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

Trabalho 2 - AEDS 1

Implementação para o problema do caixeiro viajante e análise de complexidade do algoritmo

EDMARCOS MENDES PINTO FILHO [05387]
DAVI DE SOUZA FERREIRA DO CARMO [5368]
LUCAS CURTY RODRIGUES MATHEUS [5793]

Florestal - MG
2023

Sumário

1. Introdução

Para o desenvolvimento de um bom sistema computacional, mais do que fornecer a resposta adequada para um problema, a solução necessita ser viável e possível de se utilizar no mundo real . Dessa forma, o presente projeto objetiva a aplicação dos conhecimentos adquiridos em sala de aula sobre análise de complexidade de algoritmos, para desenvolver e avaliar o custo e a complexidade de uma implementação para o **problema do caixeiro viajante**, considerado intratável por seu comportamento assintótico exponencial.

2. Organização

A estrutura do projeto possui um diretório principal, em que se encontram os arquivos git (como Readme e .gitignore), a pasta src, onde se encontra de fato o código desenvolvido, a pasta docs, em que se encontra a documentação, e a pasta tests, onde há um exemplo de entrada para o modo arquivo. Na Figura 1 é possível observar a estrutura mencionada.

A pasta src possui um arquivo main.c em seu primeiro nível, que é o arquivo principal por onde a aplicação é iniciada. Além disso, possui duas outras pastas, a headers, em que ficam os arquivos .h, e a sources, em que se encontram as implementações nos arquivos .c.

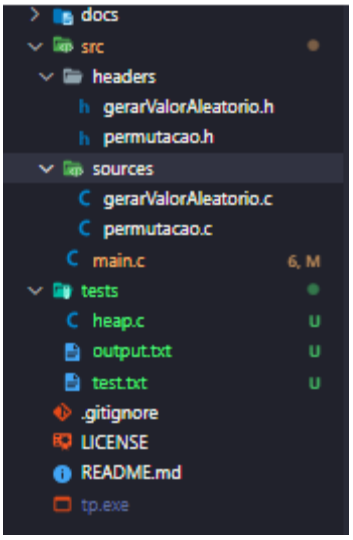

Figura 1 - Repositório do projeto.

3. Desenvolvimento

A construção do projeto envolveu tanto a compreensão do problema do Caixeiro Viajante, de como solucioná-lo, quanto a modelagem das funções necessárias para sua

solução.

Para isso foi necessário a busca de um código que fornecesse a permutação de N valores, sendo requisitado na especificações do trabalho. Assim, após uma busca minuciosa, foi encontrado o código desenvolvido por Rahul Agrawal, disponível no Web Site GeeksforGeeks [4], escolhido por ser um código mais legível pelo nível de lógica computacional do grupo, além de possuir uma boa performance comparada a outros algoritmos do gênero.

Para o cálculo do melhor projeto possível e sua distância, foi necessário analisar todas as possibilidades de trajeto do problema, partindo inicialmente de uma cidade cujo valor se daria pelo cálculo da soma de todos os algarismos da matrícula dos membros pertencentes ao grupo pelo resto da divisão por N, sendo N inserido anteriormente pelo usuário. Desta forma, foi utilizada a função de permutação para analisar todos os trajetos possíveis. Assim o caminho com a menor distância percorrida seria o melhor sentido de cidades que o problema poderia possuir, comparando assim, distância por distância para encontrar a menor entre as permutações;

Além disso, foi desenvolvido uma interface que executa a forma que o usuário preferir, podendo ter a entrada de modo interativo, no qual o próprio insere o valor de N, para formar uma matriz de N por N com valores aleatórios, valores estes representando a distância entre as cidades, e o modo de arquivo no qual é inserido um arquivo com valores predefinidos de N e dos valores da matriz. Além disso, possui dois métodos de saída dos resultados do processo, podendo ser pelo terminal ou armazenando as informações em um arquivo, contendo ambos o melhor caminho junto com sua distância total e o tempo de execução que o algoritmo demorou para encontrar a solução.

3.1 Permutação

A funcionalidade de permutação é a operação principal do programa, uma vez que proporciona todas as combinações de caminhos possíveis a serem percorridos, que são necessários para o cálculo de todas as possíveis distâncias.

