CIÊNCIA DA COMPUTAÇÃO

Algoritmo de resolução do problema do Caixeiro Viajante com implementação de listas encadeadas

Victor Emmanuel Susko Guimarães
Cristiano Augusto Dias Mafuz

Síntese

Neste trabalho, descreve-se a implementação de um algoritmo para resolver o Problema do Caixeiro Viajante utilizando recursividade e listas de adjacências. Será apresentado o problema a ser resolvido, a implementação da solução em C e a análise dos resultados. O algoritmo foi exposto a diversos cenários de testes e a ferramenta *valgrind* foi utilizada para ratificar possíveis vazamentos de memória do programa.

I. Introdução

Em um cenário hipotético, um caixeiro viajante passa por um determinado número de *n* cidades, e começa seu trajeto na mesma cidade em que irá terminar. O problema do caixeiro viajante consiste em determinar qual é o trajeto que ele deve percorrer para andar a menor distância possível passando por todas as cidades. A figura 1.1 ilustra o exemplo de um conjunto de 4 cidades, com as distâncias entre cada uma delas.

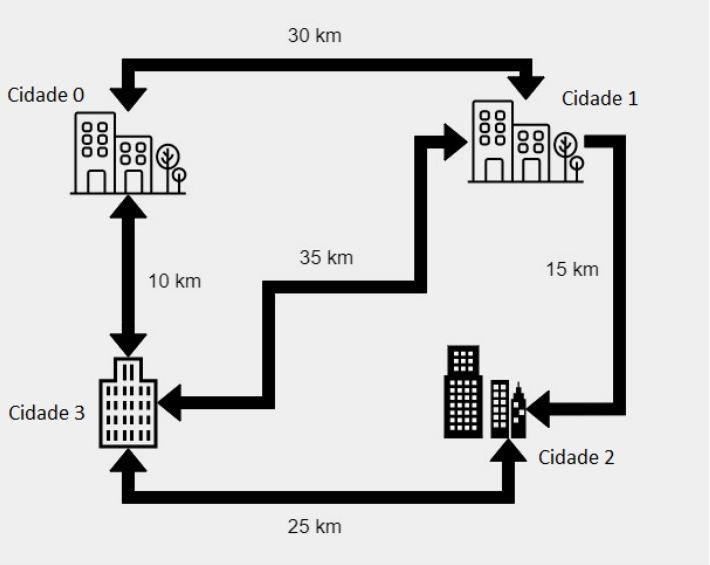

Figura 1.1 - Cidades e distâncias

Para este conjunto de cidades, existem 2 caminhos que satisfazem a menor distância possível à partir da Cidade 0, que são: Cidade 0 > Cidade 1 > Cidade 2 > Cidade 3 (ilustrado pela figura 1.2) e Cidade 0 > Cidade 3 > Cidade 2 > Cidade 1 (ilustrado pela figura 1.3)

