**SCC0202 - Algoritmos e Estruturas de Dados I**

**Relatório - Projeto 1**

**Alunos NUSP**

Pedro Henrique de Sousa Prestes 15507819

Dante Brito Lourenço 15447326

João Gabriel Pieroli da Silva 15678578

**Introdução**

No projeto em questão, da disciplina de Algoritmos e Estruturas de Dados I, ministrada pelo docente Dr. Rudinei Goularte, do Instituto de Ciências Matemáticas e de Computação (ICMC), foi proposta a construção de um algoritmo para resolução do Problema do Caixeiro Viajante (PCV), a partir das estruturas de dados vistas em sala de aula.

O Problema do Caixeiro Viajante (PCV) é um clássico na área de Computação, envolvendo a necessidade de um representante comercial visitar várias cidades e retornar à cidade de origem. O desafio reside em determinar o trajeto mais eficiente que permita visitar cada cidade exatamente uma vez, minimizando o custo total da viagem.

No contexto do presente trabalho, o custo é definido em termos de distância percorrida, considerando que o usuário fornece a quantidade de cidades, a cidade de origem, a quantidade de arestas do grafo, formado pelas cidades como vértices, de modo que nem todo caso formasse um grafo completo, isto é, nem todas as cidades estivessem diretamente conectadas.

**Parte I - Modelagem da Solução**

A fim de modelar o problema, o grupo optou, dentre as estruturas de dados estudadas em sala de aula, por utilizar as estruturas de TAD:

* **ITEM:** 
  + A estrutura ITEM possui dois campos: um inteiro e um ponteiro para void.
  + As funções do TAD ITEM “item.h” são as seguintes:
    - Criar um item = Complexidade: O(1);
    - Apagar um item = Complexidade: O(1);
    - Atribuir uma chave = Complexidade: O(1);
    - Receber a chave do item = Complexidade: O(1);
    - Receber o dado do item = Complexidade: O(1);
  + A vantagem desse TAD está atrelada ao fato de ser uma estrutura básica que encapsula um inteiro chave e um dado genérico, o qual permite armazenar qualquer tipo de dado. Isso torna a manipulação de itens em algoritmos, como os de ordenação ou busca, mais organizada e a complexidade de todas as suas funções são O(1).
  + Uma limitação desse TAD é a de que, muitas vezes, não se faz uso de ambos os campos da struct. Em vários casos, basta que se utilize a chave, então, é alocada memória para o ponteiro void sem necessidade.
* **DEQUE:**
  + A estrutura DEQUE, por ter sido implementada com caráter sequencial e circular, possui quatro campos: um vetor de itens de tamanho definido pela constante MAX\_TAMANHO, um inteiro que indica a quantidade de itens nesse vetor, um inteiro que indica a posição inicial do vetor, um inteiro que indica a próxima posição vaga no vetor, logo após o último elemento do deque.
  + As funções do TAD DEQUE “deque.h” são as seguintes:
    - Criar um deque = Complexidade: O(1);
    - Apagar um deque = Complexidade: O(n);
    - Inserção no início = Complexidade: O(1);
    - Inserção no fim = Complexidade: O(1);
    - Remoção no início e retorno do item retirado = Complexidade: O(1);
    - Remoção no fim e retorno do item retirado = Complexidade: O(1);
    - Retornar ao usuário o primeiro item = Complexidade: O(1);
    - Retornar ao usuário o último item = Complexidade: O(1);
    - Retorno da quantidade de elementos do deque = Complexidade: O(1);
    - Verificação se o deque está vazio = Complexidade: O(1);
    - Verificação se o deque está cheio = Complexidade: O(1);
    - Copiar as informações de um deque em outro = Complexidade: O(n);
  + A vantagem desse TAD está atrelada ao fato de que, quando aplicado de forma circular, torna-se uma estrutura que permite fazer inserções e remoções em um nível de complexidade O(1). Além disso, o acesso aos itens das extremidades também são proporcionados em tempo constante.
  + Uma limitação desse TAD é a de que não se consegue acessar os itens diretamente pelo índice, de modo que só é possível assim fazê-lo por meio de remoções em uma extremidade e inserção do dado retornado na outra extremidade. Além disso, por ser um deque sequencial, há um limite quanto ao tamanho do vetor de itens, seja para valores que oscilam entre poucas e grandes quantias de dados, seja para um caso em que não há um limite determinado para o número de elementos, o deque sequencial não se torna mais tão viável.
* **LISTA:**
  + A estrutura LISTA, por ter sido implementada com caráter sequencial e ordenado, possui quatro campos: um vetor de itens de tamanho definido pela constante TAM\_MAX, um inteiro que indica a quantidade de itens nesse vetor, um inteiro que indica a posição inicial do vetor e um inteiro que indica a próxima posição vaga no vetor, logo após o último elemento do deque.
  + As funções do TAD LISTA “lista.h” são as seguintes:
    - Criar uma lista = Complexidade: O(1);
    - Apagar uma lista = Complexidade: O(n);
    - Inserir um item na lista na lista ordenada = Complexidade: O(n);
    - Remover um item da lista = Complexidade: O(n);
    - Retornar o item de uma chave específica na lista ordenada = Complexidade: O(log n);
    - Retorno da quantidade de elementos da lista = Complexidade: O(1);
    - Verificação se a lista está cheia = Complexidade: O(1);
    - Verificação se a lista está vazia = Complexidade: O(1);
  + A vantagem desse TAD está associada à flexibilidade para inserção e remoção de itens em qualquer posição. Ainda, quando implementada de forma sequencial, a operação de busca é mais eficiente com o algoritmo da busca binária.
  + As limitações desse TAD estão relacionadas às inserções e às remoções, sobretudo, de itens dos primeiros índices do vetor de itens, afinal, para esse caso, é preciso que o vetor todo se desloque. Além disso, por ser uma lista sequencial, há um limite quanto ao tamanho, seja para valores que oscilam entre poucas e grandes quantias de dados, seja para um caso sem um limite determinado para o número de elementos, a lista sequencial não se torna mais tão viável.

