SERVIÇO PÚBLICO FEDERAL · MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE VIÇOSA · UFV
CAMPUS FLORESTAL

Trabalho 2 - AEDS 1

Caixeiro Viajante Assimétrico

Rafaella Ferreira [05363]
Luana Tavares[05364]
Aline Cristina [05791]

Florestal- MG
2023

SUMÁRIO

1. Introdução

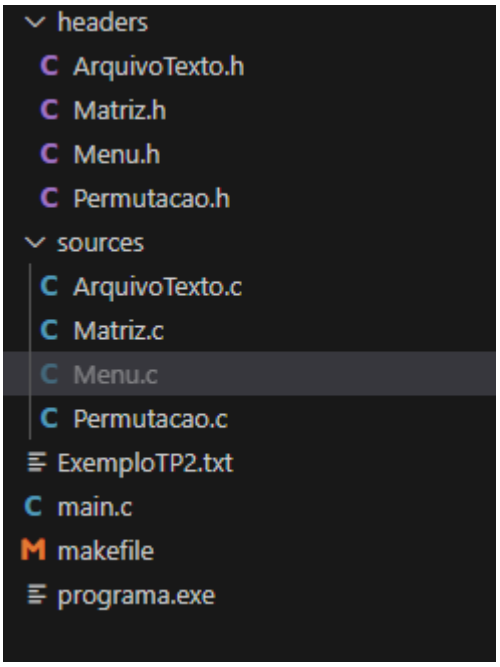
O relatório descreve o desenvolvimento e a análise de um programa implementado para resolver o Problema do Caixeiro Viajante Assimétrico. O objetivo deste trabalho prático foi avaliar o impacto causado pelo desempenho dos algoritmos na sua execução real.

A estratégia adotada para este trabalho envolveu a implementação de um algoritmo de permutação para um conjunto de N elementos, onde N representa o número de cidades no problema. O programa baseou-se na geração de todas as possíveis combinações entre as cidades para encontrar o caminho de menor distância.

Este relatório apresentará detalhadamente o método utilizado para resolver o problema, descrevendo o algoritmo de permutação implementado, a estrutura de dados utilizada para armazenar as distâncias entre as cidades e como o programa calcula o custo dos caminhos.

2. Organização

Fizemos a organização do código através de pastas. Separamos os arquivos .c na pasta sources e os arquivos .h na pasta headers. Optamos por fazer a divisão em pastas para obtermos melhor organização e controle do código que estamos criando. Ademais, a divisão do código em pastas é uma prática fundamental em qualquer desenvolvimento de software. Na imagem abaixo, mostramos como está a divisão. O arquivo main.c, makefile, programa.exe e ExemploTp2.txt, foram deixados fora de ambas as pastas.



(Figura 1: Divisão das pastas do código).

3. Desenvolvimento

O desenvolvimento do trabalho relacionado ao Problema do Caixeiro Viajante Assimétrico envolveu as seguintes etapas:

Entendimento do Problema: Antes de começar o trabalho prático, iniciamos com a análise e leitura do que foi pedido para ser feito. Assim, tivemos mais facilidade em assimilar o que foi pedido.

Implementação do Algoritmo de Permutação: Foi realizada uma busca na web para encontrar um algoritmo de permutação adequado para um conjunto de N elementos. O algoritmo selecionado foi adaptado e implementado em linguagem C. Este algoritmo gerou todas as possíveis combinações dos N−1 elementos, onde N representa o número de cidades.