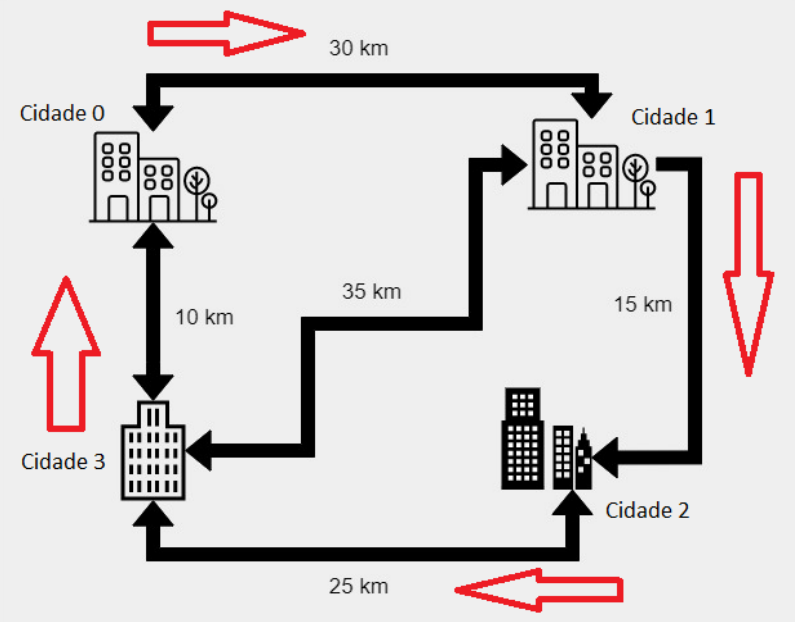

Figura 1.2 - Rota mais curta 1

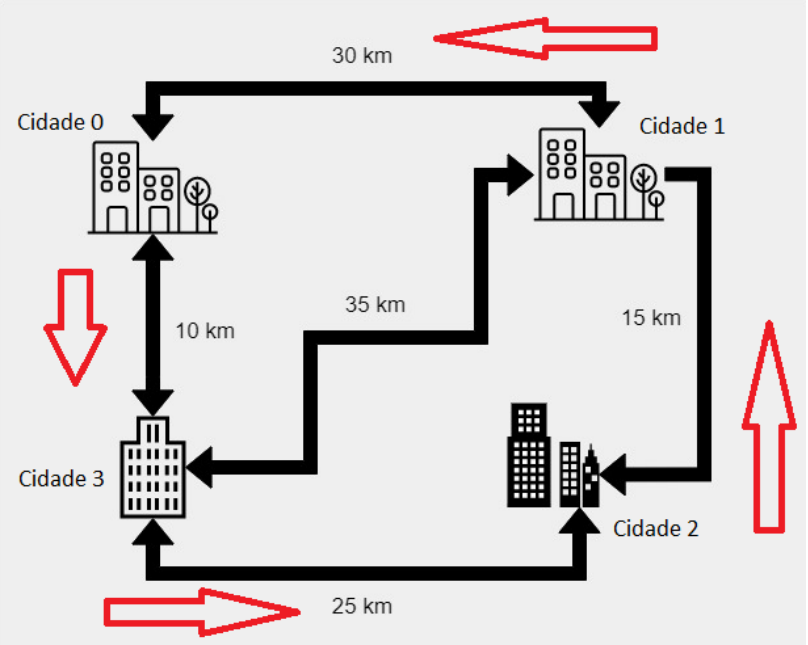

Figura 1.3 - Rota mais curta 2

Desse modo, o objetivo deste trabalho é implementar um algoritmo na linguagem de programação C que consiga, a partir da entrada do número de cidades e da distância entre elas, determinar qual o trajeto e a distância total a serem percorridos pelo caixeiro. O trabalho contará com 3 arquivos: main.c, grafo.h e grafo.c. O arquivo main.c invoca e trata as respostas das funções e procedimentos definidos no arquivo grafo.h, e o arquivo grafo.c possui todas as funções utilizadas no algoritmo.

II. Implementação

O código implementado possui, no total, 11 funções, sendo que 2 delas são as funções principais "`encontraCaminho`" e "`ordenaLista`", e as outras 9 são funções auxiliares.

O algoritmo foi implementado segundo a abordagem de força bruta, que constitui-se, essencialmente, em computar todas as possíveis rotas e a distância total percorrida em cada uma delas. Assim, a partir da análise das rotas, a solução do problema para o caminho mais curto é obtida.

Para encontrar o número de rotas segundo a função ***R(n)***, em que ***R*** é o número de rotas possíveis e ***n*** é o número de cidades, são utilizados princípios da análise combinatória. No problema em questão, para quaisquer valores de n, as posições primeira e última são fixas e não influenciam o cálculo pois o caixeiro sempre sai da cidade 0 e, por último, retorna à esta. Tomando como exemplo n = 4, tem-se como segunda possibilidade de cidade, qualquer uma das três cidades restantes (1, 2 e 3). Após essa escolha, sobram duas opções para a terceira posição. Na quarta posição, não há mais escolhas a serem feitas, pois apenas uma cidade restou. Portanto, o número de rotas possíveis é denotado por 3 x 2 x 1, que resulta em 6 rotas. Da mesma forma, para o caso de **n** cidades, observa-se que o número total de escolhas possíveis é representado pela permutação de **n-1** elementos, gerando a seguinte função: **R(n) = (n - 1)!**

Além disso, foi utilizado um vetor de listas de adjacências, sendo que cada espaço de memória (índice) do vetor corresponde às adjacências de um vértice. Dessa maneira, o vetor de listas é alocado dinamicamente com o número de cidades, e cada índice corresponde a uma lista encadeada contendo as distâncias entre o vértice em questão e outro vértice. A lista terá utilidade na função recursiva, uma vez que ela armazena as distâncias entre todas as cidades, que serão utilizadas para o cálculo do caminho que possui a menor distância.

Assim, dadas as apresentações da lógica do algoritmo, a seguir serão mostrados os protótipos das funções do código e seus respectivos papéis na biblioteca “grafo.h”:

```c
// Struct que contém os dados do grafo
typedef struct GrafoPonderado{
    int cidades;
    int **matriz;
} GrafoPonderado;

// Struct que contém os dados da lista
typedef struct ListaDeAdjacencia{
    int vertice;
    int dist;
    struct ListaDeAdjacencia* proximo;
} ListaDeAdjacencia;

GrafoPonderado *alocarGrafo(int);

ListaDeAdjacencia **alocaLista(int);

void desalocarLista(ListaDeAdjacencia*);

void  desalocarGrafo(GrafoPonderado*,  ListaDeAdjacencia**,  int*,
int );

void leGrafo(int, GrafoPonderado*);

int *criaVetor(int);

void troca(int*, int*);

void  encontraCaminho(int*,  int*,  int,  int,  ListaDeAdjacencia**,
int*);

void ordenaLista(ListaDeAdjacencia*, int, int**, int);

void imprimeOrdenado(ListaDeAdjacencia*, int);

void imprimeCaminho(int, int*);
```

1. `alocarGrafo` -> Aloca a estrutura `GrafoPonderado` e a matriz de adjacência.

2. `alocaLista` -> Aloca dinamicamente um vetor de listas de adjacências do tipo *`ListaDeAdjacencia`*.

3. `desalocarLista` -> Libera a memória alocada para o vetor de listas.

4. `desalocarGrafo` -> Desaloca a memória da estrutura do grafo, da matriz de adjacência e de outros arrays.

5. `leGrafo` -> Lê os dados do grafo.

6. `criaVetor` -> Aloca um vetor do tipo int dinamicamente.

7. `troca` -> Troca o conteúdo de duas variáveis do tipo int.

8. `encontraCaminho` -> É a função recursiva que gera todas as permutações possíveis dos vértices das cidades para encontrar todos os caminhos.

9. `ordenaLista` -> Recebe a matriz de adjacências, cria os elementos da lista de adjacências e ordena o vetor de listas de adjacências com base na menor distância entre cidades. O método de ordenação utilizado é o Selection Sort, que encontra o vértice com a menor distância e o adiciona no começo da lista.

10. `imprimeOrdenado` -> Imprime a lista de adjacências depois de ordenada.

11. `imprimeCaminho` -> Imprime o caminho mais curto encontrado.