Essas foram as estruturas de dados escolhidas para a resolução do problema, tendo em vista que: como o problema envolve fazer permutações, organizar distâncias, elas foram escolhidas por características específicas que contribuem para a eficiência e organização do algoritmo, principalmente, considerando as operações frequentes de inserção, remoção, busca, e armazenamento temporário de dados ao longo das permutações necessárias para encontrar a solução.

Sendo assim, reiterando as vantagens, o **ITEM** flexibiliza trabalhar com uma chave e um dado genérico, útil para algoritmos de busca, os quais são fundamentais na avaliação de diferentes trajetos no PCV, e armazenar arestas. Já o **DEQUE** facilita a manipulação de trajetos de ida e volta, imprescindíveis no PCV, e o tempo constante nas operações de extremidades agiliza o processo de cálculo das distâncias e construção dos trajetos, já que consegue ser utilizado para a permutação das cidades. Enquanto isso, a **LISTA** viabiliza o acesso a um item da lista com uma chave específica, dado que é feito com busca binária, em complexidade O(log n), assim, torna-se viável para percorrer a lista de adjacências, utilizada para armazenar as conexões entre as cidades (arestas) e seus respectivos custos.

**Parte II - Implementação**

Para o algoritmo da solução do PCV, que envolve fazer permutações e calcular os n! caminhos possíveis, utilizou-se da estrutura de dados DEQUE, visto que se consegue inserir elementos nas extremidades, sem aumentar a complexidade do algoritmo. Então, para tanto, a **função *caixeiroSolver*** - que detém a lógica da operação do caixeiro - faz uso de duas struct **CAMINHO**, que armazenam um deque representando a rota daquele caminho e um inteiro distância, e um deque auxiliar para permutações. O caminho global armazena o melhor caminho encontrado até então, enquanto o caminho local permuta diferentes trajetos a fim de alcançar todas as combinações de caminhos possíveis, já o deque auxiliar guarda todas as cidades que devem ser visitadas, desconsiderando a cidade em que se inicia. A permutação é feita entre os deques local e auxiliar, manipulando as extremidades de maneira recursiva para calcular todas as formas de caminho que podem ser feitas, desconsiderando os caminhos impossíveis.