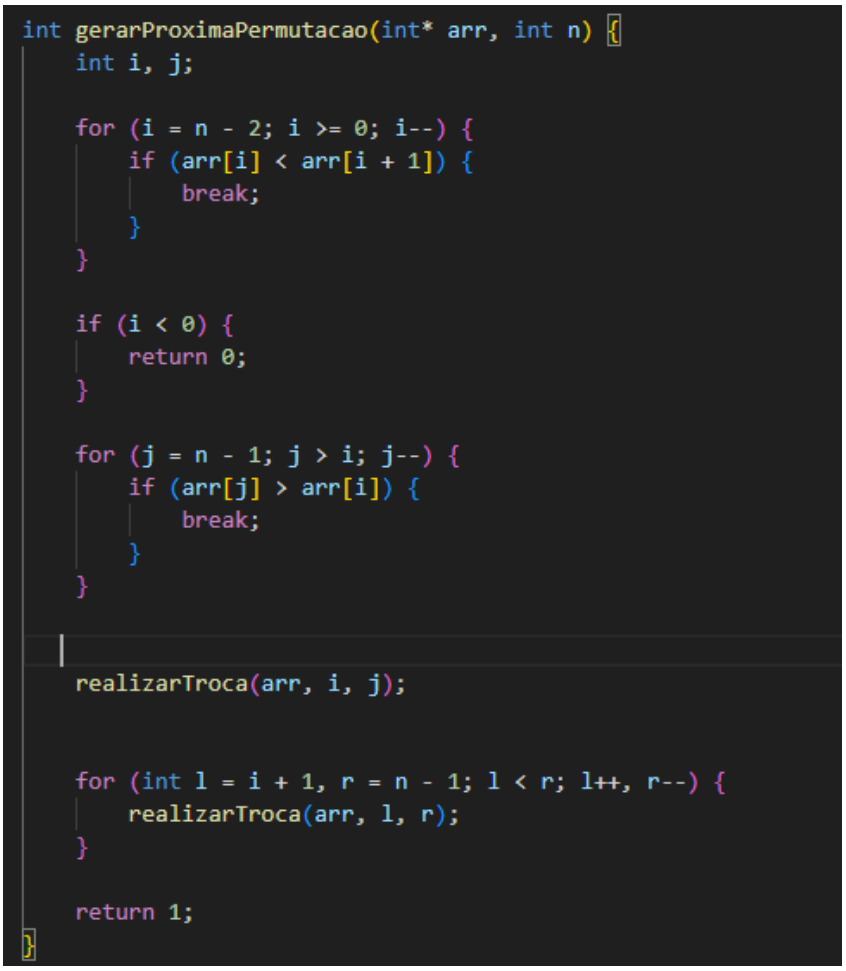
Estrutura de Dados para Armazenar Distâncias: Uma matriz M de tamanho NxN foi utilizada para armazenar as distâncias entre as N cidades do problema. Para a ligação entre duas cidades i e j, os valores das distâncias entre elas foram inseridos nas posições M[i][j]. As posições M[i][i] da matriz foram preenchidas com 0, indicando a distância de uma cidade para ela mesma.

Geração de Entradas e Ponto de Partida: O usuário tem a opção de escolher dois modos de acesso ao sistema : modo arquivo (onde é disponibilizado um arquivo de texto para teste), ou escolher o modo aleatório,no qual o usuário vai escolher todas as entradas do programa. Além disso, um número foi calculado com base na soma dos dígitos dos números de matrícula dos membros do grupo. Este valor foi utilizado como ponto de partida e retorno, sendo excluído das permutações geradas.

3.1. Algoritmo de Permutação

Optamos pelo algoritmo de permutação **lexicográfica** devido à sua simplicidade e eficiência para gerar todas as possíveis combinações. Parecido com um quebra-cabeça, ele nos permitiu experimentar todas as combinações possíveis das cidades de uma forma organizada e completa. É como organizar peças de um quebra-cabeça até encontrar a imagem perfeita.

A função **gerarProximaPermutacao** é como o caixeiro chegando a uma interseção onde ele precisa decidir qual cidade visitar a seguir. O **vetor [arr]** representa as cidades que o caixeiro ainda não visitou. O algoritmo busca a próxima "interseção" no seu caminho, onde ele pode escolher uma cidade diferente para visitar a seguir. Ele faz isso encontrando a primeira cidade que pode ser trocada com uma cidade subsequente para melhorar o percurso. Em seguida, o algoritmo inverte a ordem das cidades restantes, simulando o caixeiro voltando atrás na sua decisão para tentar um caminho alternativo. Assim, o **gerarProximaPermutacao** funciona como o caixeiro explorando todas as possíveis ordens de visita às cidades.



```
int gerarProximaPermutacao(int* arr, int n) {
    int i, j;

    for (i = n - 2; i >= 0; i--) {
        if (arr[i] < arr[i + 1]) {
            break;
        }
    }

    if (i < 0) {
        return 0;
    }

    for (j = n - 1; j > i; j--) {
        if (arr[j] > arr[i]) {
            break;
        }
    }

    realizarTroca(arr, i, j);


    for (int l = i + 1, r = n - 1; l < r; l++, r--) {
        realizarTroca(arr, l, r);
    }

    return 1;
}
```

(Figura 2: Função gerarProximaPermutacao)

3.2. Vetor Permutação

1. **Identificação da Cidade Inicial:** Antes de criarmos a permutação, calculamos uma cidade inicial com base no resto da soma das matrículas. Essa cidade inicial é importante, pois é ela que garante que o caixeiro comece e termine um trajeto (observamos isso na quarta linha do código com **cidadeInicial = calcularCidadeInicial(restoSomaMatriculas)**)
2. **Geração do Vetor Permutado:** O vetor permutado é colocado para incluir todas as cidades exceto a que iniciamos. Excluímos a cidade inicial à medida que preenchemos o vetor. Observamos isso no laço do for, onde está abaixo do comentário em que informamos a exclusão da cidade inicial. Assim geramos o vetor permutado.

