto e o TAD matriz.

A black screen with white text

Description automatically generated

Figura 1 - Repositório do projeto.

No TAD\_matriz temos uma struct chamada Matriz, que é um tipo de dado que armazena diversas informações relevantes para o desenvolvimento do código. As variáveis dessa struct irão armazenar o número de cidades digitadas pelo usuário ou contidas no arquivo de leitura, a cidade que deve ser o ponto de partida e o ponto final, temos a variável distâncias que será uma matriz alocada dinamicamente com a distância de uma cidade a outra, a variável melhorCaminho irá armazenar a rota que possue menor distância total, a variável dist\_total irá armazenar a menor distância possível ao percorrer as cidades e voltar à cidade de início, , a variável caminhoOriginal será um vetor que irá armazenar todas as cidades exceto a cidade que é o ponto de partida e o ponto final.

# **3. Desenvolvimento**

Em um primeiro momento, foi necessário procurar na web um algoritmo de permutação para um conjunto de N elementos, que produzisse todas as possíveis permutações. O algoritmo escolhido foi uma geração de permutações por trocas, considerando que durante as pesquisas, percebemos que ele foi uns dos mais recomendados por sua simplicidade e clareza de implementação (não necessariamente por sua eficiência, considerando que o algoritmo é de ordem exponencial e será usado recursivamente).

**3.1 Funcionamento do algoritmo**

1. A função Permuta é chamada com a posição inicial (inf) igual a 0 e a posição final (sup) igual ao tamanho do caminho menos 1.
2. A função verifica se a posição inicial é igual à posição final. Se sim, isso significa que uma permutação foi gerada completa, e então a distância é calculada para a permutação atual utilizando a função Calcular\_Distancia().
3. Se a posição inicial não for igual à posição final, a função entra em um loop que itera por todas as posições possíveis a partir da posição atual até a posição final.
4. Para cada posição no loop, os elementos nas posições inf e i na matriz são trocados utilizando a função Troca.
5. Em seguida, a função é chamada recursivamente com a posição inicial aumentada em 1 (inf + 1), o que significa que agora a permutação será gerada para o restante dos elementos.
6. Após a chamada recursiva, a troca é desfeita para restaurar a matriz ao seu estado original. Esse passo é crucial para garantir que o algoritmo explore todas as permutações possíveis.

**3.2 Vetor do melhor caminho e cálculo da menor distância**

Nesse módulo será explicado como foi feito o vetor de melhor caminho, que começa e termina com a primeira cidade, todos esses processos de armazenar a permutação com menor caminho e formatar o vetor afim de que o ciclo de ida e volta a primeira cidade seja cumprido, foram realizados na função Calcular\_Distancia(). Para isso, é importante enfatizar que ela é chamada na função Permuta(), para calcular a distância de cada permutação gerada pela função e verificar se ela é menor que a calculada anteriormente.

No momento de armazenar, foi utilizado os componentes da estrutura Matriz. De forma que o “caminhoOriginal” seria o vetor da permutação sem a primeira cidade inclusa, a “DistanciaCalculada” seria uma variável inteira para armazenar a distância total e “distancias” seria a matriz com as distâncias armazenadas, assim, foi necessária apenas armazenar na variável “DistanciaCalculada” a distância (relacionada pelo índice da matriz) da primeira cidade até o primeiro item do vetor de permutação, adicionar as outras distâncias do vetor de permutação e na última seria armazenar a distância do último item do vetor “caminhoOriginal” até a primeira cidade novamente. Em seguida, foi verificado se “DistanciaCalculada” era menor que o atributo “dist\_total” da estrutura Matriz, se fosse, os valores da primeira cidade e do vetor “caminhoOriginal” eram passados para o vetor “melhorCaminho” utilizando a referência da variável “num\_cidades” da estrutura Matriz no início e no fim, e os itens do vetor “caminhoOriginal” foram passados através de um loop que começa a iteração em 1 (segundo elemento do vetor “melhorCaminho”).