Além disso, cria-se uma lista de adjacências, de maneira que acessa as informações da cidade ‘i’, armazena todas as distâncias fornecidas pelo usuário no campo 'dado' de cada item e, na 'chave', as respectivas cidades ‘j’ a que estão ligadas, tais distâncias são acessadas posteriormente pela **função *calculaDistancia***, verificando se a aresta entre as duas cidades existe (o item de uma determinada chave, representando uma das cidade, existe dentro da lista de adjacência da outra cidade) e, caso exista, retorna a distância para a *caixeiroSolver*. Assim, é calculada a distância total do percurso para cada nova variação do deque local.

* **Arquivos**
  + **Pasta Força\_Bruta**
    - “item.h/item.c” = TAD item;
    - “lista.h/lista.c” = TAD lista;
    - “deque.h/deque.c” = TAD deque;
    - “caixeiro.c” = Programa principal;
    - “Makefile” = Contém as funções “all”, “clean”, “run” e “zip”;
  + **Pasta Held-Harp**
    - “item.h/item.c” = TAD item;
    - “lista.h/lista.c” = TAD lista;
    - “deque.h/deque.c” = TAD deque;
    - “main.c” = Programa principal;
    - “Makefile” = Contém as funções “all”, “clean”, “run” e “zip”;

**Parte III - Análise de Complexidade**

O arquivo principal da resolução do problema tem duas funções principais para cálculo da menor distância:

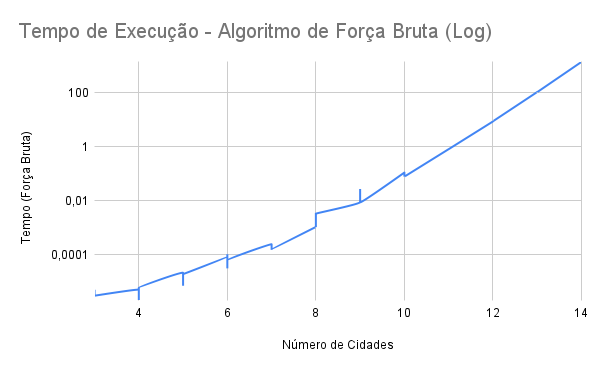
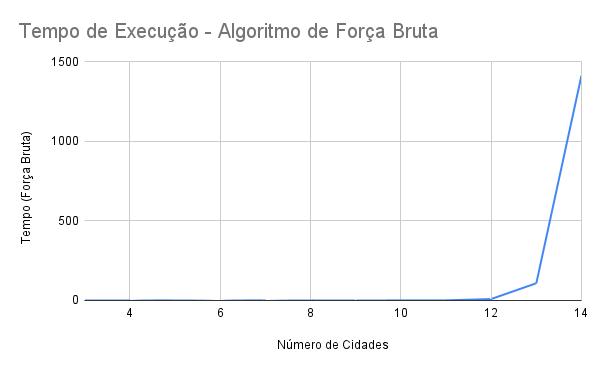
* ***calculaDistancia*:** Chama a função de *lista\_item O(log n)* para verificar se a aresta existe e retornar sua distância. É importante ressaltar que a escolha de uma lista sequencial foi crucial para a redução da complexidade, pois apesar de ser necessário fazer shifts ao inserir, o volume de acessos é muito maior e pode ser otimizado utilizando a busca binária. Complexidade: ***O(log n)****.*
* ***caixeiroSolver:*** Faz as permutações de todas as distâncias utilizando remoção e inserção em deques *O(1)*, fazendo chamadas da função *calculaDistancia* *O(log n)* e calculando todas as permutações de cidade, desconsiderando a cidade de origem/destino, pois ela é fixa: *O((n-1!))\**. Assim, a complexidade total é ***O((n-1)! log n)).***

