UNIVERSIDADE DE SÃP PAULO - USP
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

O Problema do Caixeiro Viajante.

Bruno da Freiria Mischiati Borges
Daniel Coutinho Ribeiro
Yan Köhler de Araujo

São Paulo, 29 de outubro de 2021

Resumo

Esse trabalho apresenta uma solução pelo método de força bruta para o Problema do Caixeiro Viajante. Além disso, é apresentada a Estrutura de Dados utilizada na resolução do Problema e também é feita a análise de desempenho de todas as funções criadas para implementação do algoritmo de solução.

Sumário

1 Introdução

Este relatório contém a modelagem para solução do Problema do Caixeiro Viajante e a análise assintótica de complexidade computacional da solução proposta. Na modelagem, são explicitadas a lógica utilizada para resolução do Problema e a justificativa para o tipo de Estrutura de Dados utilizada na implementação da solução. Já na análise assintótica, é apresentada a complexidade de cada função empregada na implementação do algoritmo de solução (utilizando a notação Big-Oh).

2 Modelagem da Solução

Nesse capítulo será apresentado o tipo de Estrutura de Dados escolhido para resolução do Problema do Caixeiro Viajante, bem como as vantagens e limitações que essa estrutura apresenta. Além disso, também será apresentada a lógica empregada na resolução do problema proposto.

2.1 Estrutura de Dados Escolhida

A Estrutura de Dados escolhida para modelar o problema do Caixeiro Viajante foi a de Listas Lineares Dinâmicas Duplamente Encadeadas. A escolha dessa estrutura se deu pela necessidade de efetuar, constantemente, operações de inserção, remoção, busca e acesso nas listas elaboradas para solução do problema. Além disso, o método escolhido para solução do problema envolve a criação de inúmeros nós e de outras listas. Essas tarefas são predominantemente presentes – e facilitadas – pela Estrutura de Dados escolhida.

2.2 Vantagens e Limitações da Listas Dinâmicas Duplamente Encadeadas

A implementação das Listas Duplamente Encadeadas beneficia operações de remoção e inserção, além de torna-lás mais eficientes em relação ao tempo. Por outro lado, a implementação desse tipo de Estrutura de Dados é de alta complexidade.
As operações citadas são beneficiadas porque o tipo de implementação escolhida diminui a quantidade de variáveis auxiliares necessárias para essas operações. Além disso, acessar e percorrer as listas duplamente encadeadas é mais fácil, umas vez que é possível percorrê-la em dois sentidos: do começo para o fim e do fim para o começo.
Já o desempenho em relação ao tempo decorre do fato de que a lista utilizada permite que remoções e inserções ocorram mais rapidamente, pois os elementos da lista não estão em regiões contíguas de memória e, dessa forma, para acessá-los, basta utilizar referências.
Entretanto, como já citado, a limitação do tipo de Estrutura de Dados escolhido é o fato de que a implementação de uma lista duplamente encadeada é de alta complexidade, pois envolve a implementação de inúmeras referências entre os nós da lista. Ademais, o uso incorreto de referências pode ocasionar perda total ou parcial da lista.

2.3 Lógica utilizada para solução do problema

A lógica utilizada consiste em gerar todos as rotas possíveis em sequência e salvar a melhor encontrada para o retorno do programa.
Para gerar todas as combinações de rotas possíveis foi utilizada a ordem lexicográfica. Essa ordem implica ordenação das rotas geradas e, dessa forma, também implica obter todas as combinações possíveis com garantia de sucesso.
Basicamente, a ordem lexicográfica é definida pela ordenação de um conjunto de elementos de acordo com uma ordenação lógica preestabelecida deles. Ou seja, supondo que cada cidade do problema do caixeiro viajante corresponda à um número, duas ordenações possíveis de uma rota seriam (considerando um total de 4 cidades, por exemplo): $1 \to 2 \to 4 \to 3$ e $1 \to 2 \to 3 \to 4$. Mas, como $1 < 2 < 3 < 4$, seguindo a ordem lexicogŕafica, na ordem das permutações geradas, $1 \to 2 \to 3 \to 4$ seria gerado primeiro e $1 \to 2 \to 4 \to 3$ seria gerado depois. Posteriormente, o programa começa atribuindo para a rota atual, e para a melhor rota, a rota $1 \to ... \to n$ (excluindo a cidade de início), e atribui a melhor distância para a distância da rota atual. Em seguida, o programa iniciará um laço que irá rodar até que a rota atual seja da forma $n \to ... \to 1$, atribuindo no final de cada iteração a rota atual para sua próxima permutação na ordem lexicográfica. Caso a rota atual seja melhor, ou seja, sua distância seja menor que a menor distância até agora, ela é atribuida para melhor rota e sua distância para menor distância até agora. Assim, após a execução do laço, o programa retornará a melhor rota.