Segue abaixo a implementação de “grafo.c” com as funções descritas:

```c
#include "grafo.h"
#include <stdio.h>
#include <stdlib.h>

    // Função que aloca dinamicamente um grafo
    GrafoPonderado *alocarGrafo(int cidades)
    {
        // Alocação a struct
        GrafoPonderado *grafo = malloc(sizeof(GrafoPonderado));
```

```
        // Definção do número de cidades
        grafo->cidades = cidades;

        // Alocação a matriz de adjacência
        grafo->matriz = malloc(cidades * sizeof(int *));

        for (int i = 0; i < cidades; i++)
        {
            //Alocação de cada elemento da matriz
            grafo->matriz[i] = malloc(cidades * sizeof(int));
        }

        return grafo;
    }

    //Função que aloca dinamicamente um vetor de listas de adjacências
    ListaDeAdjacencia** alocaLista(int n)
    {
        // Alocação do vetor n listas
          ListaDeAdjacencia** lista = (ListaDeAdjacencia**) malloc(n *
sizeof(ListaDeAdjacencia*));

        // Verifica se a alocação foi bem-sucedida
        if (lista != NULL)
        {
            // Inicializa cada elemento da lista
            for (int i = 0; i < n; i++)
            {
                                        lista[i]  =  (ListaDeAdjacencia*)
malloc(sizeof(ListaDeAdjacencia));
            }
        }

        // Retorna o ponteiro para o array alocado
        return lista;
    }


    //Função que libera a memória de uma lista de adjacências. No caso
deste trabalho, a função irá liberar um espaço do vetor de listas
    void desalocarLista(ListaDeAdjacencia *lista)
    {
```

```
        //auxiliares que irão percorrer a lista
        ListaDeAdjacencia *pAux = lista;
        ListaDeAdjacencia *pAux2;

            //enquanto não chegar em NULL(final da lista) a memória
continuará sendo liberada
        while(pAux != NULL)
        {
            pAux2 = pAux -> proximo;
            free(pAux);
            pAux = pAux2;
        }
    }


    // Função que desaloca um grafo
        void  desalocarGrafo(GrafoPonderado  *grafo,  ListaDeAdjacencia
**adjacencias, int *caminhos, int cidades)
    {
          //Liberação da memória de cada um dos índices do vetor de
listas
        for (int i = 0; i < cidades; i++)
            desalocarLista(adjacencias[i]);

        //Desalocação do vetor de listas
        free(adjacencias);

        //Desalocação do vetor de possíveis caminhos
        free(caminhos);
        caminhos = NULL;

        // Desalocação das linhas matriz de adjacência do grafo
        for (int i = 0; i < cidades; i++)
        {
            free(grafo->matriz[i]);
            grafo->matriz[i] = NULL;
        }

        //Desalocação final da matriz
        free(grafo->matriz);
        grafo->matriz = NULL;

        // Desaloca o grafo
```

```
        free(grafo);
        grafo = NULL;
    }


    //Função que lê os dados do grafo
    void leGrafo(int cidades, GrafoPonderado *grafo)
    {
        //auxiliares que apenas irão receber os dois primeiros números
        int k, l;
        for (int i = 0; i < cidades; i++)
        {
            for (int j = 0; j < cidades; j++)
            {
                scanf(" %d %d", &k, &l);
                //preenchimento da matriz do grafo
                scanf(" %d", &grafo->matriz[k][l]);
            }
        }
    }

    /* Função que aloca um vetor e o preenche com o número de todas as
cidades. Essa função será utilizda para armazenar caminhos, sendo que
      Cada caminho parte da cidade 0 e ao final também volta para a
cidade 0. Por isso, para que o último elemento seja 0,
    é necessário que esse vetor tenha o tamanho do número de cidades +
1.
    */

    int *criaVetor(int cidades)
    {
        //alocação dinâmica do vetor
        int *vet = malloc((cidades+1) * sizeof(int));

        for (int i = 0; i < cidades; i++)
        {
            //preenchimento do vetor com o índice i
            vet[i] = i;
        }
        //preenchimento da última posição com 0
        vet[cidades] = 0;
        return vet;
    }
```

```c
    // Função de troca simples
    void troca(int *x, int *y)
    {
        int temp = *x;
        *x = *y;
        *y = temp;
    }

    /* Função recursiva que encontra o caminho com a menor distância. A função recebe um vetor que irá computar todos os caminhos possíveis,
      um vetor que irá armazenar o melhor caminho encontrado, duas variáveis do tipo int que irão servir de parâmetro para inicio/fim da recursão,
     a lista de adjacências, que possui todas as distâncias entre as cidades, e uma variável do tipo int que irá registrar qual foi a distância
    mais curta com base no melhor caminho.*/

       void encontraCaminho(int *caminhos, int* melhor_caminho, int inicio, int fim, ListaDeAdjacencia **adjacencias, int *melhor_distancia)
    {
        //variável que irá somar as distâncias entre cidades
        int distancia_total;

        //variável auxiliar que irá percorrer o vetor de adjacências
        ListaDeAdjacencia *temp;

        /*
            quando inicio = fim, a função entra no caso base, o que significa que a recursão chegou ao fim e
             uma possibilidade de caminho foi computada no vetor de caminhos.
        */
        if (inicio == fim)
        {
               //a variável que somará as distâcias é inicializada com zero
            distancia_total = 0;

            //loop que irá as distâncias do caminho
            for(int i = 0; i < fim; i++)
```

```
            {
                /*
                 a variavel temporária precisa receber o indice do vetor de listas com base no numero do vetor de caminhos. Por exemplo,
                  se o vetor de caminhos computou o caminho [0,1,2,3,0], na primeira iteração, a variável temporária precisa
                  começar recebendo o indice de listas 0, que é de onde parte o caixeiro, e percorrer a lista até encontrar o segundo vértice,
                  que é a cidade destino, no caso, cidade 1. Desse modo, a cada iteração, o programa tem a cidade de onde o caixeiro parte
                  e a cidade destino que é o próximo elemento do vetor. Por fim, se a distância entre as duas cidades for diferente de zero,
                 então a distância precisa ser somada. Caso a distâncias entre duas cidades seja igual a zero, significa que não há caminho
                      entre essas duas cidades, e portanto, o caminho computado não pode ser uma solução e deve ser descartado.
                */
                temp = adjacencias[caminhos[i]];

                     //percorrendo a lista até encontrar o elemento do caminho
                while(temp->vertice != caminhos[i+1])