***\**** *OBS.****:*** Pode ser descrito como ***O(n!)***, já que são equivalentes em notação assintótica. Foi utilizado n-1 para deixar claro que houve desconsideração da cidade de origem na permutação, nesse caso a permutação total é de ***O(n! log n)***.

*Outras funções menores para resolução incluem:*

* ***entrada\_de\_dados:*** *Faz a leitura de todos os dados de entrada. A operação mais significativa realizada é a leitura da quantidade de arestas. No pior caso, quando o grafo é completo, a quantidade de arestas pode ser descrita por ((((n-3)\*n)/2)+n). Nesse sentido, a complexidade total é* ***O(n^2).***
* ***caminho\_criar:*** *Inicializa a struct CAMINHO. Complexidade:* ***O(1)*** *.*

Complexidade dominante da resolução: ***O((n-1)! log n)).***

**Gráficos**

Como é possível constatar no Gráfico 1 o Algoritmo de Força Bruta apresenta um tempo de execução que aumenta gritantemente conforme o número de cidades cresce, o que está alinhado com a sua complexidade fatorial. Esse comportamento é evidente na curva, especialmente após o número de cidades superar 12, onde o tempo de execução cresce drasticamente, corroborando a explosão combinatória típica do problema e tornando um algoritmo inviável para solucionar o PCV para um número de cidades superior a 14 em um curto período de tempo. Assim, o segundo gráfico com o eixo vertical em escala logarítmica denota melhor o crescimento da função.

**Complexidade de espaço:**

***DEQUE:*** A memória necessária para um deque depende de quantos elementos ele armazena. Se houver n cidades, o deque contém no máximo n-1 cidades. Portanto, a memória usada pelo deque será proporcional a O(n).

***LISTA:*** a lista de adjacência vai precisar de O(n^2) memória para armazenar todas as arestas, tendo em vista que para um grafo completo o número de arestas é dado por: *((((n-3)\*n)/2)+n).*

Complexidade espacial dominante da resolução: ***O(n^2).***

**Conclusão**

A análise do algoritmo de Força Bruta demonstrou sua ineficácia para problemas de grande escala, devido à explosão combinatória com crescimento fatorial, como observado nos experimentos. O tempo de execução cresce rapidamente, tornando-o inviável para um número maior de cidades.

As estruturas de dados escolhidas contribuíram de maneira decisiva para a eficiência dos algoritmos. O DEQUE foi essencial para a manipulação eficiente dos trajetos e permutações, enquanto a LISTA facilitou o acesso às adjacências com busca binária, otimizando o cálculo das distâncias entre as cidades.

**Caixeiro Viajante - Utilizando Programação Dinâmica e Bitmask (Held Karp)**

**Estruturas de Dados Escolhidas:**

**Matriz *dp[n][2^n]*:** Armazena o menor custo para chegar à cidade ***i*** com uma máscara de bits representando as cidades visitadas no trajeto até ***i***.

O uso desta matriz permite armazenar o custo mínimo associado a cada combinação de cidades visitadas, facilitando a recuperação da solução ótima, sem recalcular subproblemas. Optou-se por usar uma matriz pois se pode realizar acesso a índice em ***O(1)***, além de que é adequada para a técnica de bitmasking.

Contudo, estrutura tem um alto custo de memória, já que ocupa ***O(n \* 2^n),*** onde n é o número de cidades. Além disso, por utilizar uma bitmask, que representa todas as combinações de vértices, quanto menos denso for o grafo, mais ineficiente será a estrutura, pois irá alocar memória para vários trajetos inválidos que nem serão computados.

