**ICMC - Instituto de Ciências Matemáticas e da Computação**

**Disciplina: SCC0202 - Algoritmo e Estrutura de Dados I**
**Projeto 1**

**Aluno:** Alec Campos Aoki (15436800)
**Aluno:** Jhonatan Barboza da Silva (15645049)
**Aluno:** Jõao Ricardo de Almeida Lustosa (15463697)

## 1. Modelagem da Solução

A lógica usada para resolver o problema é simples. Buscamos, nessa ordem:

1. Ler a entrada de dados e armazená-la;
   (a) Número total de cidades a serem visitadas;
   (b) Cidade inicial (e final);
   (c) Quantidade de conexões (estradas) existentes;
   (d) Pares de cidades e o comprimento da conexão entre elas.
2. Gerar todas as permutações possíveis dos caminhos que percorrem todas as cidades, partindo da inicial e voltando ao final;
   (a) Gerar todas as permutações possíveis, uma a uma (sem repetir permutações);
   (b) Verificar se a permutação gerada é válida;
   (c) Armazená-la se sua distância for a menor encontrada até o momento, e descartá-la caso o contrário;
3. Retornar a distância e seu caminho.

### 1.1 Leitura de Dados

Para ler os dados do problema, criamos uma *struct* chamada *INDICE* que armazena três inteiros: dois representando duas cidades, e o terceiro representando a distância entre elas. Armazenamos todos os INDICES em uma Estrutura de Dados **lista**, mais especificamente uma lista sequencial, pois iremos consultar frequentemente esses dados, e essa estrutura nos permite fazer isso facilmente.

### 1.2 Gerar Permutações

Para guardar as permutações conforme geramos elas, utilizamos a Estrutura de Dados **fila**, mais especificamente uma fila simplesmente encadeada não-ordenada. Essa fila consiste de um TAD, e foi escolhida pois suas operações de inserção e remoção são eficazes e simples. Utilizar uma Estrutura de Dado muito flexível como

uma lista duplamente encadeada demandaria recursos a mais desnecessariamente, e utilizar uma estrutura sequencial poderia dificultar as operações de inserção e remoção.

Geramos as permutações enfileirando cidades na fila e checando, a cada caso, se:

1. A permutação já está completa, isto é, se seu tamanho equivale à quantidade total de cidades;
2. O elemento a ser inserido já está na permutação.

Após conseguirmos uma permutação completa, checamos se ela é totalmente válida (ou seja, se não estamos conectando cidades que não possuem conexões diretas entre si) e, caso seja, se a distância desse percurso é a menor que encontramos até o momento. Se for, atualizamos nosso melhor caminho. Senão, realizamos uma nova permutação. Note que, se temos $n$ cidades, teremos $P_n = n!$ permutações, cada uma de tamanho $n$. Ao checarmos todas as permutações possíveis, teremos achado o percurso com menor distância. Importante notar que também que transformamos todas as operações usadas para solucionar o Problema do Caixeiro Viajante em um TAD próprio para facilitar a organização e modularização do código.

## 2. Implementação

Para compilar o código, executar o comando *"make all"*. Para rodar o código, executar o comando *"make run"*. O programa principal é o arquivo *"main.c"*; a implementação do TAD da solução está no arquivo *"PCV.c"*, a implementação do TAD da fila está no arquivo *"fila.c"* e a implementação do TAD da lista está no arquivo *"lista.c"*.

## 3. Análise de Complexidade

### 3.1 Análise Assintótica

Vamos começar analisando a complexidade das operações da lista. As operações usadas são:

1. Criar a lista: $O(1)$;
2. Inserir na lista: $\Omega(n)$;
   (a) Apesar de inserir ser $O(1)$, teremos que inserir no mínimo $n$ vezes, por isso $\Omega(n)$;
3. Apagar a lista: $O(1)$.

