SCC0202 - Algoritmos e Estruturas de Dados I

Relatório do projeto 1

Alunos	**nºUSP**
Artur Kenzo Obara Kawazoe	15652663
Daniel Jorge Manzano	15446861
Fernando Valentim Torres	15452340

Explicação da solução força bruta

→ Estruturas

Um grafo na forma de um array de listas encadeadas dinâmicas foi utilizado para armazenar os dados do problema, cada lista armazenando as distâncias a outras cidades de uma cidade fixada. Essa estrutura apresenta o benefício de poupar espaço para casos em que certas conexões entre cidades não existem, mas também possui a desvantagem de utilizar algoritmos de busca mais lentos. Entretanto, desenvolvemos soluções usando ambas as aplicações de lista (sequencial e encadeada) e concluímos que as diferenças em tempo de execução eram mínimas, portanto, acabamos optando por usar o tipo encadeado para o ganho em memória uma vez que a vantagem de tempo da lista sequencial era baixa.

Um deque sequencial é utilizado para armazenar a ordem do caminho. Essa escolha foi feita porque a estrutura permite a utilização de um método fácil de permutar as cidades

→ Algoritmo

O algoritmo consiste em permutar uma sequência de cidades, testando a distância da rota gerada e salvando-a caso tenha sido a menor encontrada, repetidamente, até exaustir todas as possibilidades de caminhos possíveis. O algoritmo não faz permutação da cidade inicial, a qual é fixada, e não gera caminhos impossíves - por exemplo, dadas 5 cidades, se 2 não se liga a 3, nenhum caminho em que 2 é adjacente a 3 é calculado.

→ Complexidade

Para a análise da complexidade dos códigos, dividiremos nossa explicação entre os diferentes arquivos .c (graph.c, vertice.c, brute.c e main.c) e suas respectivas funções.

· **graph.c**

As funções que criam e apagam grafo devem percorrer toda a estrutura, então possuem complexidade O(n). Além disso, a função "insertEdge" possui a mesma complexidade de "createConnection" (função de vertice.c, será analisada mais a frente), que é O(n). Por fim, as funções de "get" possuem complexidade O(1).

· **vertice.c**

As funções de criação e busca de conexões ("createConnection" e "getWeightConnection") funcionam com base em um algoritmo de busca ordenada em lista encadeada, portanto possuem complexidade O(n). As demais funções, que alocam e apagam no TAD, têm complexidade O(1).