**3.3 Preenchimento da matriz de distâncias aleatórias**

A função Set\_Distancias é responsável por preencher a matriz de distâncias na estrutura Matriz com valores aleatórios que representam as distâncias entre cidades. Em resumo, a função preenche a matriz de distâncias com distâncias aleatórias entre cidades, garantindo que a diagonal principal seja composta por zeros (representando a distância de uma cidade para ela mesma) e os outros elementos recebam distâncias aleatórias variando de 1 a 100.

**3.3 Preenchimento da matriz de distâncias por arquivo**

A função Set\_Arquivo desempenha o papel de abrir um arquivo, ler a distribuição de distâncias armazenada nele e preencher a matriz de distâncias na estrutura Matriz. Após a abertura do arquivo e leitura do número de cidades, é feito a configuração da estrutura Matriz. Em seguida, A função entra em um loop duplo para percorrer todas as posições da matriz de distâncias. Se i for igual a j, isso significa que está na diagonal principal, e o valor é fixado como zero (representando a distância de uma cidade para ela mesma). Caso contrário, a função lê o valor do arquivo e o atribui à posição correspondente na matriz. O loop assegura que cada valor lido do arquivo seja atribuído à posição apropriada na matriz, considerando a lógica para a diagonal principal. Por fim, o arquivo é fechado com “fclose” para liberar recursos do sistema.

**3.4 Matriz de distâncias: criação e motivo**

A matriz foi armazenada com um ponteiro de ponteiro (criação de uma matriz com alocação dinâmica) chamado "distancias". Primeiro inicializamos a matriz com um malloc referente a quantidade de linhas necessárias para armazenar as cidades. Após isso, criamos um laço for que inicializará as colunas da matriz em cada linha, com o mesmo número sendo utilizado na criação de linhas e colunas (N cidades).Notamos que a criação e utilização da função do cálculo de distancias seria muito mais simples caso usássemos uma matriz, pois a mesma poderia ter a distância de cada cidade calculada de forma rápida.

**3.5 Cálculo da primeira cidade**

A forma utilizada para saber qual cidade começar foi criar um laço “for” que itera 4 vezes, onde cada vez que itera aumenta a variável soma pelo módulo de 10 do número da matricula de cada um, e depois divide cada matrícula por 10. Com isso, a cada vez que o “for” itera, estaremos somando o número mais à direita de cada matricula e deslocando a posição do número mais à direita para a esquerda, somando todos os dígitos e posteriormente fazendo com que o resultado dessa soma % N (número de cidades) seja a cidade inicial.

# **4. Compilação e Execução**

Comandos para executar e compilar o código.



# **5. Resultados**

Especificações do computador utilizado para os testes:

Processador: Intel Core i7 10750h

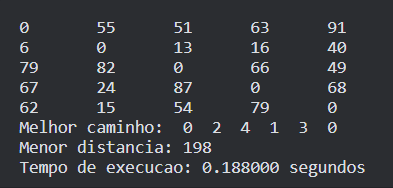
Placa de vídeo: GTX 1650

Memória RAM: DDR4-3200MHz

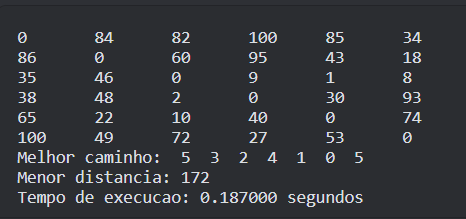
Armazenamento: SSD 512GB

Sistema Operacional: Windows 10

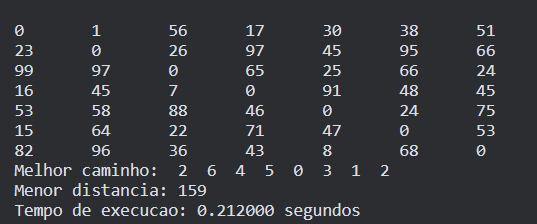
Testes realizados com matriz aleatória exibição por arquivo .txt com 5>= N <= 15:



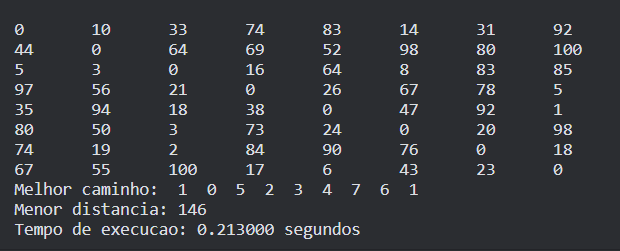
**Figura 1: Arquivo.txt quando N = 5**



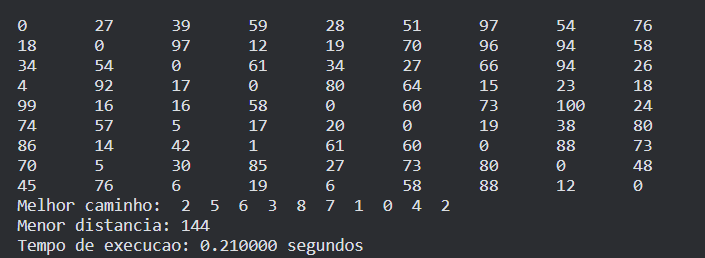
**Figura 2: Arquivo.txt quando N = 6**



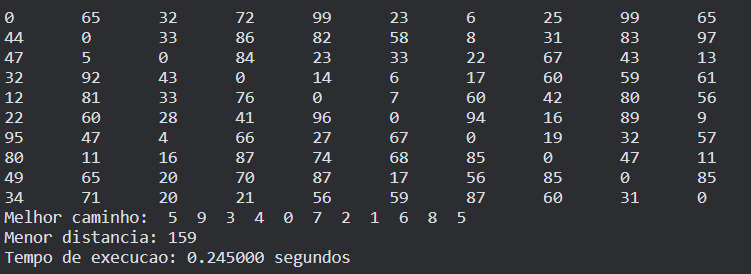
**Figura 3: Arquivo.txt quando N = 7**

****

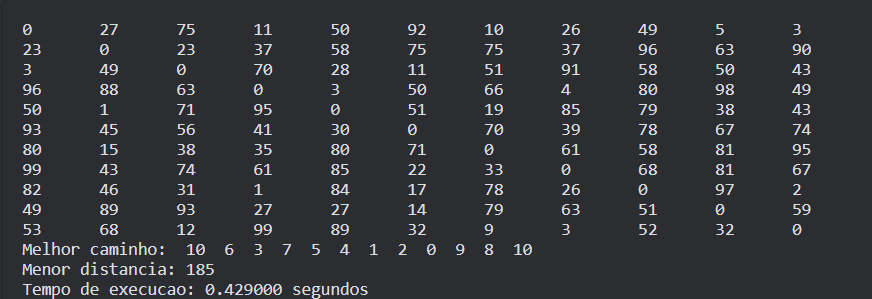
**Figura 4: Arquivo.txt quando N = 8**



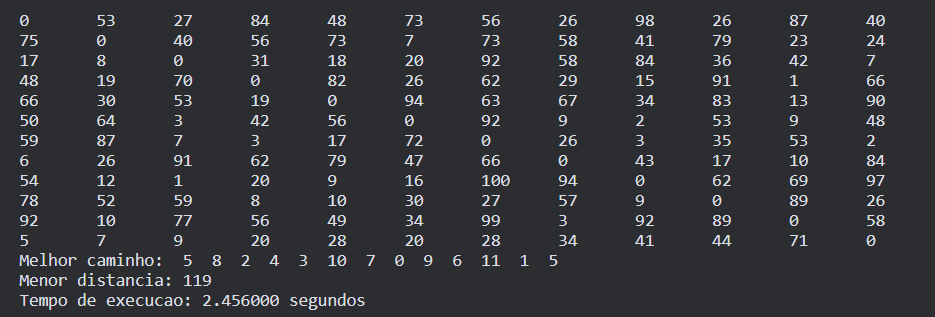
**Figura 5: Arquivo.txt quando N = 9**



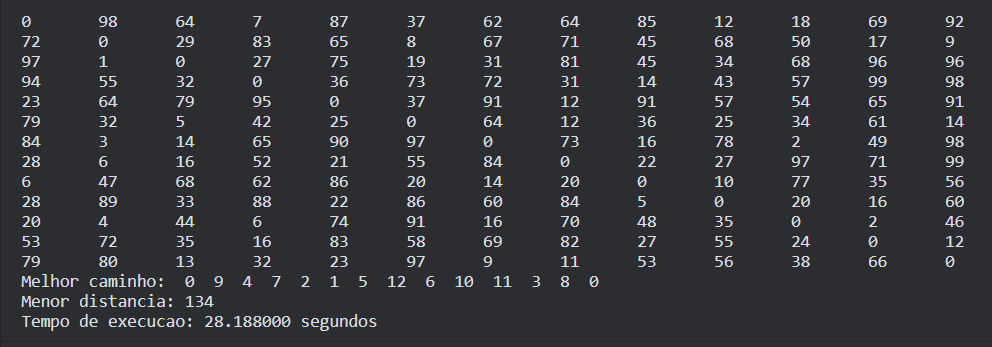
**Figura 6: Arquivo.txt quando N = 10**



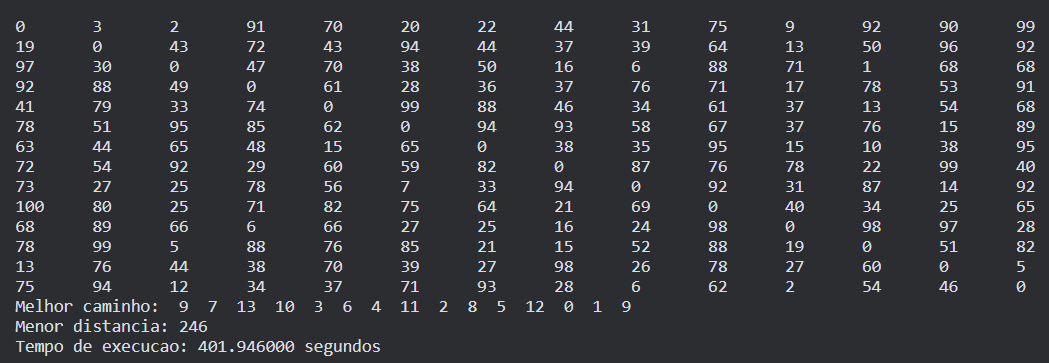
**Figura 7: Arquivo.txt quando N = 11**



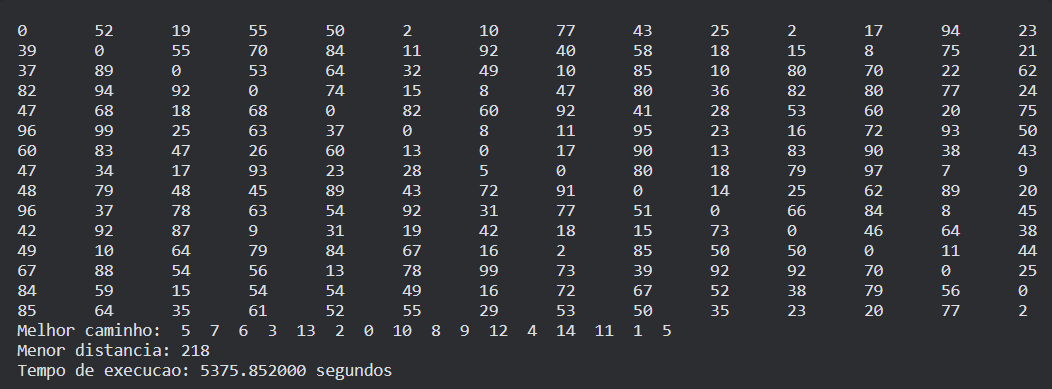
**Figura 8: Arquivo.txt quando N = 12**



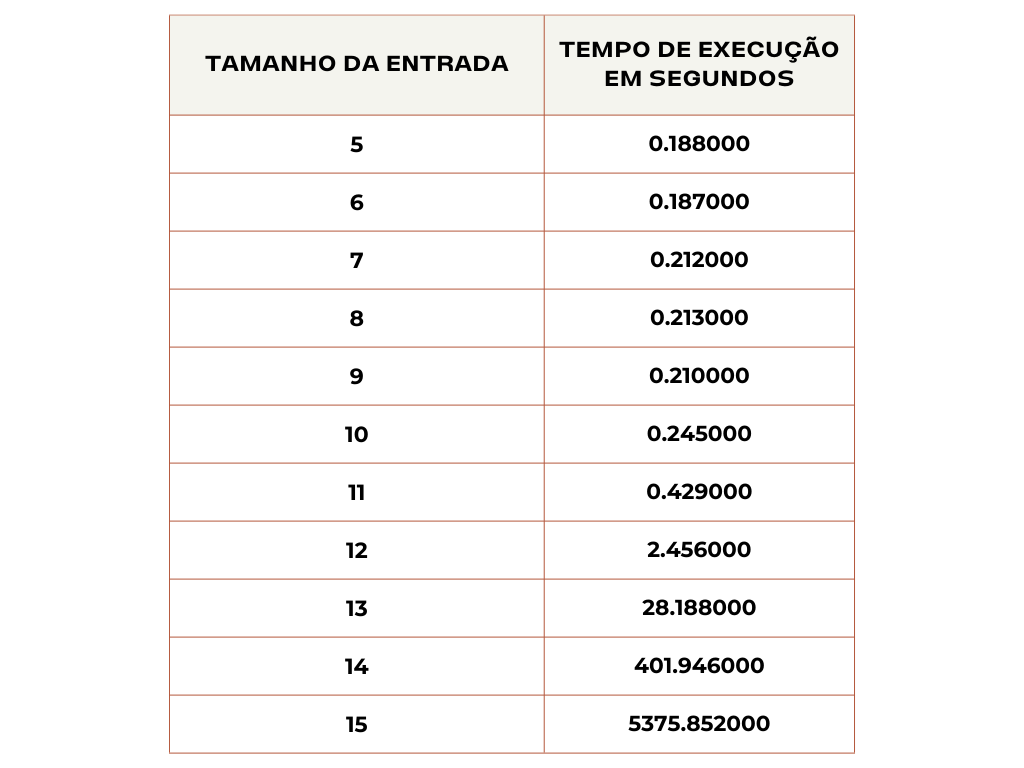