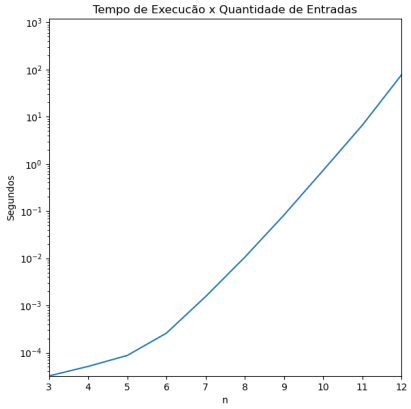
Analisando agora a complexidade das operações da fila. As operações usadas são:

1. Alocar/criar a fila: $O(1)$;

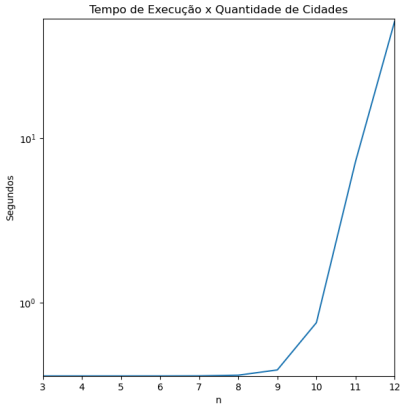
2. Inicializar a fila: $O(1)$;
3. Enfileirar um elemento: $O(1)$;
   (a) Pois enfileiramos sempre no fim da fila;
4. Desenfileirar um elemento: $O(1)$;
   (a) Pois desenfileiramos sempre do início da fila;
5. Checar se a fila está vazia/cheia: $O(1)$;
6. Apagar a fila: $O(n)$;
   (a) Como temos que percorrer cada nó na fila e apagar seu item (e o nó), vamos executar $n$ operações. Logo, a complexidade de apagar a fila é $O(n)$ (supondo que temos $n$ elementos enfileirados).

Podemos perceber que a operação mais custosa é apagar a fila. Em nossa implementação, essa operação é executada toda vez que geramos uma permutação. Como temos $n!$ permutações, executamos a operação de apagar a fila $n!$ vezes, e como essa operação tem custo $O(n)$, temos uma complexidade de $O(n \times n!)$. Vamos analisar as operações da nossa implementação da solução caso haja uma complexidade maior que a encontrada.

1. Ler dados: $\Omega(n)$;
2. Verificar se o elemento a ser inserido na permutação já está presente: $O(n)$;
3. Alocar uma permutação: $O(1)$;
4. Verificar se a permutação encontrada é o melhor caminho: $O(n)$;
   (a) Verificar se o elemento está presente: $O(n)$;
   (b) Imprimir o melhor caminho: $O(n)$;
   (c) Calcular a distância: $O(n)$;
5. Gerar as permutações: $O(n \times n!)$:
   (a) Alocar uma permutação: $O(1)$;
   (b) Inicializar uma fila: $O(1)$;
   (c) Enfileirar na fila: $O(n \times n!)$;
      i. Pensamos aqui no pior caso, em que enfileiramos todos os elementos a cada permutação, ou seja, temos que executar a operação enfileirar $n \times n!$;
   (d) Apagar a fila: $O(n \times n!)$;




(a) Gráfico 1




(b) Gráfico 2

Figura 1: Gráficos 1 e 2, Tempo de Execução x Quantidade de Cidades

| 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|
| 0.000032 | 0.000051 | 0.000087 | 0.000259 | 0.001541 |
| 8 | 9 | 10 | 11 | 12 |
| 0.010502 | 0.082572 | 0.730486 | 6.631656 | 77.403269 |

Tabela 1: Quantidade de Cidades x Tempo de Execução (em segundos)