                {
                    temp = temp->proximo;
                }

                      //verificação que valida se a distância entre as cidades é zero ou não
                if(temp->dist == 0)
                    return;
                else
                    distancia_total += temp->dist;
            }

              //Se ao final do processo a distância encontrada for menor que a anterior, significa que o caminho é melhor
            if(distancia_total < *melhor_distancia)
            {
                //atribuição da melhor distância
                *melhor_distancia = distancia_total;
                //atribuição do caminho mais curto encontrado
                for(int i = 0; i < fim; i++)
                    melhor_caminho[i] = caminhos[i];
```

```
            }
        }

        /*
         Parte da função onde ocorre a recursividade. O loop abaixo irá
realizar trocas sucessivas do vetor de caminhos para que as
          possibilidades sejam computadas. A primeira chamada da função
troca irá embaralhar o vetor toda vez que a variável i for diferente
         da variável início, e quando as variáveis forem iguais a função
de troca irá trocar números iguais e nada acontece. Já a segunda
chamada
          da função troca irá destrocar os números para que else sejam
reembaralhados posteriormente.
        */
        else
        {
            for (int i = inicio; i < fim; i++)
            {
                troca(&caminhos[inicio], &caminhos[i]);
                   encontraCaminho(caminhos,melhor_caminho, inicio + 1,
fim, adjacencias, melhor_distancia);
                troca(&caminhos[inicio], &caminhos[i]);
            }
        }
    }

     /*Função que ordena o vetor de listas de adjacencias com base na
menor distância entre cidades. A função não recebe o vetor de listas,
    mas sim um elemento desse vetor, que é uma lista. Ela também recebe
o número de cidades, a matriz do grafo e o vértice que
     está sendo ordenado. Assim, a função irá criar espaços novos para
cada uma das listas e adicionar as adjacências dos respectivos vértice
     de maneira já ordenada. Vale ressaltar que o método de ordenação
aqui utilizado foi o Selection Sort, que para cada iteração,
    percorre toda a lista, seleciona o menor valor encontrado e insere
no começo da lista.*/
     void ordenaLista(ListaDeAdjacencia *adjacencias, int cidades, int
**matriz, int vertice)
    {
            //Loop que irá gerar cada um dos parêntesis da lista de
adjacencias do vértice atual.
        for (int i = 0; i < cidades; i++)
        {
```

```
            /*
              as variáveis abaixo irão ser utilizadas na comparação das
distâncias entre os vértices(cidades) e são inicializadas com valores
                arbitrários. A variável aux_dist é a variável que irá
receber a menor distância e a variável aux_pos recebe em qual vértice
                 essa distância se encontra, ou seja, essas variáveis
receberão a cidade mais próxima da atual.
            */
            int aux_dist = 100, aux_pos = -1;

             //a lista em análise precisa ser percorrida todas as vezes
para encontrar a menor distância, por isso a necessidade de outro loop
            for (int j = 0; j < cidades; j++)
            {
                      //variável temporária que irá percorrer a lista
encadeada
                ListaDeAdjacencia *temp = adjacencias;

                /*
                  variável que irá verificar se a distância encontrada é
zero, que significa que o vértice em análise é a própria cidade.
                    Esse verificador cumpre a função de impedir que a
distância entre uma cidade e ela mesma seja computada outras vezes,
                   pois 0 seria a menor distância possível entre duas
cidades.
                */
                int verificador_0 = 0;

                if(i != 0)
                {
                    while (temp != NULL)
                    {
                        if (temp->vertice == j)
                        {
                              //o verificador passa a ter valor 1 quando
já computou a distância 0
                            verificador_0 = 1;
                            break;
                        }
                        temp = temp->proximo;
                    }
                }
```

```
                //comparação entre a distância da matriz de adjacências
e a auxiliar de distância
                if (matriz[vertice][j] < aux_dist && !verificador_0)
                {
                        //atribuição da menor distância e a posição do
vértice às auxiliares
                    aux_dist = matriz[vertice][j];
                    aux_pos = j;
                }
            }

                // Adição do novo nó à lista. caso seja a primeira
iteração, a lista está na cabeça e já foi alocada
            if (i == 0)
            {
                    //alocação do novo nó e atribuição dos vértices e
distâncias
                adjacencias->dist = aux_dist;
                adjacencias->vertice = aux_pos;
                adjacencias->proximo = NULL;
            }
              // Caso não seja a primeira iteração, a lista precisa ser
percorrida antes de ser adicionado um novo nó.
            else
            {
                ListaDeAdjacencia *temp2 = adjacencias;
                while (temp2->proximo != NULL)
                {
                    temp2 = temp2->proximo;
                }
                    //alocação do novo nó e atribuição dos vértices e
distâncias
                                                    temp2->proximo    =
(ListaDeAdjacencia*)malloc(sizeof(ListaDeAdjacencia));;
                temp2->proximo->dist = aux_dist;
                temp2->proximo->vertice = aux_pos;
                temp2->proximo->proximo = NULL;
            }
        }
    }

    // Função que imprime a lista de adjacências depois de ordenada
    void imprimeOrdenado(ListaDeAdjacencia *adjacencias, int indice)
```

```c
{
    printf("Adjacencias do vertice %d:", indice);
    //variável temporária que irá percorrer a lista
    ListaDeAdjacencia *temp = adjacencias;

    while(temp != NULL)
    {
        //impressão do vértice e da distância
        printf(" (%d, %d) ->", temp->vertice, temp->dist);
        temp = temp->proximo;
    }
    printf(" NULL\n");
}

// Função que imprime o caminho com a melhor distância encontrada
void imprimeCaminho(int cidades, int *melhor_caminho)
{
    printf("Melhor caminho: ");
    //impressão dos elementos do vetor caminho
    for (int i = 0; i <= cidades; i++)
    {
        printf("%d ", melhor_caminho[i]);
    }
    printf("\n");
}
```

