Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Departamento de Ciências de Computação (SCC)
Disciplina: Algoritmos e Estruturas de Dados 1

PROJETO 1 – PROBLEMA DO CAIXEIRO VIAJANTE

Grupo 23:
Beatriz Alves dos Santos
Kattryel Henrique Santos Rezende
Marcos Vinicius Cota Rodrigues da Trindade

1. INTRODUÇÃO

Este relatório apresenta o desenvolvimento do projeto intitulado “Problema do Caixeiro Viajante” (PCV). O objetivo é encontrar, a partir de uma cidade de origem, o trajeto de menor custo que passe por todas as cidades, retornando à cidade inicial.

A solução do projeto envolveu a modelagem do problema utilizando Estruturas de Dados, a implementação de um algoritmo de força bruta em Linguagem C, e a análise da complexidade computacional do resultado. A implementação seguiu princípios de modularização e uso de TAD, visando a organização e clareza do código.

A análise de complexidade teórica foi complementada por medições de tempo de execução para diferentes tamanhos de entrada, permitindo avaliar o desempenho da solução proposta. Os resultados obtidos e as conclusões acerca do comportamento do algoritmo são apresentados nas seções seguintes deste relatório.

2. MODELAGEM DE SOLUÇÃO

A modelagem da solução para o Problema do Caixeiro Viajante (PCV) foi realizada utilizando uma lista de adjacências para representar o grafo que descreve as conexões entre as cidades e suas respectivas distâncias. Cada cidade é representada por uma posição em um vetor de listas encadeadas. Cada nó de uma lista, representa um caminho (aresta) diferente partindo de uma mesma cidade (definida pela posição no vetor), armazenando a cidade adjacente e sua respectiva distância.

A escolha da lista de adjacências para modelar as conexões entre as cidades foi motivada pela necessidade de armazenar de forma eficiente apenas as distâncias relevantes entre pares de cidades conectadas. Essa estrutura de dados permite acessar rapidamente as cidades adjacentes a partir de qualquer nó, o que facilita o cálculo das distâncias entre cidades durante a geração das permutações dos trajetos.

A solução proposta utiliza um algoritmo de força bruta, que avalia todas as permutações possíveis das cidades para identificar o menor trajeto que passe por todas elas e retorne à cidade de origem. Para isso, uma função de permutação gera todas as sequências possíveis de visitas às cidades, fixando a cidade de origem e variando as demais cidades na sequência. O método de *backtracking* é empregado nessa geração de permutações, permitindo explorar as combinações de cidades de maneira sistemática.

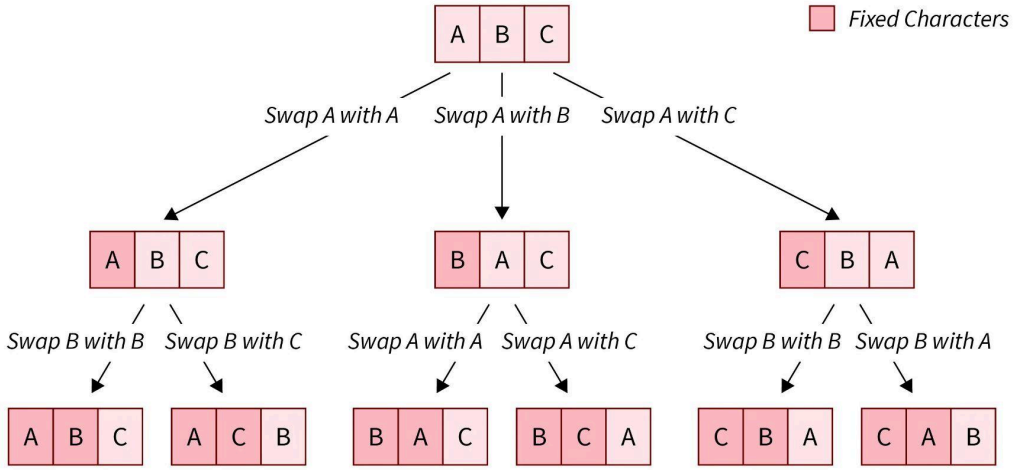


Diagrama ilustrativo retirado de https://www.scaler.in/permutation-of-string/

O *backtracking* funciona trocando elementos no vetor que representa as cidades, criando assim novas sequências. Quando uma permutação é completada, a distância total do percurso é avaliada. Em seguida, a função reverte as trocas (*backtrack*) para restaurar o estado anterior e explorar outras combinações. Esse processo continua até que todas as permutações possíveis tenham sido geradas e avaliadas.