3. **Adição da Cidade Inicial no Início e no Final:** A função **adicionarCidadeInicialNoVetor** cria um novo vetor com um tamanho maior para acomodar a cidade inicial no início e no final. As outras cidades da permutação são copiadas para o novo vetor a partir da segunda posição.

```c
int* gerarVetorPermutado(int numeroDeCidades, int restoSomaMatriculas) {
    int tamanhoVetor = numeroDeCidades - 1;
    int* vetorPermutado = (int*)malloc(sizeof(int) * tamanhoVetor);
    int cidadeInicial = calcularCidadeInicial(restoSomaMatriculas);

    int j = 0;
    for (int i = 0; i < numeroDeCidades; i++) {
        if (i == cidadeInicial) {
            continue;
        }
        vetorPermutado[j] = i;
        j++;
    }


    return vetorPermutado;
}
```

(Figura 3: Função gerarVetorPermutado)

```c
int* adicionarCidadeInicialNoVetor(int* vetorPermutado, int numeroDeCidades, int cidadeInicial) {
    int tamanhoNovoVetor = numeroDeCidades + 1;
    int* novoVetor = (int*)malloc(sizeof(int) * tamanhoNovoVetor);

    novoVetor[0] = cidadeInicial;

    for (int i = 1; i < numeroDeCidades; i++) {
        novoVetor[i] = vetorPermutado[i - 1];
    }

    novoVetor[numeroDeCidades] = cidadeInicial;

    return novoVetor;
}
```

(Figura 4: Função adicionarCidadeInicialNoVetor)

3.3 Armazenamento Matriz

1. **Alocação de Memória para a Matriz:** Quando a função **criarMatriz** é chamada, ela aloca dinamicamente memória para a estrutura **Matriz** e para a matriz de distâncias. O ponteiro para a matriz é inicializado para apontar para um bloco de memória alocado dinamicamente.
2. **Preenchimento da Matriz com Distâncias Aleatórias:** A função **gerarMatrizAleatoria** preenche a matriz com distâncias aleatórias entre as cidades. Ela usa a função **rand()** para gerar números aleatórios e preenche a matriz com valores entre 0 e 99 (por causa de **rand() % 100**).
3. **Cálculo da soma das distâncias de uma permutação:** essa função é usada para calcular a distância total percorrida em uma rota que passa pela cidades que foram permutadas. Ela soma a distância de todas as permutações e retorna a menor entre elas.
4. **Liberando Memória da Matriz:** Quando a matriz não é mais necessária, a função **destruirMatriz** libera toda a memória alocada para a matriz e a estrutura Matriz.

```
Matriz* criarMatriz(int numeroDeCidades) {
    Matriz* matriz = (Matriz*)malloc(sizeof(Matriz));
    if (matriz == NULL) {
        return NULL;
    }
    matriz->numeroDeCidades = numeroDeCidades;
    matriz->matrizDistancias = (int**)malloc(numeroDeCidades * sizeof(int*));
    if (matriz->matrizDistancias == NULL) {
        free(matriz);
        return NULL;
    }
    for (int i = 0; i < numeroDeCidades; i++) {
        matriz->matrizDistancias[i] = (int*)malloc(numeroDeCidades * sizeof(int));
        if (matriz->matrizDistancias[i] == NULL) {
            for (int j = 0; j < i; j++) {
                free(matriz->matrizDistancias[j]);
            }
            free(matriz->matrizDistancias);
            free(matriz);
            return NULL;
        }
    }

    return matriz;
}
```

(Figura 5: Função CriarMatriz)

```c
void gerarMatrizAleatoria(Matriz* matriz) {
    srand(time(NULL));

    for (int i = 0; i < matriz->numeroDeCidades; i++) {
        for (int j = 0; j < matriz->numeroDeCidades; j++) {
            if (i == j) {
                matriz->matrizDistancias[i][j] = 0;
            } else {
                int distanciaCidades = rand() % 100;
                matriz->matrizDistancias[i][j] = distanciaCidades;
            }
        }
    }

    // Imprime a matriz diretamente no terminal
    printf("Matriz de Distâncias:\n");
    for (int i = 0; i < matriz->numeroDeCidades; i++) {
        for (int j = 0; j < matriz->numeroDeCidades; j++) {
            printf("%d\t", matriz->matrizDistancias[i][j]);
        }
        printf("\n");
    }
}
```

(Figura 6: Função gerarMatrizAleatoria)

```c
int calcularSomaDistancias(Matriz* matriz, int* permutacao) {
    int resultado = 0;
    for (int i = 0; i < matriz->numeroDeCidades; i++) {
        int j = (i + 1) % matriz->numeroDeCidades;
        resultado += matriz->matrizDistancias[permutacao[i]][permutacao[j]];
    }
    return resultado;
}
```