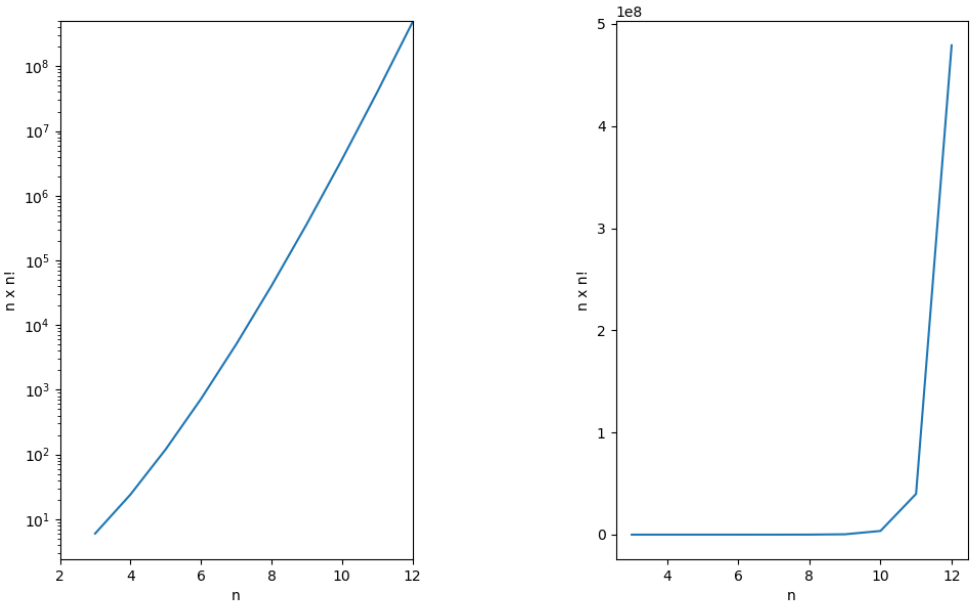
6. Apagar o vetor em que estão armazenados os dados de entrada e a fila com o melhor caminho: $O(1)$.

Temos, portanto, que a complexidade da nossa solução é $O(n \times n!)$, já que na notação *big O* consideramos como limite superior o processo representado pela função com maior taxa de crescimento.

### 3.2 Tempo de Execução

Medimos o tempo de execução para cada n (utilizando a biblioteca *"time.h"*) três vezes e utilizamos a média desses tempos para criar os gráficos abaixo, cada qual com o eixo y (segundos) em uma escala diferente.

Devido à natureza da notação *big O* (isto é, ela desconsidera constantes que multiplicam/somam à função, além de considerar somente a função com maior taxa de crescimento), fator esse somado a atributos inerentes ao computador que roda o código, o formato do gráfico não é idêntico à função $y = n \times n!$. Podemos, contudo, afirmar que seu comportamento é similar, ou seja, que a análise assintótica condiz com a realidade.




(a) Gráfico 3

(b) Gráfico 4

Figura 2: Gráficos 3 e 4, $y = n \times n!$

## 4. Solução Otimizada

### 4.1 Modelagem da Solução

Otimizamos o algoritmo da Força Bruta utilizando os conceitos de Programação Dinâmica e Matriz de Adjacência. Uma Matriz de Adjacência é uma forma de representar um grafo e suas conexões em uma matriz (se o grafo tem $n$ elementos, a matriz terá formato $n \times n$). O conceito de Programação Dinâmica é utilizado pois iremos computar a solução para o caminho de tamanho $N$ utilizando soluções já calculadas para um caminho de tamanho $N - 1$. Essa solução consiste de:

1. Ler as entradas, como na solução anterior;
2. Guardar as informações coletadas (os pares de cidades a distância entre elas) em uma matriz de adjacência;
   (a) Sendo cada cidade um inteiro ($A$ e $B$, por exemplo), preenchemos as posições $[A][B]$ e $[B][A]$ com a distância direta entre essas duas cidades (sendo a distância também um inteiro);
   (b) Caso não haja uma conexão direta entre duas cidades, preenchemos suas posições na matriz com um valor inválido (0, por exemplo);
3. Calculamos todas as distâncias possíveis considerando a cidade inicial e as cidades com quem ela tem uma conexão direta (ou seja, resolvemos o problema para um percurso de tamanho $N = 2$);
   (a) Guardamos, nesse passo, *o conjunto de nós que visitamos* (representado em nossa implementação por um vetor) e *o índice do último nó visitado*;
4. Adicionamos então outro nó (coerente com a matriz de adjacência) a esta solução e utilizamos as respostas anteriormente obtidas para calcular o novo percurso com a menor distância;
   (a) Fazemos isso utilizando chamadas recursivas;
   (b) A cada chamada, verificamos se visitamos todas as cidades (e se as conexões são válidas);
   (c) Se ainda não visitamos todas as cidades, chamamos essa função recursivamente;
   (d) Caso contrário, temos nosso percurso de menor distância;
5. Retornamos a distância e o caminho encontrados.