O algoritmo utilizado vale-se da recursividade para realizar as trocas entre os elementos de um dado vetor **a** de tamanho **n**. Sua lógica consiste em realizar uma repetição pelo número de vezes correspondente à quantidade de cidades e, para cada repetição, a função de permutação é chamada novamente com o parâmetro **size** reduzido em uma unidade, sendo utilizado para determinar que os elementos nas posições [i] ou [0] e [size-1]

sejam permutados entre si. No fim, a lógica do algoritmo é executada **size x size-1 x size -2 x […] x 1** vezes, garantindo que todas as combinações possíveis sejam atendidas. A implementação pode ser visualizada na figura 2.

```
23  int permutar(int a[], int size, int n, int x, int **distancias, int *melhorCaminho) {
24      int menorDistancia = 0;
25      for (int i = 0; i < size; i++) {
26          int distancia = 0;
27          permutar(a, size - 1, n, x, distancias, melhorCaminho);
28          distancia+= distancias[x][a[0]];
29          for(int k = 0; k < n; k++) {
30              if( k+1 < n) {
31                  int cidadeA = a[k];
32                  int cidadeB = a[k+1];
33
34                  distancia+= distancias[cidadeA][cidadeB];
35              };
36              melhorCaminho[k] = a[k];
37          }
38
39          distancia+= distancias[a[n-1]][x];
40
41          if(i == 0) {
42              menorDistancia = distancia;
43          } else {
44              if(distancia < menorDistancia) {
45                  menorDistancia = distancia;
46              }
47          }
48
49          if (size % 2 == 1) {
50              trocar(&a[0], &a[size - 1]);
51          } else {
52              trocar(&a[i], &a[size - 1]);
53          }
54      }
55
56      return menorDistancia;
57  }
```

Figura 2 - permutacao.c - permutar().

3.2 Cálculo de distâncias

Além da permutação, outro ponto importante para a solução do problema é o cálculo da menor distância possível. Essa função encontra-se acoplada dentro da função de permutação e pode ser verificada ainda na figura 2.

É sabido que, como boa prática de desenvolvimento de software, cada função deve possuir uma responsabilidade única, por isso foi inicialmente idealizado que a função permutação retornaria uma matriz com todas as permutações possíveis e a distância seria calculada percorrendo cada linha da matriz e realizando o somatório das distâncias entre as cidades nela presentes. Apesar disso, foi percebido que isso agravaria os custos do algoritmo,

uma vez que uma nova operação de complexidade O(n^2) seria adicionada.

Dessa forma, optou-se por, no momento em que a permutação é gerada, realizar o somatório das distâncias entre os elementos, considerando sempre a cidade a[k] e a cidade seguinte a[k+1] como os índices para se obter o valor na matriz de distâncias. Ao mesmo tempo em que cada somatório é realizado, a permutação candidata a menor caminho é salva no vetor menor caminho. Ao final, comparando cada somatório, encontra-se o menor valor de distância possível e o menor caminho correspondente se encontra salvo em um vetor para ser utilizado.

4. Compilação e Execução

Para compilar e executar o projeto não é necessária nenhuma configuração especial, sendo necessário apenas possuir o compilador GCC instalado.

Para compilar, basta digitar o comando: *gcc .\src\main.c .\src\sources\permutacao.c .\src\sources\gerarValorAleatorio.c -o tp*. E para executar basta apenas executar o arquivo binário "tp" gerado. Em sistema Windows: *.\tp.exe*. Em sistemas Unix: *./tp*.

Vale ressaltar que os comandos citados consideram o diretório raiz como local onde serão executados. Caso o utilizador esteja em outro diretório, deverá ajustar o caminho relativo dos arquivos. Quanto ao arquivo de entrada, pode estar em qualquer diretório e basta que o caminho relativo ao executável do projeto seja digitado corretamente.

5. Resultados

Após a realização da atividade proposta, foi possível construir a aplicação solicitada com todas as suas funcionalidades e seus dois modos de operação, interativo e por leitura de arquivo. O modo é definido pelo usuário ao iniciar o algoritmo.

No modo interativo, é criado uma matriz N por N, com valores aleatórios. Após é calculado a menor distância possível entre todas as possibilidades, apresentando a menor distância possível, melhor trajeto e o tempo de execução do algoritmo para analisar todas as possíveis rotas. Essas informações podem ser passadas por intermédio do console ou por arquivo, escolha essa realizada pelo próprio usuário.

O outro modo existente, é de ler um arquivo com valores predefinidos, possuindo o tamanho N, sendo o tamanho da matriz N por N, podendo ter as saídas demonstradas tanto em arquivo quanto no terminal. Os modos de saída podem ser conferidos nas figuras 3 e 4.