(Figura 7: Função calcularSomaDistancias)

```c
void destruirMatriz(Matriz* matriz) {
    for (int i = 0; i < matriz->numeroDeCidades; i++) {
        free(matriz->matrizDistancias[i]);
    }
    free(matriz->matrizDistancias);
    free(matriz);
}
```

(Figura 8: Função destruirMatriz)

3.4 Preenchimento da matriz

O preenchimento da matriz é feito pela função **gerarMatrizAleatoria**.Ela percorre cada célula da matriz (ou seja, cada par de cidades) e preenche com um número aleatório entre 0 e 99. Isso é feito usando a função **rand()** que gera números aleatórios e, em seguida, calcula o resto da divisão por 100 para garantir que o número gerado seja entre 0 e 99. Então, cada vez que chamamos a função **gerarMatrizAleatoria**, ela simula diferentes distâncias entre as cidades, criando uma matriz de distâncias aleatórias.

```c
void gerarMatrizAleatoria(Matriz* matriz) {
    srand(time(NULL));

    for (int i = 0; i < matriz->numeroDeCidades; i++) {
        for (int j = 0; j < matriz->numeroDeCidades; j++) {
            if (i == j) {
                matriz->matrizDistancias[i][j] = 0;
            } else {
                int distanciaCidades = rand() % 100;
                matriz->matrizDistancias[i][j] = distanciaCidades;
            }
        }
    }

    // Imprime a matriz diretamente no terminal
    printf("Matriz de Distâncias:\n");
    for (int i = 0; i < matriz->numeroDeCidades; i++) {
        for (int j = 0; j < matriz->numeroDeCidades; j++) {
            printf("%d\t", matriz->matrizDistancias[i][j]);
        }
        printf("\n");
    }
}
```

(Figura 9: Função gerarMatrizAleatoria)

3.5. Valor X

A função **calcularCidadeInicial** junto com a função **calcularSomadeDigitos** são responsáveis por calcular o resto, que é a cidade inicial. O valor de X é calculado como entrada para a função **calcularCidadeInicial**. O algoritmo usa esse valor

como um índice para determinar qual cidade será a inicial na rota e a final também.

```
int calcularCidadeInicial(int restoSomaMatriculas) {
    int cidadeInicial = restoSomaMatriculas;
    return cidadeInicial;
}
```

(Figura 10: Função calcularCidadeInicial)

```
int calcularSomaDigitos(int matricula) {
    int soma = 0;
    int tempMatricula = matricula;

    while (tempMatricula > 0) {
        soma += tempMatricula % 10;
        tempMatricula /= 10;
    }

    return soma;
}
```

(Figura 11: Função calcularSomaDigitos)

3.6. Cálculo da distância

No processo de cálculo da distância percorrida pelo caixeiro viajante, foi utilizado um método que envolve a **matriz de distâncias entre as cidades** e o **vetor de permutação** representando a ordem das cidades a serem visitadas. Implementamos este processo por meio da função **calcularSomaDistancias**. Para cada par de cidades consecutivas no vetor de permutação, a função acessa a matriz de distâncias para encontrar a distância entre essas duas cidades. Ela itera sobre o vetor de permutação, somando as distâncias correspondentes na matriz. A variável **i** representa o índice atual no vetor de permutação e a variável **j** representa o próximo índice. Multiplicamos o **resultado final por 2**, porque o caixeiro viajante precisa voltar para a cidade de origem ao final da viagem. Este processo é repetido para **todas as permutações possíveis**, e a menor distância encontrada representa o caminho mais curto que o caixeiro viajante pode seguir para visitar todas as cidades uma vez e retornar à cidade de origem.

```c
int calcularSomaDistancias(Matriz* matriz, int* permutacao) {
    int resultado = 0;
    for (int i = 0; i < matriz->numeroDeCidades; i++) {
        int j = (i + 1) % matriz->numeroDeCidades;
        resultado += matriz->matrizDistancias[permutacao[i]][permutacao[j]];
    }
    return resultado;
}
// Função para liberar a memória da matriz
```

(Figura 12: Função calcularSomaDistancias)

4. Compilação e Execução

Para compilar e executar o nosso código, usamos o Makefile, pois ele nos permite executar todos os nossos arquivos de código mesmo não estando no mesmo diretório. Assim, criamos o Makefile no mesmo diretório onde estão seus arquivos de código-fonte.