Note que usamos a Estrutura de Dados **pilha** (mais especificamente, uma pilha encadeada) para armazenar o melhor caminho, pelo mesmo motivo que escolhemos uma fila na solução anterior.

### 4.2 Análise Assintótica

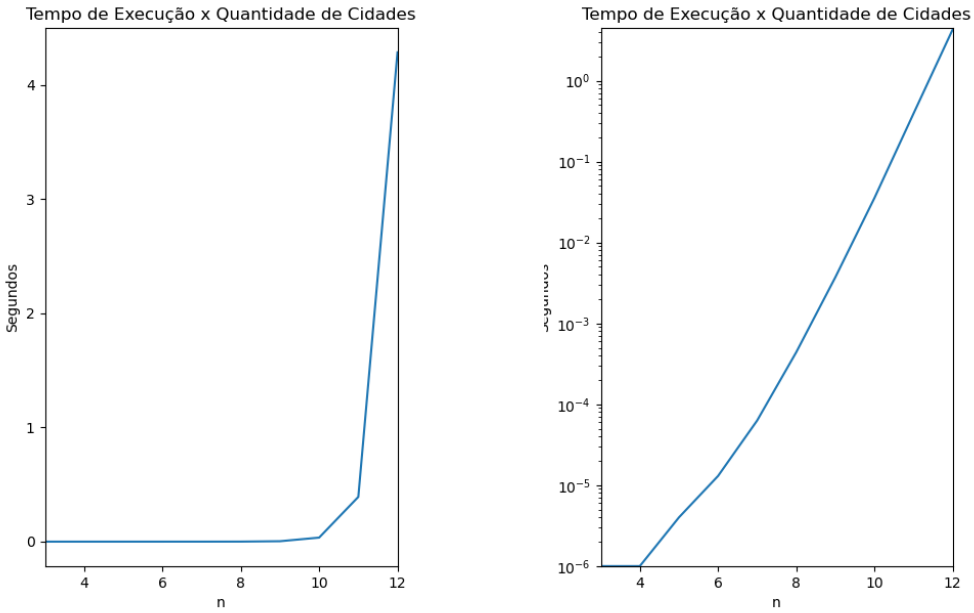
Temos que a complexidade das operações da pilha é $O(1)$ para toda função, com exceção de apagar a pilha, que tem complexidade $O(n)$ (assumindo $n$ elementos nela) pelo mesmo motivo que a fila. Vamos analisar cada função da nossa implementação da solução otimizada para descobrirmos a complexidade dela (assumindo $n$ cidades e o pior caso).

1. Encontrar o menor número entre dois inteiros: $O(1)$;
2. Alocar e inicializar a matriz de adjacência: $O(n^2)$;
3. Ler os dados: $\omega(n)$;
4. Verificar se todas as cidades foram visitadas: $O(n)$;
5. Criar e inicializar o vetor para verificar as cidades visitadas: $O(n)$;
6. Criar o vetor auxiliar para armazenarmos o caminho atual: $O(1)$;
7. Calcular a melhor rota: $O(n \times n!)$;
   (a) Visitar as cidades não visitadas: $O(n!)$, pois precisamos checar todas as permutações possíveis de caminhos (estamos fazendo $n!$ chamadas recursivas);
   (b) Verificar se todas as cidades foram visitadas: $O(n)$, pois temos que percorrer o vetor de tamanho $n$;
   (c) Como chamamos a função recursivamente $n!$ vezes, e cada chamada tem custo $O(n)$, a complexidade total desse algoritmo é $O(n \times n!)$;
8. Liberar a memória: $O(n)$;
   (a) Apagar a pilha: $O(n)$.

