Documentação do Projeto Caixeiro Viajante

Alexsander Justa - 12559444 - 2021201
Arthur Santorum Lorenzetto - 12559465 - 2021201
Lucas Ivars Cadima Ciziks - 12559472 - 2021201

1. Modelagem da Solução

- Estruturas de Dados

Para resolver o problema do caixeiro viajante proposto no projeto, implementamos a estrutura de dados Lista Simplesmente Encadeada Não Ordenada. A lista simplesmente encadeada **Caminhos** foi utilizada para encontrar a melhor trajetória dentre as opções de viagem, pois permite manusear de forma recursiva as informações sobre o caminho, organizadas em um tipo abstrato de dados (TAD), facilitando, assim, o desenvolvimento da lógica quando comparado à solução iterativa. Além disso, como não sabemos a quantidade de caminhos que serão armazenados dentro da lista em tempo de compilação, a organização de memória **encadeada** é a melhor alternativa, pois permite alocar memória dinamicamente em tempo de execução.

A outra lista **Conexões** foi escolhida para armazenar as informações inseridas pela entrada padrão sobre as conexões entre as cidades, que assim como o caminho, são organizadas por meio de um TAD. Como a quantidade de conexões a serem inseridas é indefinida em tempo de compilação, a lista encadeada é a melhor opção pois permite armazenar os dados de maneira dinâmica e de fácil acesso pelas duas extremidades. A fim de diminuir a complexidade do projeto, optou-se pela lista simplesmente encadeada, pois não há necessidade em navegar nos dois sentidos da estrutura para nenhum dos casos.

- Lógica da Solução

A primeira parte do programa é responsável por armazenar na lista encadeada as conexões de entrada, organizadas pelo TAD Conexão, o qual contém as **duas cidades** envolvidas na conexão e a sua respectiva **distância**. Além disso, a lista de caminhos é inicializada, sendo que o TAD Caminho é responsável por armazenar a **cidade atual do viajante** e a **distância acumulada** até então em determinada trajetória. É importante frisar que o TAD Conexão é utilizado para armazenar os

“pesos” (que nesse problema são distâncias) entre cada cidade, apenas para referência nos cálculos posteriores, enquanto o TAD Caminho serve ao propósito de armazenar a cidade em que o viajante está naquele momento e a distância já acumulada até aquela cidade em um caminho específico.

Após essas inicializações, o programa recupera a cidade de origem da viagem, pressupondo que é sempre a cidade de número 1, uma vez que é arbitrário e não altera o resultado. Com isso, é iniciada a etapa recursiva do algoritmo, em que de fato encontra-se a melhor trajetória de viagem. A condição de parada dessa função recursiva é disparada quando o viajante já passou por todas as cidades, ou seja, apresenta uma distância total maior que zero e no final da lista Caminhos está novamente a cidade de início da viagem. Assim, se a distância do novo caminho encontrado for menor que a anteriormente estabelecida pelo melhor caminho, atualiza-se o melhor caminho.

Para se encontrar um caminho válido, isto é, uma trajetória que passa por todas as cidades e retorna à cidade de origem, foi utilizado uma iteração pelas conexões que possuem alguma relação com a cidade em que o viajante está no momento, para que assim fossem testados apenas os caminhos possíveis, estabelecendo, desse modo, um **algoritmo de força bruta** conforme requisitado pelo projeto. Há 4 tipos de caminhos que são válidos e inseridos na lista de Caminhos, somando-se sempre a distância acumulada até então:

- Se a cidade atual do viajante é **a origem** de uma conexão e o seu destino ainda não foi visitado/incorporado à lista;
- Se a cidade atual do viajante é **o destino** de uma conexão e o sua origem ainda não foi visitada/incorporada à lista;
- Se a origem da conexão analisada é a cidade de início da viagem;
- Se o destino da conexão analisada é a cidade de início da viagem.

Desse modo, a lista de Caminhos é uma estrutura auxiliar que armazena os caminhos válidos para que no fim da recursão a sua distância total seja comparada com a distância do melhor caminho, chegando a uma trajetória única com a menor distância possível.

Por fim, o resultado é impresso conforme detalhado no projeto, com a cidade de início escolhida, a melhor trajetória e sua respectiva distância. Posteriormente, a memória utilizada é totalmente desalocada.

2. Implementação

Dentro da raíz do projeto, há um arquivo Makefile em que é possível gerar e rodar os arquivos binários e executáveis do código com os seguintes comandos:

```
make
make run
```

Nossa solução também pode ser encontrada no seguinte repositório do Github. Para acessá-la, basta clonar localmente o repositório:

```
git clone https://github.com/Cizika/projeto-caixeiro-viajante.git
```

3. Análise de Complexidade

Avaliando a complexidade das operações básicas nas estruturas de dados utilizadas e do algoritmo para encontrar o menor trajeto para o PCV, concluiu-se através da análise assintótica que o custo da solução é dado por $54(n^3) + 121(n^2) - 207(n) - 190$, sendo n o número de cidades da entrada. Vale destacar que a análise foi realizada com base no pior caso, desconsiderando a análise de recorrência.

A complexidade computacional da solução proposta, utilizando-se a notação Big O, é de $O(n^3)$, isto é, uma ordem de crescimento **cúbica**. Ou seja, sempre que n dobra, o tempo de execução é multiplicado por 8, sendo útil para resolver problemas de tamanho relativamente pequeno e explodindo para um número de cidades maior.

O custo de cada linha e função pode ser acessado nos comentários ao lado do código disponibilizado no Github. Abaixo está a complexidade computacional de cada função, separada por TAD:

Operação	**Caminho.c**	**Conexao.c**	**Lista_Caminho.c**	**Lista_Conexao.c**
Criar	O(1)	O(1)	O(1)	O(1)
Apagar	O(1)	O(1)	O(n)	O(n)
Get	O(1)	O(1)	-	O(n)
Inserir	-	-	O(1)	O(1)
Tamanho	-	-	O(1)	O(1)
Vazia	-	-	O(1)	O(1)
Remover	-	-	O(n)	-
Buscar	-	-	O(n)	-
Final	-	-	O(1)	-
Copiar	-	-	O(n)	-
Print	O(1)	-	O(n)	-

Através da função `difftime(t2, t1)` da biblioteca `time.h` obteve-se o tempo de execução para o processamento dos conjuntos de entrada (máximo de n = 12),

desconsiderando tempos de leitura/escrita em arquivos. Os tempos registrados foram coletados em uma máquina com 16 GB de memória RAM.

Número de cidades	Tempo de execução
5	0 segundos
8	0 segundos
10	17 segundos
11	186 segundos
12	2294 segundos

Percebe-se pelo gráfico abaixo que à medida que n aumenta, o tempo de execução também aumenta. Para $n > 10$ o tempo de execução aumenta consideravelmente, sendo que a curva formada se aproxima da função x^3, a mesma ordem de crescimento encontrada na análise assintótica, evidenciando que para entradas maiores o algoritmo acaba sendo pouco eficiente por se tratar de uma solução de força bruta.