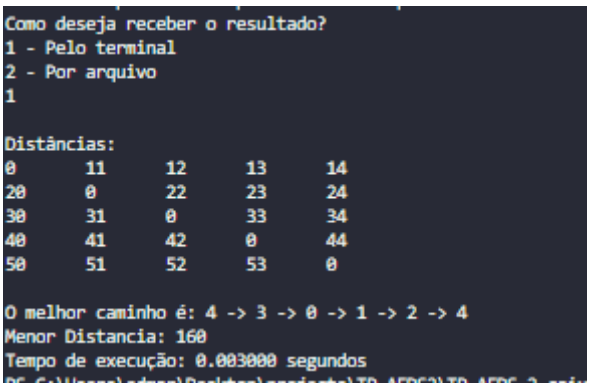

Figura 3 - Saída pelo terminal

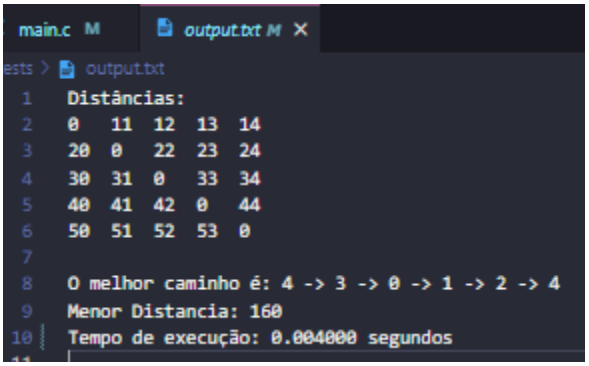

Figura 4 - Saída por arquivo

O algoritmo para um número pequeno de entradas possui o tempo de execução baixo, porém ao inserir valores médios e relativamente grandes o tempo de execução assume valores estratosféricos, analisados na Tabela 1 e na Figura 5.

Tamanho de Entrada	Tempo de Execução
5	0.000000
6	0.000000
7	0.000000
8	0.001000
9	0.011000

10	0.067000
11	0.622000
12	12.595000
13	93,480000
14	1436,086000

Tabela 1 - Tamanho de Entrada x Tempo de Execução.