III. Testes

Foram realizados cinco testes, que foram incluídos na pasta "*tests*". Para cada teste, a primeira linha da entrada especifica o número de cidades a serem visitadas. Depois, são fornecidas, respectivamente e em cada linha, a cidade, suas cidades vizinhas e a distância entre elas. A saída esperada para cada teste é a lista de adjacências de cada vértice ordenada de forma crescente em relação às distâncias, seguido do melhor caminho e da melhor distância.

Desse modo, seguem abaixo todas as entradas dos 5 testes realizados com as saídas esperadas.

Teste 1:
4
0 0 0
0 1 10
0 2 15
0 3 20
1 0 10
1 1 0
1 2 35
1 3 25
2 0 15
2 1 35
2 2 0
2 3 30
3 0 20
3 1 25
3 2 30
3 3 0

Saída esperada:
Adjacencias do vertice 0: (0, 0) -> (1, 10) -> (2, 15) -> (3, 20) -> NULL
Adjacencias do vertice 1: (1, 0) -> (0, 10) -> (3, 25) -> (2, 35) -> NULL
Adjacencias do vertice 2: (2, 0) -> (0, 15) -> (3, 30) -> (1, 35) -> NULL
Adjacencias do vertice 3: (3, 0) -> (0, 20) -> (1, 25) -> (2, 30) -> NULL
Melhor caminho: 0 1 3 2 0
Melhor distancia: 80

Teste 2:

```
6
0 0 0
0 1 1
0 2 2
0 3 1
0 4 1
0 5 2
1 0 1
1 1 0
1 2 7
1 3 1
1 4 4
1 5 3
2 0 2
2 1 7
2 2 0
2 3 3
2 4 1
2 5 1
3 0 1
3 1 1
3 2 3
3 3 0
3 4 8
3 5 1
4 0 1
4 1 2
4 2 1
4 3 8
4 4 0
4 5 1
5 0 2
5 1 3
5 2 1
5 3 1
5 4 1
5 5 0
```

Saída esperada:
Adjacencias do vertice 0: (0, 0) -> (1, 1) -> (3, 1) -> (4, 1) -> (2, 2) -> (5, 2) -> NULL
Adjacencias do vertice 1: (1, 0) -> (0, 1) -> (3, 1) -> (5, 3) -> (4, 4) -> (2, 7) -> NULL
Adjacencias do vertice 2: (2, 0) -> (4, 1) -> (5, 1) -> (0, 2) -> (3, 3) -> (1, 7) -> NULL
Adjacencias do vertice 3: (3, 0) -> (0, 1) -> (1, 1) -> (5, 1) -> (2, 3) -> (4, 8) -> NULL
Adjacencias do vertice 4: (4, 0) -> (0, 1) -> (2, 1) -> (5, 1) -> (1, 2) -> (3, 8) -> NULL
Adjacencias do vertice 5: (5, 0) -> (2, 1) -> (3, 1) -> (4, 1) -> (0, 2) -> (1, 3) -> NULL
Melhor caminho: 0 1 3 5 2 4 0
Melhor distancia: 6

Teste 3:
4
0 0 0
0 1 1
0 2 1
0 3 3
1 0 1
1 1 0
1 2 4
1 3 5
2 0 1
2 1 4
2 2 0
2 3 6
3 0 3
3 1 5
3 2 6
3 3 0

Saída esperada:
Adjacencias do vertice 0: (0, 0) -> (1, 1) -> (2, 1) -> (3, 3) -> NULL
Adjacencias do vertice 1: (1, 0) -> (0, 1) -> (2, 4) -> (3, 5) -> NULL
Adjacencias do vertice 2: (2, 0) -> (0, 1) -> (1, 4) -> (3, 6) -> NULL
Adjacencias do vertice 3: (3, 0) -> (0, 3) -> (1, 5) -> (2, 6) -> NULL
Melhor caminho: 0 1 3 2 0
Melhor distancia: 13

Teste 4:
5
0 0 0
0 1 2
0 2 0
0 3 3
0 4 6
1 0 2
1 1 0
1 2 4
1 3 3
1 4 0
2 0 0
2 1 4
2 2 0
2 3 7
2 4 3
3 0 3
3 1 3
3 2 7
3 3 0
3 4 3
4 0 6
4 1 0
4 2 3
4 3 3
4 4 0

Saída esperada:
Adjacencias do vertice 0: (0, 0) -> (2, 0) -> (1, 2) -> (3, 3) -> (4, 6) -> NULL
Adjacencias do vertice 1: (1, 0) -> (4, 0) -> (0, 2) -> (3, 3) -> (2, 4) -> NULL
Adjacencias do vertice 2: (0, 0) -> (2, 0) -> (4, 3) -> (1, 4) -> (3, 7) -> NULL
Adjacencias do vertice 3: (3, 0) -> (0, 3) -> (1, 3) -> (4, 3) -> (2, 7) -> NULL
Adjacencias do vertice 4: (1, 0) -> (4, 0) -> (2, 3) -> (3, 3) -> (0, 6) -> NULL
Melhor caminho: 0 1 2 4 3 0
Melhor distancia: 15

Teste 5:
5
0 0 0
0 1 5
0 2 10
0 3 0
0 4 1
1 0 5
1 1 0
1 2 0
1 3 10
1 4 1
2 0 10
2 1 0
2 2 0
2 3 2
2 4 1
3 0 0
3 1 10
3 2 2
3 3 0
3 4 1
4 0 1
4 1 1
4 2 1
4 3 1
4 4 0

