O Problema do Caixeiro Viajante

Relatório

# Força Bruta

O método de “força bruta” para a resolução do Problema do Caixeiro Viajante consiste em percorrer todos os caminhos possíveis e comparar eles entre si até encontrar o menor caminho possível.

## Estruturas de Dados Escolhidas

No código, duas estruturas de dados principais são utilizadas:

Estrutura GARFO: Esta estrutura representa as conexões entre cidades. Ela contém um array de cidades, e cada cidade possui um array de conexões (arcos) para outras cidades. Cada conexão contém a cidade de destino e a distância.

Estrutura CIDADE: Cada cidade tem um índice único e um array de conexões (CONEXAO), onde são armazenadas as informações sobre os destinos e distâncias para outras cidades.

Estrutura CONEXAO: Modela cada arco entre cidades, contendo a cidade de destino e a distância até ela.

## Vantagens:

**Clareza e Simplicidade**: A modelagem do problema com arrays dinâmicos de conexões para cada cidade torna o código mais fácil de entender, já que as operações envolvem manipulação direta de listas de conexões entre cidades.

**Flexibilidade**: A estrutura permite adicionar quantas conexões forem necessárias entre as cidades sem um limite fixo de arcos, já que as conexões são dinâmicas.

**Eficiência em Busca Local**: O acesso a conexões de uma cidade é eficiente porque o array armazena diretamente essas conexões, facilitando a exploração de destinos durante a busca recursiva.

## Limitações:

**Complexidade de Espaço**: Como o array de conexões é alocado para todas as cidades, pode haver consumo de memória ineficiente, já que a estrutura aloca espaço para todas as possíveis conexões.

**Desempenho com Grandes Números**: Embora a estrutura seja eficiente para pequenos números de cidades e conexões, em maiores o tempo de execução pode aumentar significativamente devido à complexidade da busca recursiva e à exploração de todas as conexões.

## Lógica do código:

### main

É necessário que o usuário insira um número de cidades [num\_cidades], a cidade de partida [comeco] e o número de conexões entre as cidades [num\_conexoes].

Como vamos usar cidades a partir do índice 0, decrescemos 1 da variável começo.

A função garfo\_criar é chamada para alocar e inicializar a estrutura de dados do garfo com o número de cidades especificado,

Após inicializar as cidades, o programa então processa as conexões entre elas, onde o usuário insere duas cidades [a] e [b] e a distância entre elas [dist]. A função [garfo\_add\_connection] é chamado duas vezes para adicionar a conexão de [a] a [b] e de [b] a [a], já que o garfo é não direcionado.

É inicializado um array [visitados] que marca as cidades visitadas durante a busca (inicialmente todas são não visitadas).

Também um array [path] que armazena o caminho encontrado. O primeiro elemento é a cidade de partida (comeco), e os outros são inicializados com -1 para indicar que ainda não foram preenchidos.

E uma variável [best] para armazenar a melhor (menor) distância encontrada, inicializada com um valor alto (1000000).

A função [my\_recursion] é chamada para começar a busca pelo caminho mais curto, passando o garfo, os arrays de controle, a cidade inicial e outras informações necessárias. O resultado (menor distância) é armazenado em [best].

É feito um print do menor caminho encontrado chamando a função recursiva [encontrar\_menor\_caminho] e depois é printado todos os caminhos.

Um loop for imprime o caminho encontrado (ajustando os índices de volta para 1). Depois, a melhor distância é impressa.

A função [garfo\_apagar] é chamada para liberar toda a memória alocada para o garfo, evitando vazamentos de memória.

### encontrar\_menor\_caminho

É iniciado marcando o caminho atual como visitado para que ele não seja percorrido novamente.

As conexões da cidade atual são todas iteradas em um loop baseado no número de conexões da cidade atual.

Se a cidade sendo percorrida ainda não foi visitada, a função chama a si mesma usando de parâmetro a cidade que está sendo percorrida como a atual, a distância como a distância mandada no parâmetro + a distância da conexão atual, e [num\_visitadas] como a própria somado de 1. O resto dos parâmetros se mantém.

Se depois do loop o atual melhor [best] for menor que o antigo melhor [prev], a cidade da atual conexão [i] é adicionada no caminho [path] no próximo ponto após a cidade atual.

A condição de parada do loop é se a cidade [i] conectada na cidade atual for igual ao destino, a contagem atual de cidades já visitadas é igual ao número de cidades (subtraído por 1 por conta de o índice ser contado a partir de 0) e a distância total é menor do que a atual melhor.

Quando isso acontece a função registra que a cidade de destino já foi alcançada e atualiza o valor da variável [best] com a nova distância mínima encontrada, que é a distância total até o destino por essa rota, e o loop é encerrado.