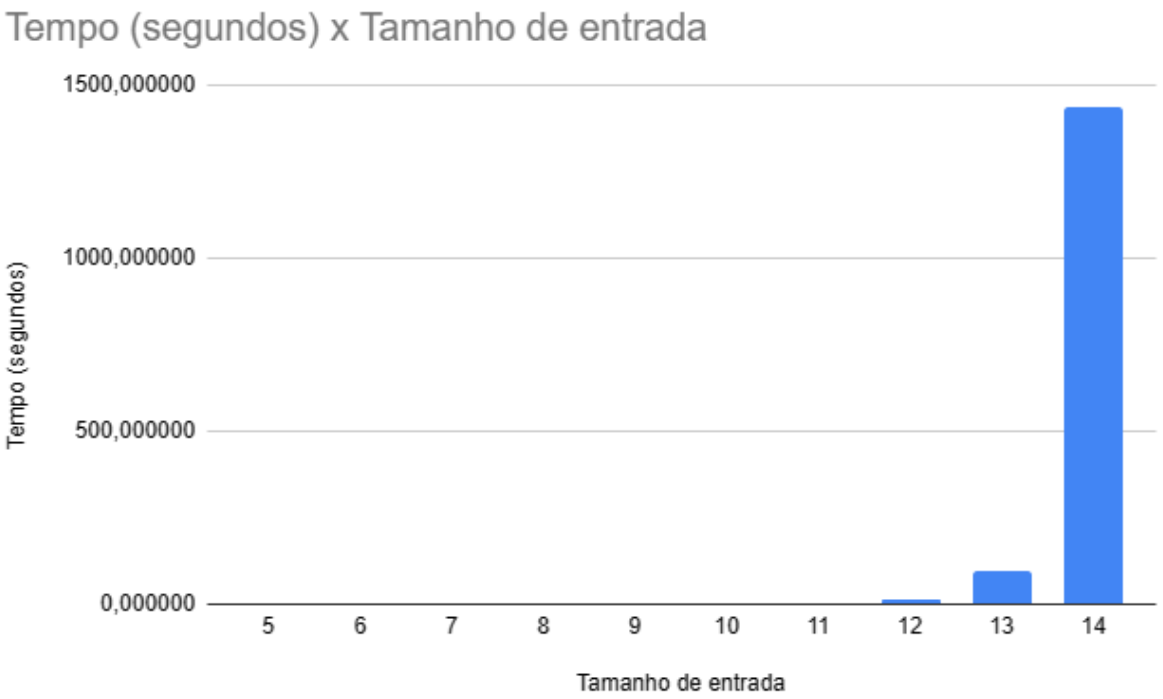

Figura 5 - Gráfico de Tempo x Tamanho de Entrada

Ressalta-se, também, que o tempo de processamento pode variar de acordo com as especificações do computador. Um computador mais robusto, por exemplo, é capaz de processar dados de maneira mais eficiente e rápida. Isso significa que ele pode executar soluções de maneira mais econômica em termos de tempo. Nos casos de teste deste trabalho, foi utilizado uma máquina com as configurações apresentadas na figura 6.

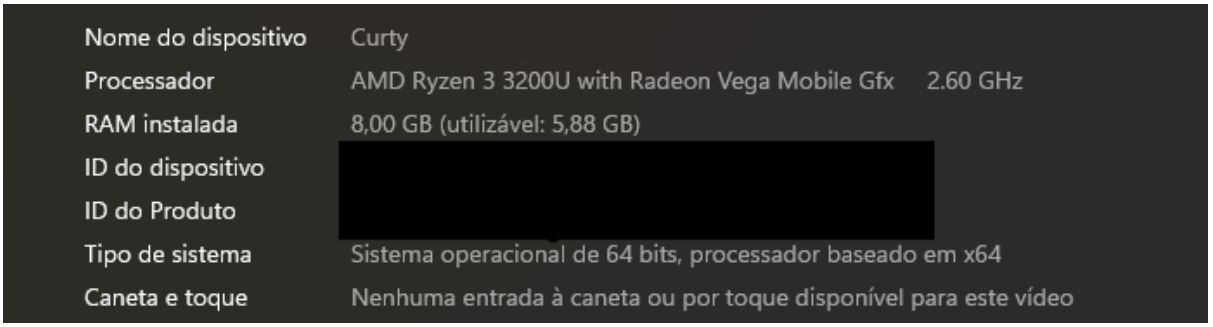

Figura 6 - Configurações do computador de testes

6. Conclusão

Diante do que foi apresentado, percebe-se que o desenvolvimento do trabalho contribuiu para um melhor aproveitamento e absorção dos conteúdos abordados em sala de aula, visto que a análise da medida do tempo de execução de um programa foi primordial para o entendimento do resultado final do trabalho.

O presente estudo desenvolveu um projeto no qual resolve o problema do caixeiro viajante para valores pequenos, encontrando o menor valor de distância total dentre as N cidades. Para tanto, foi necessário o desenvolvimento de diversas funcionalidades para encontrar um valor coeso para o problema. Porém ao analisar o problema por um olhar mais próximo à realidade, foi verificado um grande tempo para a máquina processar todos os dados e possibilidades para gerar o melhor trajeto possível, tornando inviável para valores de entradas grandes. Desta forma, os testes foram feitos de forma em que a resposta fosse dada em um tempo razoável, gerando com êxito os melhores valores.

Destaca-se portanto, que a implementação do algoritmo em paralelo a realidade, tornaria inviável o processamento da solução para valores médios ou grandes de entrada, pois se trataria de um gasto de tempo impraticável, posto uma empresa que necessita realizar entrega de mercadorias em centenas de cidades, tempo esse que poderia passar de anos, décadas ou até séculos para identificar a melhor rota. Portanto, pode-se afirmar que uma empresa não utilizaria tal solução em seu dia a dia.

7. Referências

[1] Github. Disponível em: <https://github.com/> Último acesso em: 31 de outubro de 2023.

[2] Visual Studio Code. Disponível em: <https://visualstudio.microsof t.com/pt-br/> Último acesso em: 22 de outubro de 2023.

[3] Linguagem C Descomplicada. Disponível em: <https://programacaodescomplicada.wordpress.com/> Último acesso em: 28 de outubro de 2023.

[4] Heap's Algorithm for generating permutation. Disponível em: <https://www.geeksforgeeks.org/heaps-algorithm-for-generating-permutations/> Último acesso em: 31 de outubro de 2023.