- Para compilar o código, basta digitar **make all,** e em seguida digitar o nome que escolhemos para chamar os comandos(**./programa**).

makefile

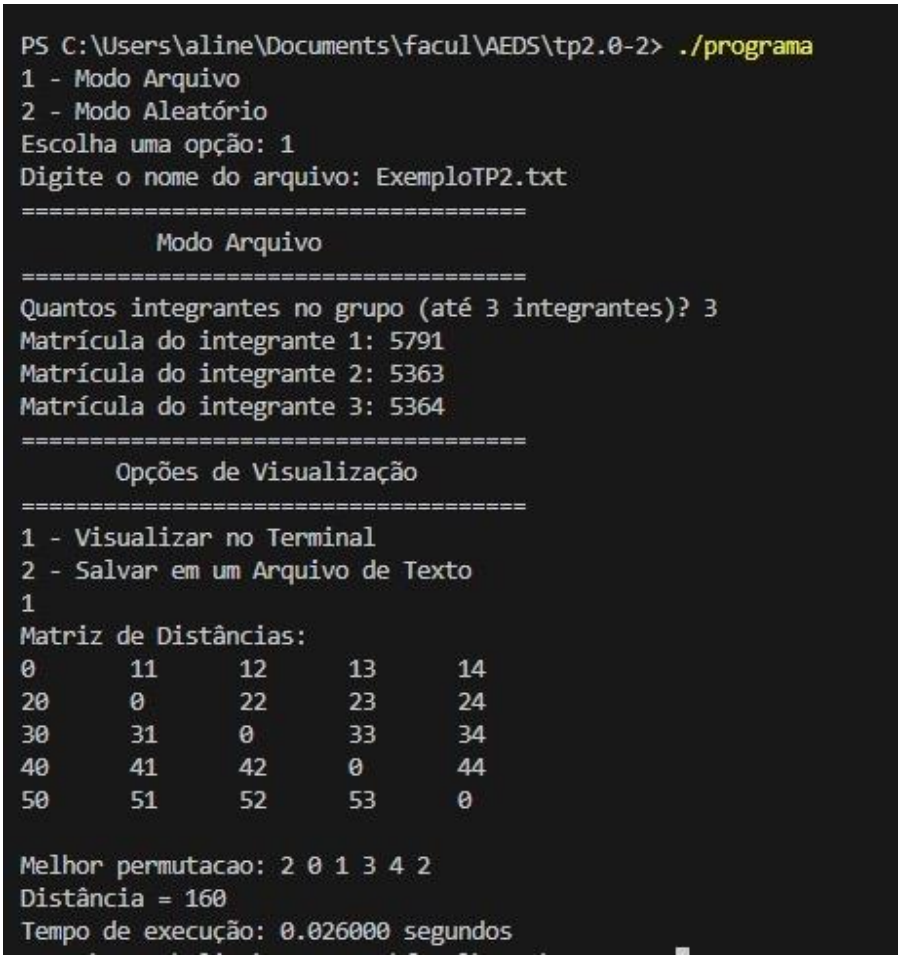
```
1  all: main.c sources/Permutacao.c sources/Menu.c sources/Matriz.c sources/ArquivoTexto.c
2      gcc -o programa main.c sources/Permutacao.c sources/Menu.c sources/Matriz.c sources/ArquivoTexto.c
3
```

(Figura 13: Arquivo makefile; Comandos)