Cada permutação criada é analisada por uma função que calcula a distância total do percurso. Essa função percorre a lista de adjacências para encontrar as distâncias entre cidades consecutivas no trajeto e calcula a soma dessas distâncias. Caso uma conexão entre duas cidades não exista, o valor da distância é considerado infinito (representado por uma constante), o que exclui o trajeto da avaliação como um possível candidato ao menor caminho.

O uso de um Tipo Abstrato de Dados (TAD) para a lista de adjacências e para os itens de cada aresta (cidade inicial, cidade destino e distância) garante a modularização do código, separando as operações de manipulação dos dados de sua representação interna. Isto torna a

implementação mais organizada e facilita futuras alterações ou melhorias no código, além de promover a reutilização dos componentes desenvolvidos.

3. IMPLEMENTAÇÃO

Arquivos do projeto:

- **Lista.c:** Implementação do TAD lista encadeada de adjacências;
- **Lista.h**: Interface do TAD lista;
- **pcv.c**: programa cliente para resolução por força bruta;
- **pcv_otimo.c**: programa cliente para solução otimizada;
- **MakeFile**: arquivo makefile com os atalhos para a compilação e execução.
    - *make all* : executa a compilação dos dois programas clientes da forma correta;
    - *make run_pcv* : roda o programa cliente com a solução por força bruta;
    - *make run_pcv_otimo* : roda o programa cliente com a solução otimizada;
    - *make clean* : deleta os arquivos objetos e executáveis.

4. ANÁLISE DE COMPLEXIDADE

A análise da complexidade computacional da solução proposta para o Problema do Caixeiro Viajante (PCV) envolve a avaliação das operações realizadas nas estruturas de dados utilizadas, bem como a complexidade do algoritmo de força bruta empregado para encontrar o menor trajeto.

4.1.LISTA DE ADJACÊNCIAS

- Cada lista de adjacências armazena as conexões entre as cidades e suas respectivas distâncias.
- As operações básicas realizadas nesta estrutura são:
    - Inserção de Aresta: Para inserir uma aresta entre duas cidades, a complexidade é O(1) pois a nova adjacência é inserida no início da lista. No entanto, a complexidade total para inserir todas as arestas será O(n), onde n é o número total de arestas.
    - Busca de Distância: Para encontrar a distância entre duas cidades na lista de adjacências, a complexidade no pior caso é O(n), onde n é o número de cidades. Isso ocorre porque, na busca, pode ser necessário percorrer toda a lista até encontrar a cidade desejada.

4.2.CÁLCULO DAS DISTÂNCIAS

- A função *calcula_distancia* percorre a lista de adjacências para cada par de cidades na permutação, resultando em uma complexidade de O(n) para cada par, onde n é o número total de cidades. Considerando que a função verifica $n - 1$ pares (cidades consecutivas)

e um adicional para o retorno à cidade inicial, a complexidade total para calcular a distância de uma permutação é O(n^2).