Saída esperada:
Adjacencias do vertice 0: (0, 0) -> (3, 0) -> (4, 1) -> (1, 5) -> (2, 10) -> NULL
Adjacencias do vertice 1: (1, 0) -> (2, 0) -> (4, 1) -> (0, 5) -> (3, 10) -> NULL
Adjacencias do vertice 2: (1, 0) -> (2, 0) -> (4, 1) -> (3, 2) -> (0, 10) -> NULL
Adjacencias do vertice 3: (0, 0) -> (3, 0) -> (4, 1) -> (2, 2) -> (1, 10) -> NULL
Adjacencias do vertice 4: (4, 0) -> (0, 1) -> (1, 1) -> (2, 1) -> (3, 1) -> NULL
Melhor caminho: 0 1 3 2 4 0
Melhor distancia: 19

IV. Análise

Para a análise do programa, a ferramenta *valgrind* foi utilizada para checar possíveis vazamentos de memória. As figuras de 2.1 a 2.5 mostram os resultados das análises da ferramenta e o texto gerado indica que não houve vazamentos de memória com as alocações, e a figura 6 mostra o resultado dos testes segundo o corretor de testes.

Saída do programa, teste 1:

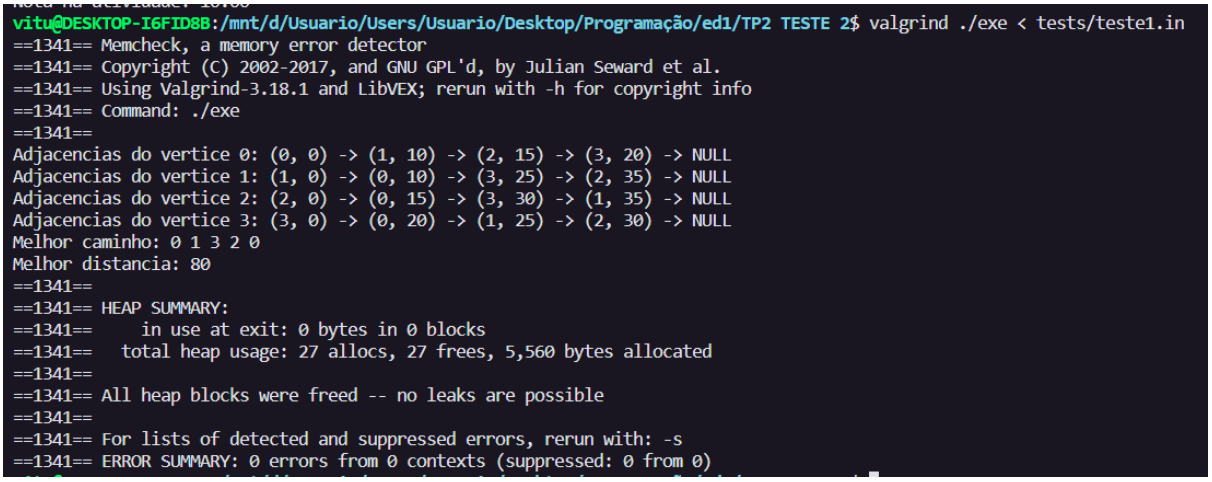

Figura 2.1 - Saída do programa - teste 1

Saída do programa, teste 2:

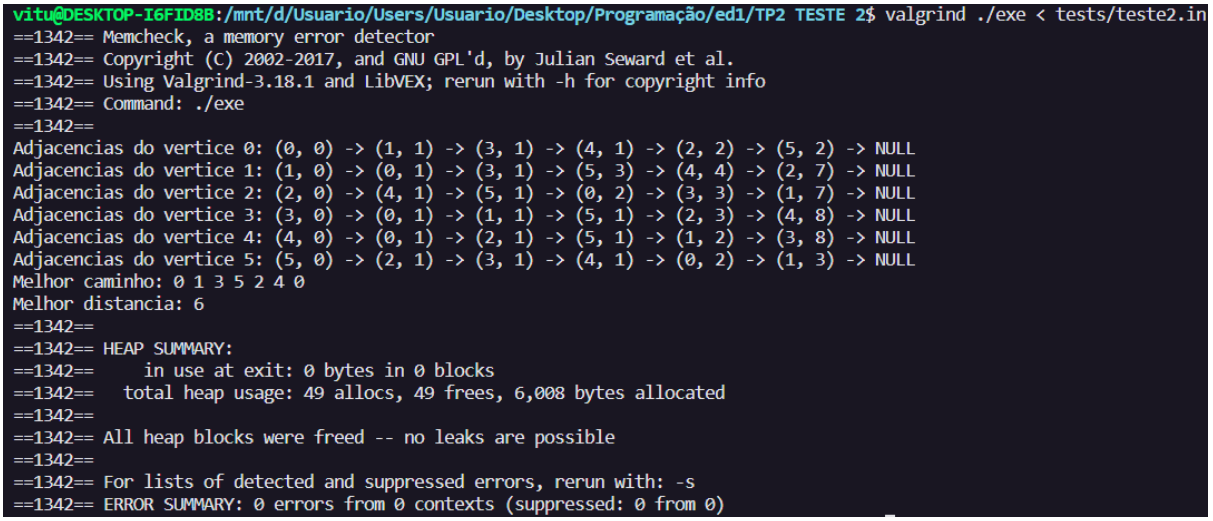

Figura 2.2 - Saída do programa - teste 2

Saída do programa, teste 3:

```
vitu@DESKTOP-I6FID8B:/mnt/d/Usuario/Users/Usuario/Desktop/Programação/ed1/TP2 TESTE 2$ valgrind ./exe < tests/teste3.in
==1343== Memcheck, a memory error detector
==1343== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1343== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==1343== Command: ./exe
==1343==
Adjacencias do vertice 0: (0, 0) -> (1, 1) -> (2, 1) -> (3, 3) -> NULL
Adjacencias do vertice 1: (1, 0) -> (0, 1) -> (2, 4) -> (3, 5) -> NULL
Adjacencias do vertice 2: (2, 0) -> (0, 1) -> (1, 4) -> (3, 6) -> NULL
Adjacencias do vertice 3: (3, 0) -> (0, 3) -> (1, 5) -> (2, 6) -> NULL
Melhor caminho: 0 1 3 2 0
Melhor distancia: 13
==1343==
==1343== HEAP SUMMARY:
==1343==     in use at exit: 0 bytes in 0 blocks
==1343==   total heap usage: 27 allocs, 27 frees, 5,560 bytes allocated
==1343==
==1343== All heap blocks were freed -- no leaks are possible
==1343==
==1343== For lists of detected and suppressed errors, rerun with: -s
==1343== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 2.3 - Saída do programa - teste 3

Saída do programa, teste 4:

```
vitu@DESKTOP-I6FID8B:/mnt/d/Usuario/Users/Usuario/Desktop/Programação/ed1/TP2 TESTE 2$ valgrind ./exe < tests/teste4.in
==1344== Memcheck, a memory error detector
==1344== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1344== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==1344== Command: ./exe
==1344==
Adjacencias do vertice 0: (0, 0) -> (2, 0) -> (1, 2) -> (3, 3) -> (4, 6) -> NULL
Adjacencias do vertice 1: (1, 0) -> (4, 0) -> (0, 2) -> (3, 3) -> (2, 4) -> NULL
Adjacencias do vertice 2: (0, 0) -> (2, 0) -> (4, 3) -> (1, 4) -> (3, 7) -> NULL
Adjacencias do vertice 3: (3, 0) -> (0, 3) -> (1, 3) -> (4, 3) -> (2, 7) -> NULL
Adjacencias do vertice 4: (1, 0) -> (4, 0) -> (2, 3) -> (3, 3) -> (0, 6) -> NULL
Melhor caminho: 0 1 2 4 3 0
Melhor distancia: 15
==1344==
==1344== HEAP SUMMARY:
==1344==     in use at exit: 0 bytes in 0 blocks
==1344==   total heap usage: 37 allocs, 37 frees, 5,764 bytes allocated
==1344==
==1344== All heap blocks were freed -- no leaks are possible
==1344==
==1344== For lists of detected and suppressed errors, rerun with: -s
==1344== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 2.4 - Saída do programa - teste 4

Saída do programa, teste 5:

```
vitu@DESKTOP-I6FID8B:/mnt/d/Usuario/Users/Usuario/Desktop/Programação/ed1/TP2 TESTE 2$ valgrind ./exe < tests/teste5.in
==1345== Memcheck, a memory error detector
==1345== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1345== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==1345== Command: ./exe
==1345==
Adjacencias do vertice 0: (0, 0) -> (3, 0) -> (4, 1) -> (1, 5) -> (2, 10) -> NULL
Adjacencias do vertice 1: (1, 0) -> (2, 0) -> (4, 1) -> (0, 5) -> (3, 10) -> NULL
Adjacencias do vertice 2: (1, 0) -> (2, 0) -> (4, 1) -> (3, 2) -> (0, 10) -> NULL
Adjacencias do vertice 3: (0, 0) -> (3, 0) -> (4, 1) -> (2, 2) -> (1, 10) -> NULL
Adjacencias do vertice 4: (4, 0) -> (0, 1) -> (1, 1) -> (2, 1) -> (3, 1) -> NULL
Melhor caminho: 0 1 3 2 4 0
Melhor distancia: 19
==1345==
==1345== HEAP SUMMARY:
==1345==     in use at exit: 0 bytes in 0 blocks
==1345==   total heap usage: 37 allocs, 37 frees, 5,764 bytes allocated
==1345==
==1345== All heap blocks were freed -- no leaks are possible
==1345==
==1345== For lists of detected and suppressed errors, rerun with: -s
==1345== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 2.5 - Saída do programa - teste 5

Saída do programa, corretor:

```
vitu@DESKTOP-I6FID8B:/mnt/d/Usuario/Users/Usuario/Desktop/Programação/ed1/TP2 TESTE 2$ python3 corretor.py
Analisando atividade:
    teste1.in  OK
    teste2.in  OK
    teste3.in  OK
    teste4.in  OK
    teste5.in  OK
Nota na atividade: 10.00
```

Figura 3 - Saída do programa - corretor

Além disso, no arquivo main.c, a biblioteca “sys/time.h” foi inserida para conferir o tempo de execução do algoritmo para cada caso de teste. Como o cálculo final do tempo de execução é dado na ordem de microssegundos (10^{-6} segundos), o valor final foi dividido por 1000 para que a ordem fosse convertida para milissegundos. A figura 4 mostra os testes com o cálculo do tempo.