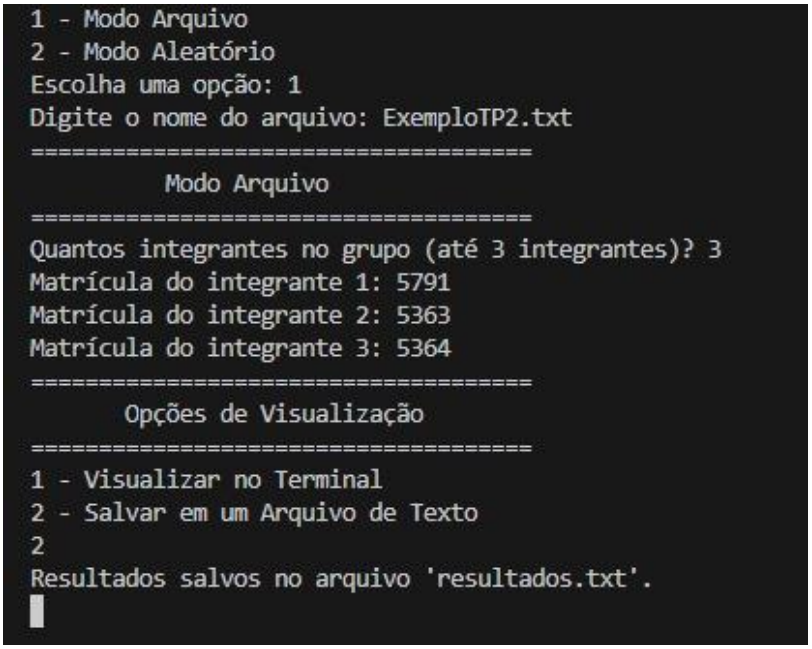
5. Resultados

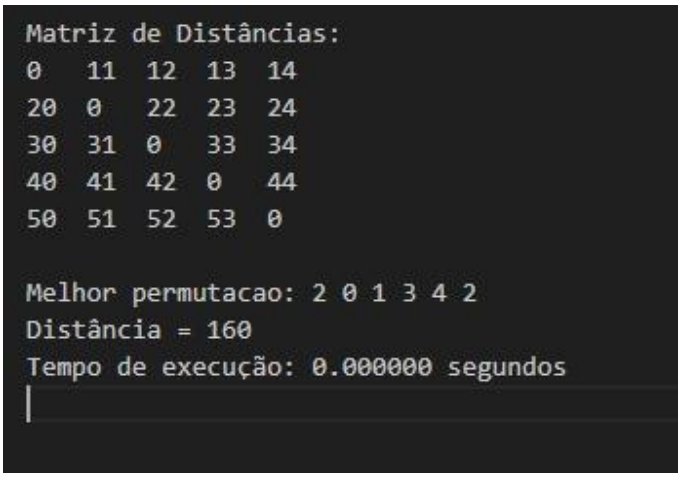
Nesta etapa, serão exibidos os resultados obtidos ao executar o programa nos modos interativo e no arquivo. Para executar as saídas, usamos o notebook Lenovo Ideapad s145, equipado com o processador intel core i5, 4GB de memória RAM , um SSD de 1TB:

5.1 Print das Saídas Modo Arquivo



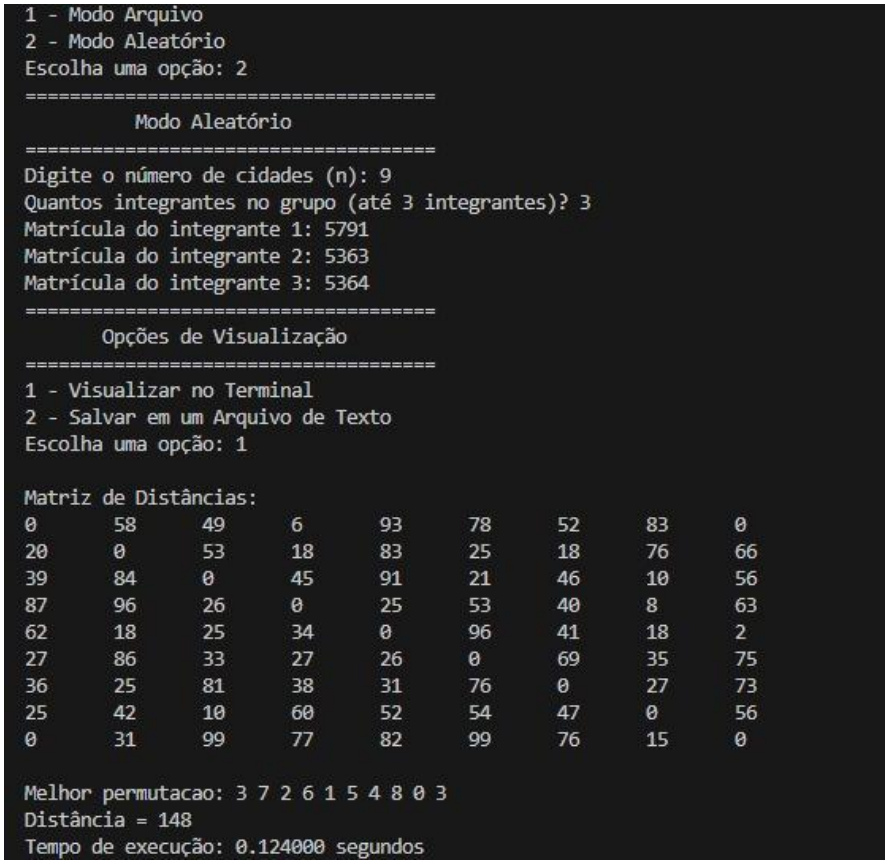
(Figura 14: Modo Arquivo, saída no terminal)