4.3.ALGORITMO DE FORÇA BRUTA

O algoritmo de força bruta gera todas as permutações possíveis das cidades, exceto a cidade inicial, para avaliar o caminho mais curto. O número total de permutações de n cidades é dado por $(n-1)!$, pois a cidade inicial é fixa e as outras $n-1$ cidades podem ser permutadas de qualquer maneira. Assim, a complexidade do algoritmo para gerar todas as permutações é O($(n-1)!$). Para cada uma destas combinações, é calculado o custo do trajeto. Logo, combinando as duas partes, temos:

1. Cálculo das permutações: O($(n-1)!$);
2. Cálculo da distância para cada permutação: O(n^2);

Portanto, a complexidade total da solução implementada que encontra o menor trajeto para o PCV pode ser expressa como:

$$O((n-1)! \cdot n^2) = O(n! \cdot n^2)$$

5. EXTRA - SOLUÇÃO MAIS EFICIENTE

Para resolver o problema de forma mais eficiente, utilizamos o algoritmo *Branch and Bound*, que explora os caminhos possíveis e busca a menor distância de forma inteligente. Em vez de gerar todas as permutações possíveis, como na força bruta, o Branch and Bound utiliza o método de poda de ramos quando se depara com um caminho parcial que já não cumpre os requisitos para ser a melhor solução.

A estrutura utilizada foi a mesma da solução anterior: utilizamos um vetor de lista encadeada onde cada nó representa uma adjacência (aresta) da cidade correspondente à posição no vetor, armazenando o ID da cidade adjacente e a distância relativa a essa cidade.

Durante a execução do algoritmo, o percurso é construído de forma incremental. Para isso, é utilizado um vetor de inteiros que contém a identificação de cada cidade, exceto a da cidade onde o percurso se inicia, e um vetor auxiliar para indicar se uma cidade já foi visitada ou não em um caminho específico. O percurso é construído recursivamente de forma que em cada chamada da função é escolhida uma cidade não visitada para ser acrescentada ao caminho. Para cada cidade adicionada, calcula-se a distância percorrida até o momento e, caso essa distância exceda o valor da menor distância encontrada até então, o caminho é descartado

(poda). Esse processo de poda permite reduzir significativamente o número de soluções analisadas, eliminando caminhos que não levariam à solução ótima.

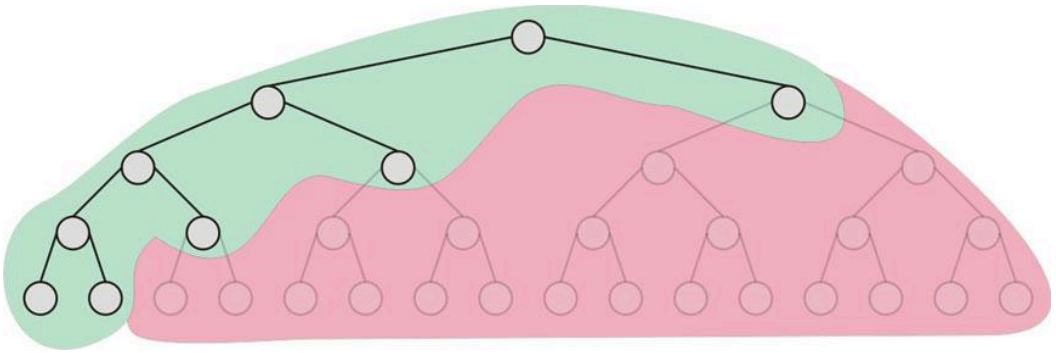


Diagrama ilustrativo retirado de https://www.control.isy.liu.se/en/research/optim/index.html

O algoritmo também utiliza *backtracking* para desfazer escolhas, ou seja, marcar uma cidade como não visitada novamente após o retorno da recursão, possibilitando explorar outros caminhos possíveis. Quando um percurso é completado (ou seja, todas as cidades foram visitadas), a distância total do trajeto é avaliada, ao incluir a distância da última cidade para a cidade de origem. Se sua distância for menor que a distância mínima registrada, ele é atualizado como o melhor trajeto encontrado até o momento. Vale mencionar que quando não há caminho entre duas cidades, a distância entre elas é assumida como sendo um valor grande o suficiente (definido anteriormente) para que o caminho seja considerado inviável.

O TAD utilizado foi o mesmo que na abordagem anterior, a diferença está apenas nas funções empregadas no programa e na maneira como elas foram utilizadas.