```
vitu@DESKTOP-I6FID8B:/mnt/d/Usuario/Users/Usuario/Desktop/Programação/ed1/TP2$ ./exe <tests/teste1.in
Adjacencias do vertice 0: (0, 0) -> (1, 10) -> (2, 15) -> (3, 20) -> NULL
Adjacencias do vertice 1: (1, 0) -> (0, 10) -> (3, 25) -> (2, 35) -> NULL
Adjacencias do vertice 2: (2, 0) -> (0, 15) -> (3, 30) -> (1, 35) -> NULL
Adjacencias do vertice 3: (3, 0) -> (0, 20) -> (1, 25) -> (2, 30) -> NULL
Melhor caminho: 0 1 3 2 0
Melhor distancia: 80
Tempo gasto: 1.142 milissegundos
vitu@DESKTOP-I6FID8B:/mnt/d/Usuario/Users/Usuario/Desktop/Programação/ed1/TP2$ ./exe <tests/teste2.in
Adjacencias do vertice 0: (0, 0) -> (1, 1) -> (3, 1) -> (4, 1) -> (2, 2) -> (5, 2) -> NULL
Adjacencias do vertice 1: (1, 0) -> (0, 1) -> (3, 1) -> (5, 3) -> (4, 4) -> (2, 7) -> NULL
Adjacencias do vertice 2: (2, 0) -> (4, 1) -> (5, 1) -> (0, 2) -> (3, 3) -> (1, 7) -> NULL
Adjacencias do vertice 3: (3, 0) -> (0, 1) -> (1, 1) -> (5, 1) -> (2, 3) -> (4, 8) -> NULL
Adjacencias do vertice 4: (4, 0) -> (0, 1) -> (2, 1) -> (5, 1) -> (1, 2) -> (3, 8) -> NULL
Adjacencias do vertice 5: (5, 0) -> (2, 1) -> (3, 1) -> (4, 1) -> (0, 2) -> (1, 3) -> NULL
Melhor caminho: 0 1 3 5 2 4 0
Melhor distancia: 6
Tempo gasto: 1.047 milissegundos
vitu@DESKTOP-I6FID8B:/mnt/d/Usuario/Users/Usuario/Desktop/Programação/ed1/TP2$ ./exe <tests/teste3.in
Adjacencias do vertice 0: (0, 0) -> (1, 1) -> (2, 1) -> (3, 3) -> NULL
Adjacencias do vertice 1: (1, 0) -> (0, 1) -> (2, 4) -> (3, 5) -> NULL
Adjacencias do vertice 2: (2, 0) -> (0, 1) -> (1, 4) -> (3, 6) -> NULL
Adjacencias do vertice 3: (3, 0) -> (0, 3) -> (1, 5) -> (2, 6) -> NULL
Melhor caminho: 0 1 3 2 0
Melhor distancia: 13
Tempo gasto: 1.000 milissegundos
vitu@DESKTOP-I6FID8B:/mnt/d/Usuario/Users/Usuario/Desktop/Programação/ed1/TP2$ ./exe <tests/teste4.in
Adjacencias do vertice 0: (0, 0) -> (2, 0) -> (1, 2) -> (3, 3) -> (4, 6) -> NULL
Adjacencias do vertice 1: (1, 0) -> (4, 0) -> (0, 2) -> (3, 3) -> (2, 4) -> NULL
Adjacencias do vertice 2: (0, 0) -> (2, 0) -> (4, 3) -> (1, 4) -> (3, 7) -> NULL
Adjacencias do vertice 3: (3, 0) -> (0, 3) -> (1, 3) -> (4, 3) -> (2, 7) -> NULL
Adjacencias do vertice 4: (1, 0) -> (4, 0) -> (2, 3) -> (3, 3) -> (0, 6) -> NULL
Melhor caminho: 0 1 2 4 3 0
Melhor distancia: 15
Tempo gasto: 0.985 milissegundos
vitu@DESKTOP-I6FID8B:/mnt/d/Usuario/Users/Usuario/Desktop/Programação/ed1/TP2$ ./exe <tests/teste5.in
Adjacencias do vertice 0: (0, 0) -> (3, 0) -> (4, 1) -> (1, 5) -> (2, 10) -> NULL
Adjacencias do vertice 1: (1, 0) -> (2, 0) -> (4, 1) -> (0, 5) -> (3, 10) -> NULL
Adjacencias do vertice 2: (1, 0) -> (2, 0) -> (4, 1) -> (3, 2) -> (0, 10) -> NULL
Adjacencias do vertice 3: (0, 0) -> (3, 0) -> (4, 1) -> (2, 2) -> (1, 10) -> NULL
Adjacencias do vertice 4: (4, 0) -> (0, 1) -> (1, 1) -> (2, 1) -> (3, 1) -> NULL
Melhor caminho: 0 1 3 2 4 0
Melhor distancia: 19
Tempo gasto: 0.805 milissegundos
vitu@DESKTOP-I6FID8B:/mnt/d/Usuario/Users/Usuario/Desktop/Programação/ed1/TP2$
```

Figura 4 - Saída do programa - tempo de execução do algoritmo

Observa-se que todas as saídas obedeceram ao esperado em cada teste.

V. Conclusão

O problema do Caixeiro Viajante consiste na procura do trajeto mais curto em um dado número de cidades, em que são conhecidas o número de cidades e as distâncias entre cada uma delas. Além disso, este é um caso de otimização abordado pela matemática e que, atualmente, não existe uma fórmula matemática capaz de prever a menor rota e o menor trajeto do viajante.

Dessa maneira, o objetivo deste trabalho é criar um algoritmo que faça os cálculos das distâncias e encontre a melhor rota a partir de todas as rotas descobertas.

O algoritmo construído demonstrou êxito nos cinco testes ao qual ele foi submetido, apresentando a saída esperada em todos, sem vazamentos de memórias e com tempo de execução menor do que 1 segundo.

Conclui-se, portanto, que as maiores dificuldades encontradas no trabalho foram a criação da função recursiva que computasse todas as diferentes rotas pelas quais o viajante poderia passar e integrar as distâncias entre as cidades em cada uma delas e a implementação da ordenação de um vetor de listas de adjacências que contêm todas as distâncias de cada uma das cidades tomadas duas a duas.

VI. Bibliografia

- O Problema de 1 MILHÃO de DÓLARES: https://youtu.be/9WwYO1Jtr7Y

- Pesquisa Operacional II - Aula 08 - O problema do Caixeiro Viajante: https://youtu.be/yI9bRgXbE1c

- Playlist Introdução à Teoria dos Grafos: https://youtube.com/playlist?list=PLrVGp617x0hAm90-7zQzbRsSOnN2Vbr-I&si=nWsA8v5WgUYKm8RT