**Figura 9: Arquivo.txt quando N = 13**



**Figura 10: Arquivo.txt quando N = 14**



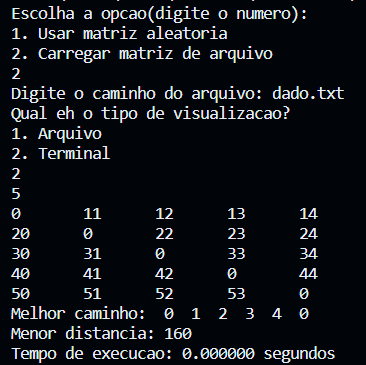
**Figura 11: Arquivo.txt quando N = 15**



# 

**Figura 12: Tabela com o tempo de execução e n° de entrada para matriz aleatórias**

# Durante o teste com o arquivo do moodle, utilizar o mesmo arquivo estava dando erro na leitura, portanto nós criamos outro .txt com os mesmos dados e realizamos o teste.



**Figura 13: Teste c/ dado.txt que possui os mesmos valores do arquivo do moodle.**

# **6.Conclusão**

Caso fossemos contratados por uma transportadora para desenvolver um sistema que encontrasse a menor rota a ser percorrida por seus caminhões, saindo e chegando do galpão de estoque e percorrendo uma única vez um conjunto N de localidades, não utilizariamos a solução desenvolvida neste trabalho, pois, em nossa opinião, o código utilizado neste trabalho não seria o mais eficiente no contexto da resolução do problema no caixeiro viajante, pois o algoritmo que usamos possue a complexidade O(n!) (calculamos como (n-1)! já que sabemos que uma das cidades será repetida no inicio e no final), enquanto o algoritmo Held-Karp, um algoritmo feito para o problema do caixeiro viajante, tem a complexidade O(n²), sendo mais eficiente ao achar o melhor caminho. Contudo, caso o número de cidades seja menor que 6 cidades, o algoritmo utilizado por nós se revela mais eficiente (4\*3\*2\*1 < 5 \* 5 => 24 < 25).

**7. Referências**

Para versionar o projeto foi utilizado o Github [1].

[1] Github. Disponível em: <<https://github.com/>>