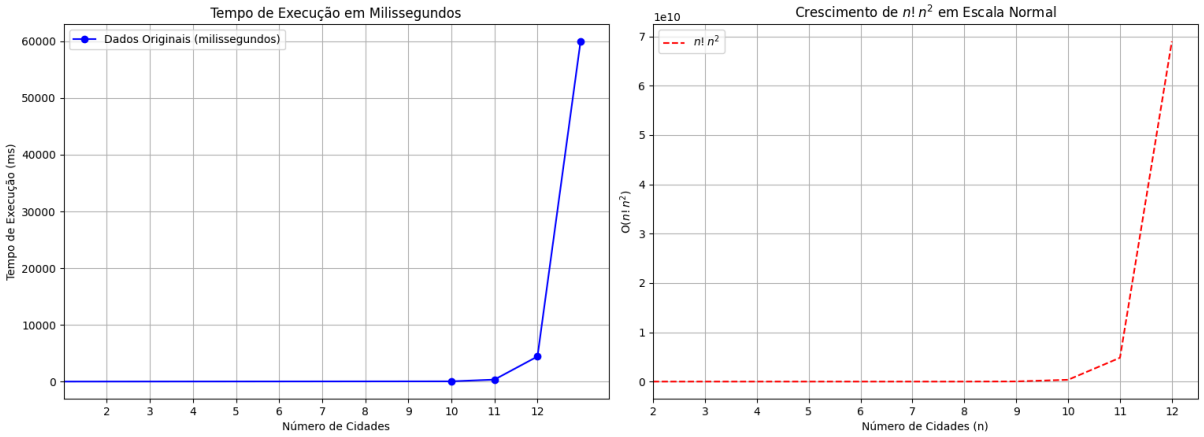
6. COMPARAÇÃO DE EFICIÊNCIA

Após a execução de casos de teste, os dados coletados foram os seguintes:

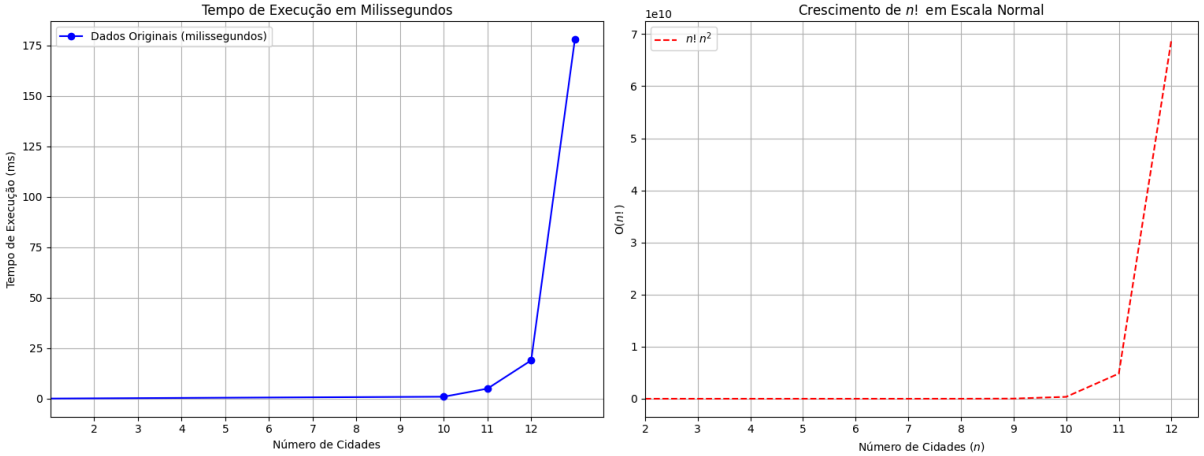
	Tempo de execução (em segundos)	
Nº de cidades	branch-and-bound	força bruta
13	0.178	59.979
12	0.019	4.421
11	0.005	0.346
10	0.001	0.04