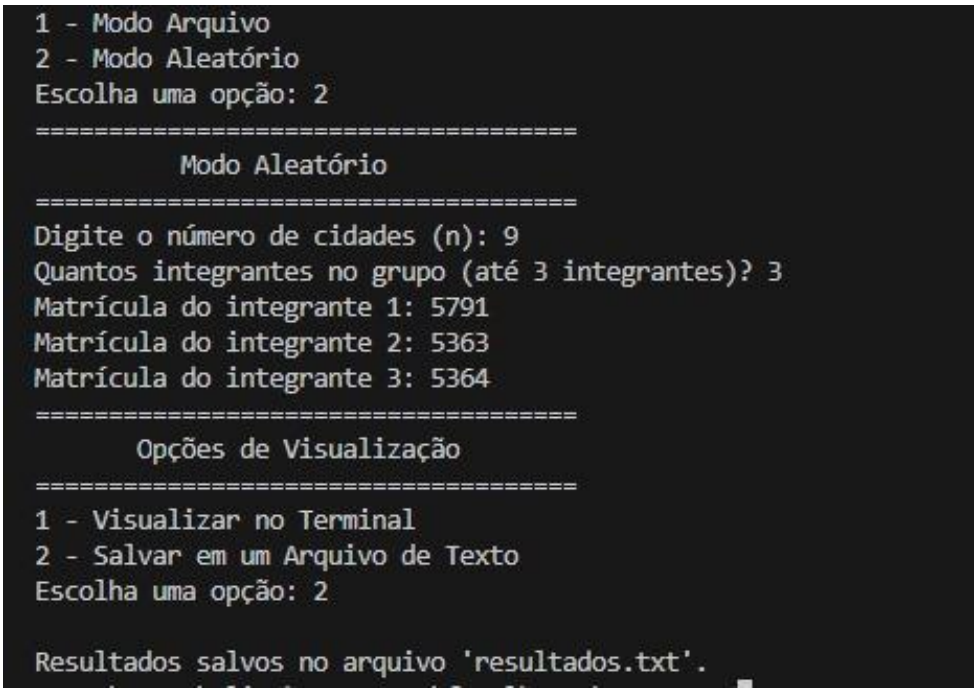


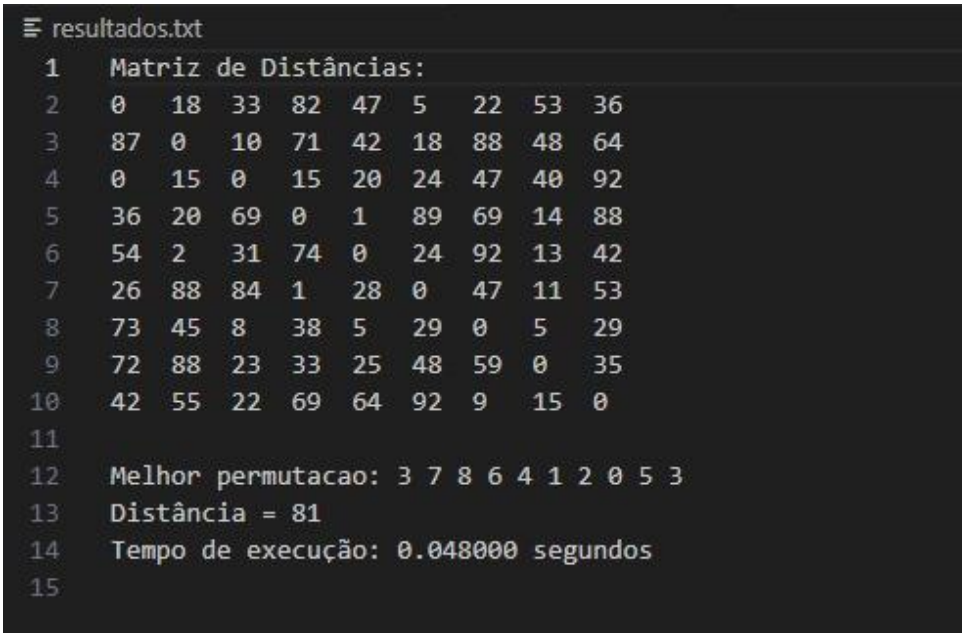
(Figura 15 e 16: Modo Arquivo, saída no arquivo.txt)

5.2 Print das saídas Modo Aleatório



(Figura 17: Modo Aleatório, saída no terminal)