**Deque:** Utilizada para armazenar o caminho atual, permitindo inserção e remoção eficiente das cidades.

A *deque* permite realizar a operação de inserção e remoção de elementos nas extremidades de maneira eficiente, com complexidade ***O(1)***, o que é útil para manipular o caminho das cidades. Não é ideal para manipulações mais complexas, como buscas, o que não é necessário no problema dado.

**Lista de Adjacências:** Utilizada para armazenar as conexões entre as cidades (arestas) e seus respectivos custos. Foi utilizada uma implementação sequencial circular.

Essa estrutura economiza espaço em comparação a uma matriz de adjacência, especialmente em grafos esparsos, e oferece complexidade eficiente de ***O(1)*** para acessar as conexões diretas de uma cidade.

Obs: conforme conversado com o professor, ele preferiu que, ao invés de acessar os índices da lista diretamente, fosse usada a operação de busca para acessar as adjacências, visando à aplicação dos conteúdos aprendidos em aula. Como se utiliza busca binária, cada iteração no percorrimento das adjacências possui complexidade ***O(log(n))***.

Por se tratar de uma lista sequencial, a inserção e a remoção no meio da lista possuem complexidade O(n), mas como não se está utilizando esses recursos, essa implementação provou ser a ideal.

**Lógica Utilizada para Resolver o Problema:**

O algoritmo escolhido utiliza programação dinâmica com bitmasking. A ideia é calcular o menor custo para percorrer todas as cidades, armazenando os resultados em subproblemas (máscaras de bits que representam o conjunto de cidades já visitadas). Foi escolhido construir a solução de forma iterativa e incremental, partindo da cidade inicial e expandindo para as próximas cidades, enquanto se minimiza o custo total.

A matriz ***dp[i][bm]*** representa o menor custo para visitar todas as cidades do conjunto representado por ‘*bm’* (bitmask) e terminar na cidade *i*. Essa melhoria em relação ao algoritmo de força bruta se baseia no fato de que este método calcula, utilizando tabulação *(bottom-up dynamic programming),* apenas os subconjuntos da cidade, para cada último representante, ao invés de todas as permutações, evitando cálculos certamente desnecessários.

Exemplo: se foi calculado que o caminho 1->2->3->4 tem peso 32, e o caminho 1->3->2->4 possui peso 21, é necessário guardar apenas o segundo caminho para gerar outros subconjuntos que utilizam os mesmos vértices e continuam a construção a partir do 4, tendo em vista que se busca minimizar o custo total.

**Complexidade Assintótica (Big O):**

**Complexidade de tempo:**

O número de subproblemas é *n\*2^n* (onde n é o número de cidades, e *2^n* é o número de subconjuntos possíveis).

Para cada subproblema, o algoritmo percorre todas as *n* cidades para avaliar o próximo possível destino e, considerando que é verificado se o próximo elemento está na lista de adjacências com busca binária, de complexidade ***O(log(n))***, o resultado é uma complexidade final de ***O(n^2 \* 2^n \* log(n))***.

**Complexidade de espaço:**

A matriz dp tem tamanho ***O(n \* 2^n)***, o que pode se tornar um problema para valores elevados de n devido ao alto consumo de memória.. Contudo, o maior consumo de memória se dá na matriz de deques *paths*, em que cada deque *paths[i][bm]* armazena o caminho percorrido para alcançar o valor de *dp[i][bm]*. Considerando que cada deque tem no máximo tamanho n, a complexidade espacial de paths é de ***O(n^2 \* 2^n)***.

Além disso, as estruturas auxiliares, como o deque e a lista de adjacência, aumentam o uso de memória, mas não afetam a complexidade dominante.

Essa análise mostra que o algoritmo Held-Karp, embora seja muito mais eficiente que força bruta, e desempenhe bem para valores pequenos de n, se torna impraticável para valores muito altos devido ao crescimento exponencial.