3 Análise de Complexidade

Nesse capítulo será feita a análise assintótica da complexidade temporal da solução proposta para o Problema do Caixeiro Viajante.Para isso, serão apresentadas tabelas em que cada linha de uma delas contem uma função e sua respectiva complexidade. Além disso, também será apresentado um gráfico com as medidas de tempo de execução em função do tamanho do conjunto de entrada.

3.1 Análise de assintótica das funções auxiliares

A partir da análise assintótica de cada função utilizada na solução do Problema do Caixeiro Viajante, foi possível definir a complexidade de cada uma dessas funções. A seguir, serão apresentadas tabelaa contendo cada função e sua respectiva complexidade computacional, em seus respectivos TAD's.

TAD TSP	
Função	Complexidade
tsp_read_distance_file	$\mathcal{O}(n)$
tsp_solve_bruteforce	$\mathcal{O}(n!)$

TAD Distance List	
Função	Complexidade
distance_list_node_get_distance_from_to	$\mathcal{O}(n)$
distance_list_print	$\mathcal{O}(n)$
distance_list_is_full	$\mathcal{O}(1)$
distance_list_unshift	$\mathcal{O}(1)$
distance_list_insert_after	$\mathcal{O}(1)$
distance_list_insert_before	$\mathcal{O}(1)$
distance_list_push	$\mathcal{O}(1)$
distance_list_free	$\mathcal{O}(1)$
distance_list_delete_head	$\mathcal{O}(1)$
distance_list_delete_tail	$\mathcal{O}(1)$
distance_list_search_with_from_to	$\mathcal{O}(n)$
distance_list_search	$\mathcal{O}(n)$
distance_list_delete	$\mathcal{O}(1)$
distance_list_get_size	$\mathcal{O}(n)$
distance_list_set_head	$\mathcal{O}(1)$

TAD Distance List	
Função	Complexidade
distance_list_set_tail	$\mathcal{O}(1)$
distance_list_get_head	$\mathcal{O}(1)$
distance_list_get_tail	$\mathcal{O}(1)$
distance_list_is_empty	$\mathcal{O}(1)$
distance_list_new	$\mathcal{O}(1)$

TAD Distance List Node	
Função	Complexidade
distance_list_node_get_prev	$\mathcal{O}(1)$
distance_list_node_get_next	$\mathcal{O}(1)$
distance_list_node_set_prev	$\mathcal{O}(1)$
distance_list_node_set_next	$\mathcal{O}(1)$
distance_list_node_create	$\mathcal{O}(1)$
distance_list_node_free	$\mathcal{O}(1)$
distance_list_node_get_key	$\mathcal{O}(1)$
distance_list_node_set_key	$\mathcal{O}(1)$
distance_list_node_get_to	$\mathcal{O}(1)$
distance_list_node_set_to	$\mathcal{O}(1)$
distance_list_node_get_from	$\mathcal{O}(1)$
distance_list_node_set_from	$\mathcal{O}(1)$
distance_list_node_get_distance	$\mathcal{O}(1)$
distance_list_node_set_distance	$\mathcal{O}(1)$
distance_list_node_print	$\mathcal{O}(1)$

TAD Path	
Função	Complexidade
path_unshift	$\mathcal{O}(1)$
path_copy_to	$\mathcal{O}(n)$
path_populate	$\mathcal{O}(n^2)$
path_get_next_permutation	$\mathcal{O}(n^2)$
path_calculate_distance	$\mathcal{O}(n^2)$
path_insert_after	$\mathcal{O}(1)$
path_insert_before	$\mathcal{O}(1)$
path_print	$\mathcal{O}(n)$
path_print_with_start	$\mathcal{O}(n)$
path_is_full	$\mathcal{O}(n)$