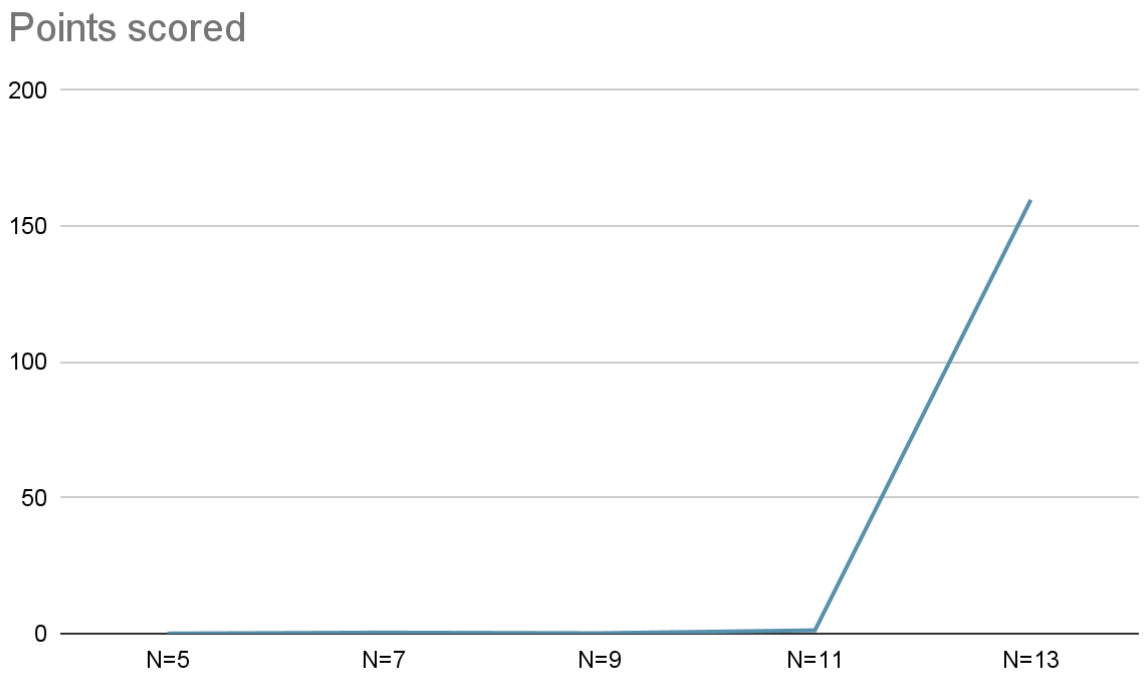


(Figura 18 e 19: Modo Aleatório, saída no terminal, direcionada para um arquivo txt)

6.Resposta da pergunta

A pergunta passada pelos monitores é analisada pelo conceito de complexidade, onde analisamos as saídas do nosso terminal, fazendo o agrupamento dos tempos de cada terminal, correspondente com suas saídas e entradas. O tempo de execução foi de extrema importância para analisar se o nosso programa atenderia a

transportadora. Abaixo, está o gráfico, pedido pelos monitores, e que vai basear a nossa resposta.



(Figura 20: Imagem do gráfico N x Tempo. Onde N=5 tem tempo igual a 0.026 segundos; N= 7 onde o tempo é igual a 0.38 segundos; N= 9 onde o tempo é igual a 0.15 segundos ; N= 11 onde o tempo é igual a 1.211 segundos ; N= 13 onde o tempo é igual a 159.679 segundos.)

Assim, o nosso programa não atenderia a demanda da empresa , pois, para atender uma transportadora que deseja encontrar a menor rota a ser percorrida entre o galpão de estoque e as N cidades (o processo inverso também), teríamos que ter um limite de 13 cidades, pois, o nosso programa começaria a rodar de um modo muito lento, prejudicando a eficiência. Isso, para um transportadora de alto nível, seria horrível, pois a alta demanda de carretos, levaria este programa ser inadequado. Agora, se tivéssemos falando de uma transportadora de nível pequeno/médio, onde ela atenderia até 13 cidades, este programa funcionaria de modo satisfatório.

7. Conclusão

O trabalho prático de resolução do Problema do Caixeiro Viajante Assimétrico foi uma jornada desafiadora e educativa, oferecendo uma oportunidade valiosa para explorar profundamente conceitos teóricos de algoritmos e aplicá-los na prática. Ao

longo deste projeto, nós enfrentamos o desafio computacional de encontrar a melhor rota possível que um caixeiro viajante poderia tomar para visitar um conjunto de cidades, minimizando o custo total do percurso.

Este trabalho não apenas ampliou nosso conhecimento sobre algoritmos e estruturas de dados, mas também nos deu uma compreensão mais profunda das complexidades envolvidas na resolução de problemas intratáveis no mundo da computação. Ao enfrentar esses desafios, fortalecemos nossas habilidades de resolução de problemas, aprendemos a importância da análise de desempenho e ganhamos uma apreciação mais profunda pela ciência da computação.

Este trabalho representa não apenas o término de um projeto acadêmico, mas também um marco significativo em nossa jornada educativa, preparando-nos para enfrentar desafios futuros com confiança e habilidade.

8. Referências

[1] DIAS, Cayo. Algoritmos de ordenação explicados com exemplos em Python, Java e C++. FreeCodeCamp, 2022. Disponível em : https://www.freecodecamp.org/portuguese/news/algoritmos-de-ordenacao-explicados-com-exemplos-em-python-java-e-c/