Relatório
Projeto 1 - Caixeiro Viajante
Estrutura de Dados 1

Integrantes: Bianca Ângelo da Rocha Barreto - 11832371
Davi Valentino Malaspina Fileti - 12557171
Gabriel Penido de Oliveira - 12558770

Parte I, Modelagem da Solução

Apresentação dos TAD's

A solução encontrada pelo grupo requer dois TAD's diferentes, e os dois serão listas. Abaixo, uma explicação de cada para então ser discutida a solução:

Lista Input

A primeira será a "*lista input*" que é uma lista simplesmente encadeada e não ordenada com o intuito de armazenar os dados inseridos pelo usuário. Portanto, seus itens serão dois inteiros para representar cada "par de cidades", e uma "distância" em float entre elas. Note que ela não precisa ser ordenada pois a ordem é estipulada pelo usuário, então não é uma grande preocupação! A razão por ela ser simplesmente encadeada é que não haverá uma grande movimentação de informações que exigirá uma lista duplamente encadeada, já que ela só servirá para a criação da próxima lista.

Lista Mov

A segunda lista, mencionada anteriormente e nomeada como "*lista mov*" servirá como as regras de um jogo de xadrez, estipulando o "tipo de movimento" que cada cidade faz. Ela registrará quais cidades uma dada cidade pode ir. Isso implica que cada cidade terá sua *lista mov*! Atente-se também para o fato de que cada item da *lista mov* será um int "cidade" e um float "distância". Exemplificando, a cidade X poderá ter uma *lista mov* com 6 itens, ou seja, ela pode ir para 6 diferentes cidades e registrará obrigatoriamente sua respectiva distância para cada uma das 6. Novamente, será uma

lista não ordenada pois não fará diferença pro algoritmo como cada elemento da lista se dispõe. A razão para ela ser simplesmente encadeada é idêntica à *lista input*.

*OBS: Repare que essas listas agem como listas de mercado ou listas de convocação, elas são unicamente para consulta no resto da resolução; Como as regras de um jogo de tabuleiro. Não faria sentido implementá-las esperando movimentação e fluxo intenso de itens, então fizemos elas simplesmente encadeadas.

Discussão da Solução

Para resolver o PCV, pegamos a informação do usuário, armazenamos no *lista input* e "convertemos" o *lista input* no *lista mov* para cada uma das cidades. Com isso em mãos, temos os ingredientes para desenvolver a solução. A ideia é, começando pela cidade de origem, pegar o último elemento da *lista mov* da cidade de origem e analisar sob alguns critérios (discutidos posteriormente) se é possível "ir" para tal cidade. Se sim, analisaremos a *lista mov* dessa nova cidade com o mesmo intuito: Analisar se existe um elemento dessa nova *lista mov* que atende aos critérios para seguirmos adiante. Se os critérios de um elemento falham, ao invés de irmos para a nova lista, tentaremos outros elementos. Caso nenhum elemento da *lista mov* consiga prosseguir, retornamos à cidade anterior e lá testamos um novo elemento de sua *lista mov*. Muito semelhante ao problema do Labirinto, vamos testando caminho por caminho até chegarmos em um caminho que atende às restrições. Quando isso ocorre, esta Lista de cidades que achamos é posta em um vetor, onde passará por uma avaliação de distância, sempre comparando com a rota válida anterior (ou quando for a primeira, será apenas aceita como caminho melhor) para checar qual caminho é de menor custo.

Note que a solução é simples, o importante agora é estabelecer esses critérios para definir se um caminho é válido ou não para ser colocado no vetor e tentar se candidatar a caminho de menor custo. Sabemos do enunciado que nosso camarada Caixeiro-Viajante só pode passar uma única vez por cada cidade e deve ao fim de sua jornada de serviço retornar à cidade inicial. Estipulamos os seguintes critérios para os caminhos candidatos: Eles devem conter todas as cidades, e uma única vez, exceto a primeira, que aparecerá como origem e término de seu trajeto. Outra restrição foi feita discretamente em *lista mov*, mas é a ideia de que uma cidade X não pode ir para qualquer outra cidade, mas apenas para aquelas em sua respectiva *lista mov*. O último critério para determinar a solução é óbvio: O caminho deve cumprir com todos os anteriores requisitos e ter o menor custo de distância. Após analisar com esses "filtros" teremos um único caminho como resposta e imprimimos ele.

Parte 2, Implementação

Dentro da pasta raiz existem as subpastas "*functions*" (com as funções do programa), "*casos_testes*" (com os casos a serem testados) e os TAD's que foram utilizados. Além disso, temos os arquivos "*main.c*" e o "*makefile*" abaixo:

```
CFLAGS = -std=c99 -pedantic-errors -Wall -lm
all: functions.o lista_info.o it_info.o lista_mov.o it_mov.o main.o
    gcc functions.o lista_info.o it_info.o lista_mov.o it_mov.o main.o -o run $(CFLAGS);./run
functions.o:
    gcc -c ./functions/functions.c $(CFLAGS) -I/functions -o functions.o
main.o:
    gcc -c main.c $(CFLAGS) -o main.o
lista_info.o:
    gcc -c ./lista_info/lista_info.c $(CFLAGS) -I/lista_info -o lista_info.o
it_info.o:
    gcc -c ./lista_info/it_info/it_info.c $(CFLAGS) -I/lista_info/it_info -o it_info.o
lista_mov.o:
    gcc -c ./lista_mov/lista_mov.c $(CFLAGS) -I/lista_mov -o lista_mov.o
it_mov.o:
    gcc -c ./lista_mov/it_mov/it_mov.c $(CFLAGS) -I/lista_mov/it_mov -o it_mov.o
run:
    ./run
clean:
    rm *.o *.zip run
```

Para rodar o código basta estar no diretório da pasta (dentro do terminal) e digitar "*make*". Tenha certeza que no *main.c*, o arquivo *.txt* está com o nome correspondente ao input desejado.