· **brute.c**

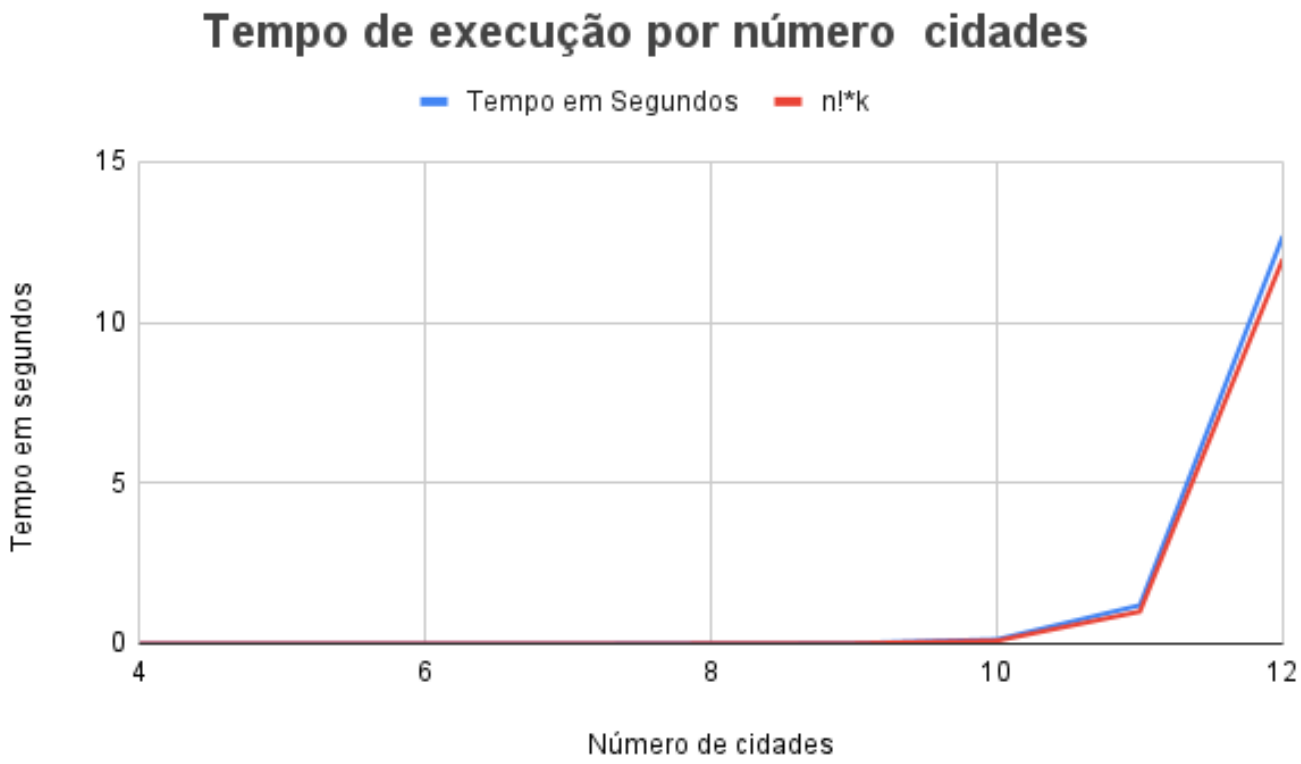
As função "bruteForce" e "bestPath" são onde ocorre, essencialmente, a resolução do problema como um todo. Fazendo todas permutações de caminhos possíveis recursivamente, excluindo a cidade de partida, bestPath possui complexidade de O((n-1)!) e, além disso, ao seu fim, chama a função "getWeightConnection", de complexidade O(n). Assim, tanto bestPath

quanto bruteForce (que faz a chamada para bestPath) possuem complexidade de O(n.(n-1)!), ou seja, O(n!). As demais funções se baseiam em operações mais "pontuais", todas com complexidade de O(1).

· **main.c**

Na main, ocorrem apenas chamadas a funções já criadas nos TADs e, dentre essas, a complexidade que "predomina" é a da função "bruteForce". Dessa forma, a complexidade da função main do programa é, também, O(n!).

→ Gráfico



O gráfico apresenta o tempo de execução do programa e a linha de $n!k$, k sendo uma constante para que o os valores estejam em escala próxima. O fato do formato dos gráficos se aproximarem confirma que a complexidade é, como foi calculada antes, de $n!$

Explicação da solução otimizada

→ Estruturas

Para a solução do problema de maneira eficiente foi utilizada: uma matriz de adjacências, que armazena as distâncias entre cidades; uma matriz de programação dinâmica, que armazena as distâncias ideais para cada conjunto de cidades; uma matriz pai utilizada para armazenar o caminho ideal; uma bitmask utilizada para armazenar as combinações de cidades, desempenhando o papel de marcar as cidades já visitadas em um caminho.

→ Algoritmo

O algoritmo resolve o Problema do Caixeiro Viajante utilizando a técnica de programação dinâmica. Ele calcula o caminho mais curto para subconjuntos de cidades, começando com conjuntos menores e aumentando progressivamente até alcaçar o tamanho do número de cidades do problema

A implementação envolve uma estratégia de recursão para calcular a distância mínima para cada máscara (conjunto de cidades visitadas) e uma cidade específica. Para cada máscara, o algoritmo testa todas as cidades que ainda não estão no conjunto, calculando o custo de adicionar cada uma delas à rota atual. A menor distância encontrada após testar todas as cidades fora do conjunto é armazenada na tabela de programação dinâmica (dp).