# Resolução otimizada

A solução seguinte foi pensada para resolver o Problema do Caixeiro Viajante de forma otimizada. Foi usada programação dinâmica e bitmasking, uma técnica que representará os estados das cidades em 1s e 0s, para fazer essa resolução, que ainda é exata, mas que é mais eficaz do que a abordagem de bruteforcing. A programação dinâmica diminui os custos reutilizando resultados intermediários para evitar redundâncias e reduzir o número de combinações a serem exploradas.

### main

É necessário que o usuário insira um número de cidades [num\_cidades], a cidade de partida [comeco] e o número de conexões entre as cidades [num\_conexoes].

Como vamos usar cidades a partir do índice 0, decrescemos 1 da variável começo.

É inicializada um ponteiro para uma struct [DISTANCIAS] de nome [memorizacao] de dimensões [num\_cidades][2^num\_cidades], que será responsável por armazenar a menor distância para visitar certos subconjuntos de cidades. Um índice [i][j] registra a menor distância para percorrer o subconjunto [j] de cidades a partir da cidade [i]. Em bitmasking, se [j] = 5, significa o subconjunto [0101], ou seja, cidades 0 e 2 incluídas no subconjunto, mas não a 1 e a 3.

É inicializada um ponteiro para uma struct [DISTANCIAS] de nome [ligacoes] de dimensões [num\_cidades][num\_cidades], que armazenará as distâncias diretas entre as cidades. Um índice [i][j] registra a distânia da cidade [i] para a cidade [j].

Então é chamado a função [inicializar\_distancias] que irá inicializar [ligacoes] e [memorizacao], configurando as distâncias entre as cidades e preparando a estrutura para o algoritmo de programação dinâmica.

Logo em seguida, a função [calcular\_distancias\_minimas], que vai calcular as distâncias mínimas entre as cidades considerando todas as combinações de cidades que podem ser visitadas usando programação dinâmica, armazenando as distâncias intermediárias na [memorização].

Baseado nos caminhos armazenados [memorização] pela [calcular\_distancias\_minimas], é chamada a função [encontrar\_menor\_custo] que irá buscar o menor custo total [memorização] e armazená-lo na função [menor\_distancia].

Baseado no trabalho feito pelas duas funções anteriores, será executada a função [reconstruir\_melhor\_rota], que usa a [memorizacao] para reconstruir o caminho exato correspondente ao menor custo encontrado por [encontrar\_menor\_custo] e isso será armazenado na lista [menor\_rota].

É iterado um laço for imprimindo cada cidade da lista [menor\_rota] acrescentado de 1 para fazer sentido com a indexamento das cidades, que representa a rota de menor custo.

Depois, é imprimido a distância total da rota de menor custo [menor\_distancia].

Por fim, as variáveis são desalocadas.

### reconstruir\_melhor\_rota

Começa inicializando uma lista [caminho] que irá armazenar a sequência que forma a melhor rota, um *int* [estado] que guarda em *bitmask* que todas as cidades foram visitadas e inicializa um *int* [cidade\_anterior] que irá armazenar o índice da última cidade visitada começando com a cidade inicial.

É então iterado um loop externo que começa na última cidade e irá indo de forma decrescente até a segunda. É ao contrário pois temos que reconstruir o caminho de trás para frente, e a cada iteração determinamos qual foi a cidade anterior na rota.

É inicializada uma variável [melhor\_cidade] que irá armazenar a possível cidade anterior no caminho mais curto. É no começo -1, que é um índice de cidade nulo.

E então um loop interno [cidade\_atual] que percorre todas as cidades e tenta encontrar a cidade que pode ter sido a última cidade antes da cidade atual.

As condições de continuação são se o índice do loop interno for o [comeco], já que ela não pode ser uma cidade intermediária ou se o índice não estiver no subconjunto de cidades não visitadas.

O primeiro índice que passa pelas condições acima se torna o primeiro candidato para a cidade anterior no caminho. Isso acontece apenas na primeira execução do loop interno, quando o valor de [melhor\_cidade] é -1.

O algoritmo então inicializa uma variável [distancia\_previa] que armazena a distância total que passaria pela cidade em [melhor\_cidade] e uma variável [distancia\_atual], com a distância total que passaria pela cidade com o índice atual iterado no loop interno.

Se a variável [distancia\_atual] for menor que a [distancia\_previa], a variável [melhor\_cidade] é atualizada para a [cidade\_atual].

Após a comparação, o índice da atual melhor cidade se torna o [cidade\_anterior] e é armazenado no índice externo da lista de cidades [caminho]. E então, ela é removida do sobconjunto de cidades a serem consideradas.

No final, é colocado a cidade inicial no primeiro e no último índice da lista [caminho] para garantir o ciclo do viajante começar e terminar no mesmo lugar e completa o percurso. E então é retornado a lista.

### encontrar\_menor\_custo