Temos, portanto, que a complexidade do algoritmo continua sendo $O(n \times n!)$. A otimização deste código está na simplificação do código e na redução de operações redundantes quando comparado à implementação de força bruta. O uso da Matriz de Adjacência, por exemplo, permite acesso em tempo constante ($O(1)$) à distância entre cidades, enquanto no algoritmo anterior, precisamos percorrer o caminho com as $n$ cidades para achar a distância ($O(n)$). Além disso, ao utilizar um vetor para checarmos se já visitamos uma certa cidade, evitamos revisitar cidades e temos a chance de terminar o programa "cedo", sem precisar gerar permutações desnecessárias ou inválidas.

### 4.3 Tempo de Execução

Medimos o tempo da mesma forma que anteriormente. Note que o formato da função se mantém similar a $y = n \times n!$, o que coincide novamente com a notação *big O*. Perceba, contudo, que ao compararmos os tempos de execução dos dois algoritmos, o algoritmo Otimizado é muito mais rápido que o Força Bruta (para um valor razoável de $n$), o que também condiz com sua implementação.



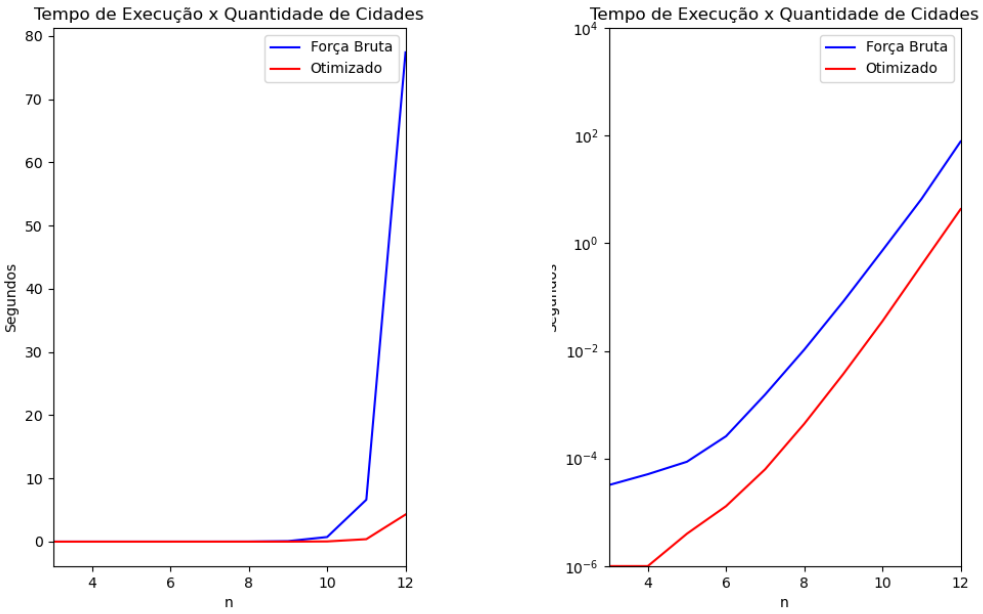

(a) Gráfico 5

(b) Gráfico 6

Figura 3: Gráficos 5 e 6, Tempo de Execução x Quantidade de Cidades

| 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|
| 0.000001 | 0.000001 | 0.000004 | 0.000013 | 0.000063 |
| 8 | 9 | 10 | 11 | 12 |
| 0.000440 | 0.003713 | 0.035417 | 0.391957 | 4.284960 |

Tabela 2: Quantidade de Cidades x Tempo de Execução (em segundos)




(a) Gráfico 7

(b) Gráfico 8

Figura 4: Gráficos 7 e 8, comparação entre o tempo de execução dos dois algoritmos