Parte 3, Análise de Complexidade

Dados os códigos da parte 2, faremos agora uma análise assintótica do desempenho do programa em seus TAD's e *main.c* em notação *big O*.

it_info.c: todas as funções deste arquivo são $O(1)$;

lista_info.c: com exceção de "*lista_info_busca*" e "*lista_info_imprimir*", que são $O(n)$, todas as funções são $O(1)$;

it_mov.c : todas as funções deste arquivo são $O(1)$;

lista_mov.c:

- *"lista_mov_esvazia"*, *"lista_mov_apagar"*, *"lista_mov_busca_posicao"*, *"lista_mov_busca_cidade"*, *"lista_mov_imprimir"*, *"mov_criar"*, *"mov_imprimir"* são $O(n)$;
- o restante é $O(1)$;

functions.c:

- "*procura_caminhos*", "*completa*" e "*volta*" são $O(n^2)$;
- "*guia_criar*", "*final_criar*", "*pre_condicoes*", "*pos_condicoes*", "*verificador*", "*printa_final*", "*resolve*" e "*ler_aquivo*" são $O(n)$;
- "*apaga_aux*" é $O(1)$;

Conclui-se, portanto, que a ***main.c*** é $O(n^2)$.

Quanto ao gráfico de desempenho em relação ao número de cidades, será discutido ao final do próximo tópico!

Parte Extra: Método Inteligente

O método inteligente que desenvolvemos opera com uma única lista que é utilizada apenas para formar uma matriz com os inputs, essa é a chamada "Matriz Adjacência"
e possui tamanho (número de cidades) X (número de cidades) e cada elemento "Mij" relaciona a distância da cidade i para j (ou de j para i, já que são as mesmas).

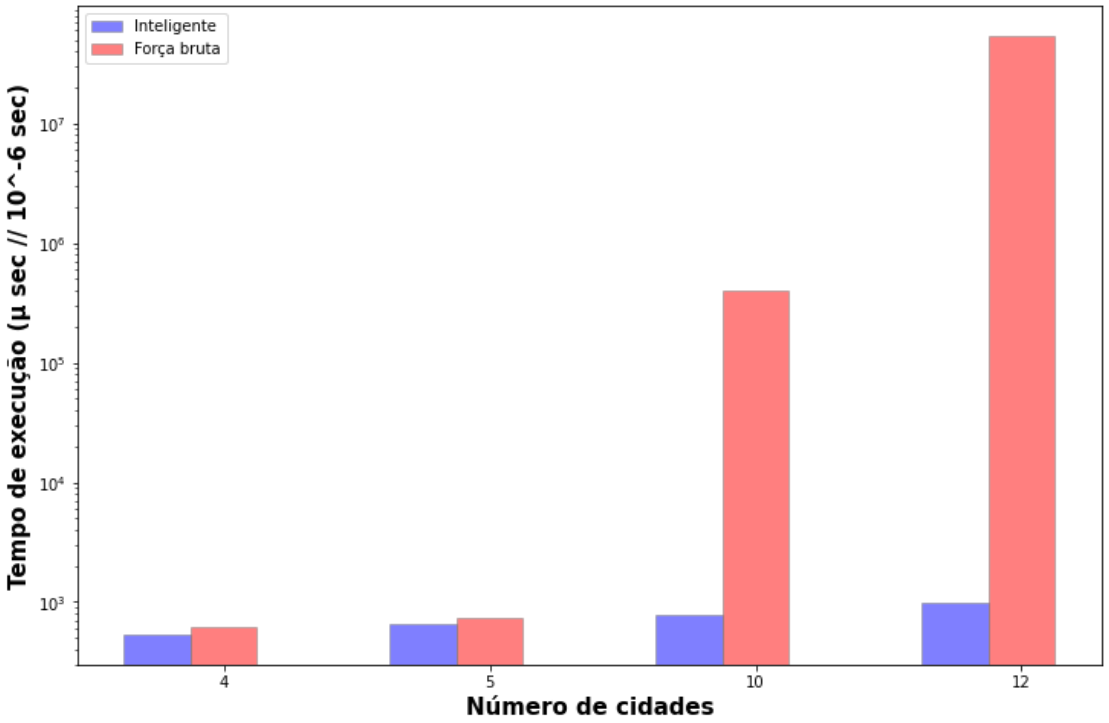
O programa seleciona o menor caminho que cada cidade pode fazer de uma em uma e após ter essa primeira rota, trabalha em cima dela, buscando alterar uma cidade por outra vizinha, uma de cada vez, analisando se o custo aumentou ou não. Tal loop de alterações roda "o número de cidade" vezes. Em seguida imprime a rota que resultou em menor custo!

O código do método inteligente é $O(n^2)$ também, no entanto a constante que multiplica n^2 é muito menor do que a do método de força bruta, e isso justifica a discrepância nos tempos registrados!

Veja abaixo o gráfico comparativo entre o método bruto e o inteligente:

Note também que o gráfico se dispõe em escala logarítmica, e de fato o método bruto custa muito mais tempo, o caso de 12 cidades por exemplo demorou 54 segundos, já o método inteligente não custou mais de um segundo!

*OBS: Caso não ocorra o vínculo entre 2 cidades, a matriz colocará 999 no lugar (como se o custo fosse altíssimo)



E portanto, como esperado, a teoria e a prática se encontraram, provando por experiência que de fato o código de força bruta é mais lento, não apenas em análise assintótica de *big* O mas também graficamente para casos tangíveis!