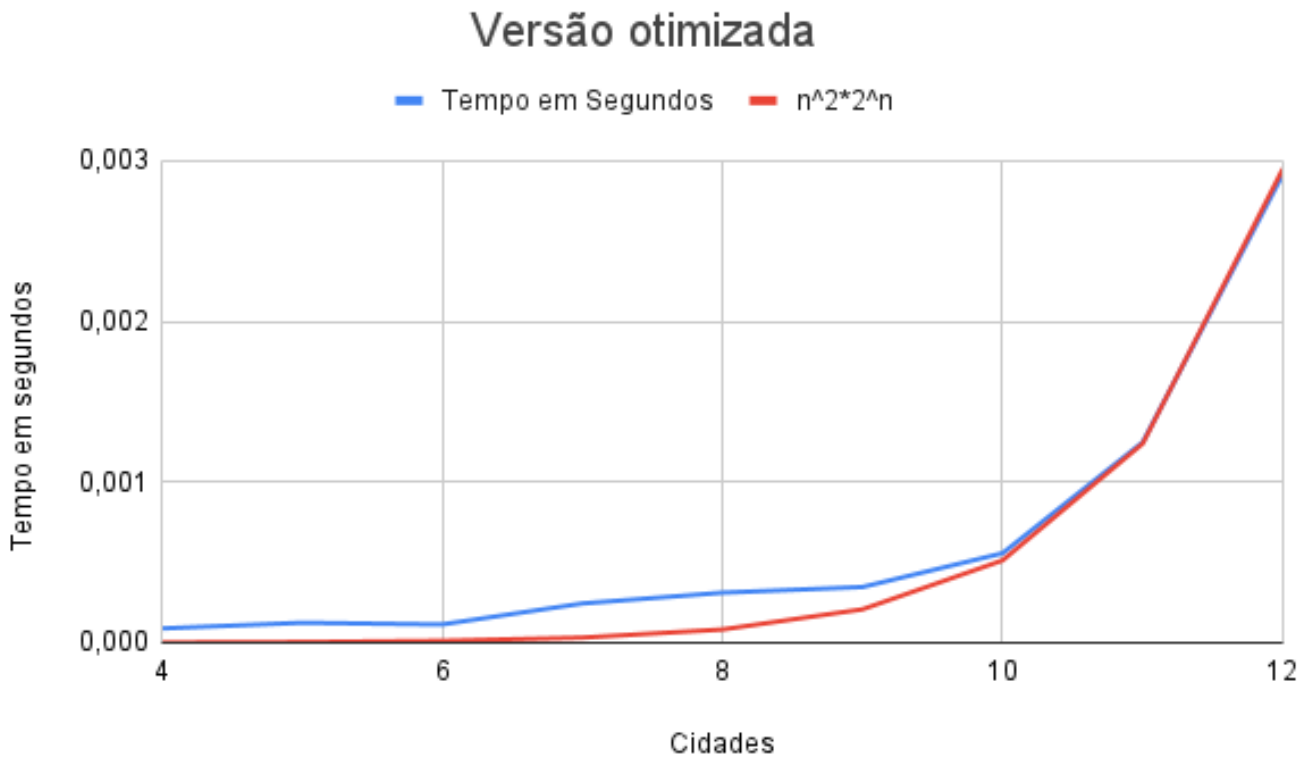
Ao mesmo tempo, a matriz pai registra a última cidade visitada no caminho ótimo para cada máscara. Isso permite que o caminho seja reconstruído posteriormente. Ao fim do cálculo, o caminho ótimo é retraçado a partir da matriz pai, começando na máscara que contém apenas a cidade inicial. Como essa posição contém a última cidade visitada no caminho, ela aponta para a penúltima cidade, que aponta para a antepenúltima, e assim por diante.

A otimização do algoritmo reside na possibilidade de aproveitar resultados de cálculos feitos previamente nos cálculos seguintes, poupando processos desnecessários.

→ **Análise de complexidade**

O algoritmo realiza o cálculo da distância dos 2^n conjuntos possíveis representados pela bitmask, para cada conjunto é testada uma quantidade proporcional as n candidatas para ser adicionada ao subconjunto, também para cada subconjunto devem ser testadss uma quantidade proporcional a n de possíveis fim de rota. Portanto a complexidade do algoritmo é de $O(n^2 2^n)$.

→ **Gráfico**



Como se pode observar o gráfico de tempo de execução parecido com a expressão $n^2 2^n$, mostrando que essa é mesmo a complexidade do algoritmo. Também se percebe que o algoritmo é bem mais rápido que o força bruta pelos valores apresentados.