Abaixo estão essas informações organizadas em dois gráficos, um para cada função. Além disso, esses gráficos estão junto de seus respectivos gráficos de complexidade em

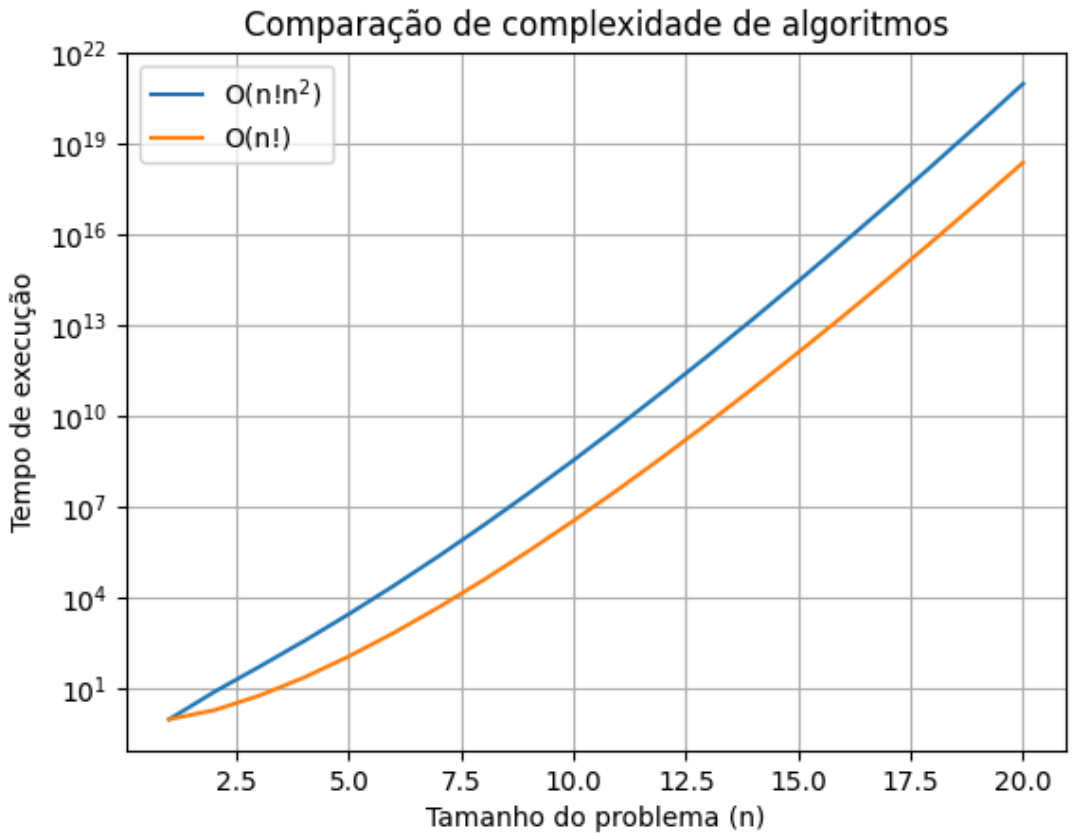
notação big-Oh, confirmando que eles de fato possuem a mesma forma. Por último há um gráfico comparando a complexidade dos dois algoritmos.



Comparação em gráficos dos casos de teste com o gráfico O(n²n!) do algoritmo de força bruta



Comparação em gráficos dos casos de teste com o gráfico O(n!) do algoritmo de branch-and-bound



Comparação em gráfico da complexidade dos dois algoritmos

Em suma, a força bruta garante a solução ótima, mas é inviável para instâncias grandes devido à sua complexidade. Já o branch-and-bound, ao eliminar soluções ineficientes, mantém a precisão com muito mais eficiência, sendo assim a melhor opção para resolver o problema. Isso pode ser visto já que, para 13 cidades, enquanto o algoritmo de força bruta leva quase 1 minuto para encontrar o melhor caminho, o branch-and-bound leva apenas 178 milissegundos. Essa tendência se repete para casos menores e maiores.