TAD Path	
Função	Complexidade
path_push	$\mathcal{O}(1)$
path_free	$\mathcal{O}(1)$
path_swap	$\mathcal{O}(n)$
path_delete_head	$\mathcal{O}(1)$
path_delete_tail	$\mathcal{O}(1)$
path_search	$\mathcal{O}(n)$
path_delete	$\mathcal{O}(1)$
path_get_size	$\mathcal{O}(n)$
path_set_head	$\mathcal{O}(1)$
path_set_tail	$\mathcal{O}(1)$
path_get_head	$\mathcal{O}(1)$
path_get_tail	$\mathcal{O}(1)$
path_is_empty	$\mathcal{O}(1)$
path_new	$\mathcal{O}(1)$

TAD Path Node	
Função	Complexidade
path_node_get_prev	$\mathcal{O}(1)$
path_node_get_next	$\mathcal{O}(1)$
path_node_set_prev	$\mathcal{O}(1)$
path_node_set_next	$\mathcal{O}(1)$
path_node_create	$\mathcal{O}(1)$
path_node_free	$\mathcal{O}(1)$
path_node_get_key	$\mathcal{O}(1)$
path_node_set_key	$\mathcal{O}(1)$
path_node_print	$\mathcal{O}(1)$

3.2 Análise assintótica e gráfico da função brute_force (função principal)

A partir da análise do tempo de execução do código desenvolvido para solução do problema proposto, foi possível criar um gráfico que apresenta as medidas de tempo de execução em função do número de cidades fornecidas como arquivo de entrada. A complexidade da função que soluciona o problema (função brute_force) é $\mathcal{O}(n!)$ e, portanto, o gráfico apresentado também possui um comportamento fatorial.

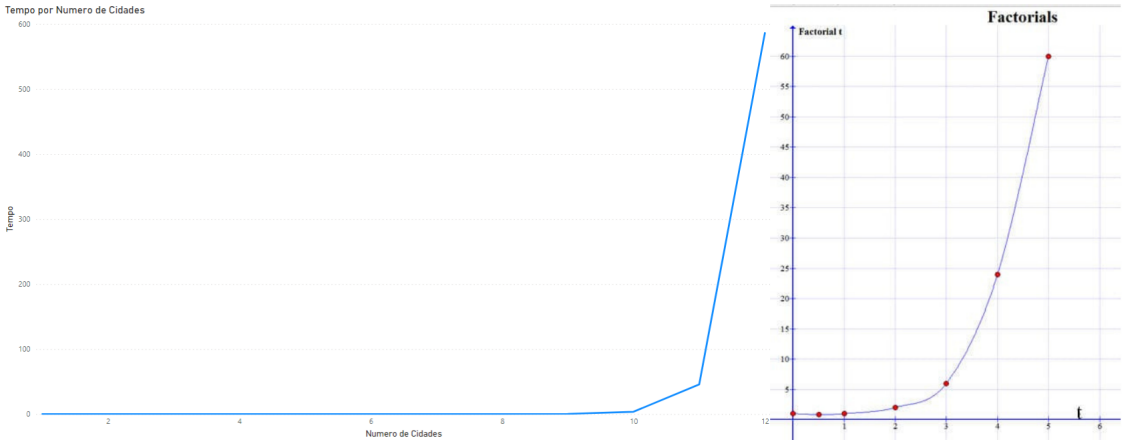

Gráfico da esquerda: tempo de execução da função. Gráfico da direita: $n!$

4 Referências

[1]. The Coding Train – Coding Challenge 35.1: Traveling Salesperson. Youtube. Disponível em: <https://www.youtube.com/watch?v=BAejnwN4Ccwt=550s>. Acesso em: 01 de out. 2021.

[2]. FORISEK, Michal. How would you explain an algorithm that generates permutations using lexicographic ordering?. Quora, 2021. Disponível em: <https://www.quora.com/How-would-you-explain-an-algorithm-that-generates-permutations-using-lexicographic-ordering>. Acesso em: 01 de out. 2021.

[3]. Wikipedia. Travelling salesman problem. Wikipedia, 2021. Disponível em: <https://en.wikipedia.org/wiki/Travelling_salesman_problem>. Acesso em: 01 de out. 2021.

[4]. Wikipedia. Doubly linked list. Wikipedia, 2021. Disponível em: <https://en.wikipedia.org/wiki/Doubly_linked_list>. Acesso em: 01 de out. 2021.