



Trabalho Prático I (TP I) - 10 pontos, peso 1.

- Data de entrega: 29/10/2023 até 23:55. O que vale é o horário do *Moodle*, e não do *seu*, ou do *meu* relógio!!!
- Clareza, identificação e comentários no código também vão valer pontos. Por isso, escolha cuidadosamente o nome das variáveis e torne o código o mais legível possível.
- O padrão de entrada e saída deve ser respeitado exatamente como determinado no enunciado. Parte da correção é automática, não respeitar as instruções enunciadas pode acarretar em perda de pontos.
- Durante a correção, os programas serão submetidos a vários casos de testes, com características variadas.
- A avaliação considerará o tempo de execução e o percentual de respostas corretas.
- Eventualmente serão realizadas entrevistas sobre os estudos dirigidos para complementar a avaliação;
- O trabalho é em grupo de até 2 (duas) pessoas.
- Entregar um relatório.
- Os códigos fonte serão submetidos a uma ferramenta de detecção de plágios em software.
- Códigos cuja autoria não seja do aluno, com alto nível de similaridade em relação a outros trabalhos, ou que não puder ser explicado, acarretará na perda da nota.
- Códigos ou funções prontas específicas de algoritmos para solução dos problemas elencados não são aceitos
- Não serão considerados algoritmos parcialmente implementados.
- Procedimento para a entrega:
 1. Submissão: via *Moodle*.
 2. Os nomes dos arquivos e das funções devem ser especificados considerando boas práticas de programação.
 3. Funções auxiliares, complementares aquelas definidas, podem ser especificadas e implementadas, se necessário.
 4. A solução deve ser devidamente modularizada e separar a especificação da implementação em arquivos *.h* e *.c* sempre que cabível.
 5. Os arquivos a serem entregues, incluindo aquele que contém *main()*, devem ser compactados (*.zip*), sendo o arquivo resultante submetido via *Moodle*.
 6. Você deve submeter os arquivos *.h*, *.c* e o *.pdf* (relatório) na raiz do arquivo *.zip*. Use os nomes dos arquivos *.h* e *.c* exatamente como pedido.
 7. Caracteres como acento, cedilha e afins não devem ser utilizados para especificar nomes de arquivos ou comentários no código.
- **Bom trabalho!**

Problema do Caixeiro Viajante

Imagine um cenário em que um caixeiro viajante necessita percorrer um conjunto de n cidades distintas, começando e terminando sua jornada na primeira cidade, dado um conjunto de cidades e as distâncias entre todas as possíveis duplas de cidades. Nesse contexto, a ordem de visita das cidades não influencia. O problema do caixeiro viajante consiste em determinar a trajetória que resulta na menor distância total de viagem. A Figura 1 mostra um conjunto de cidades e as distâncias entre elas.

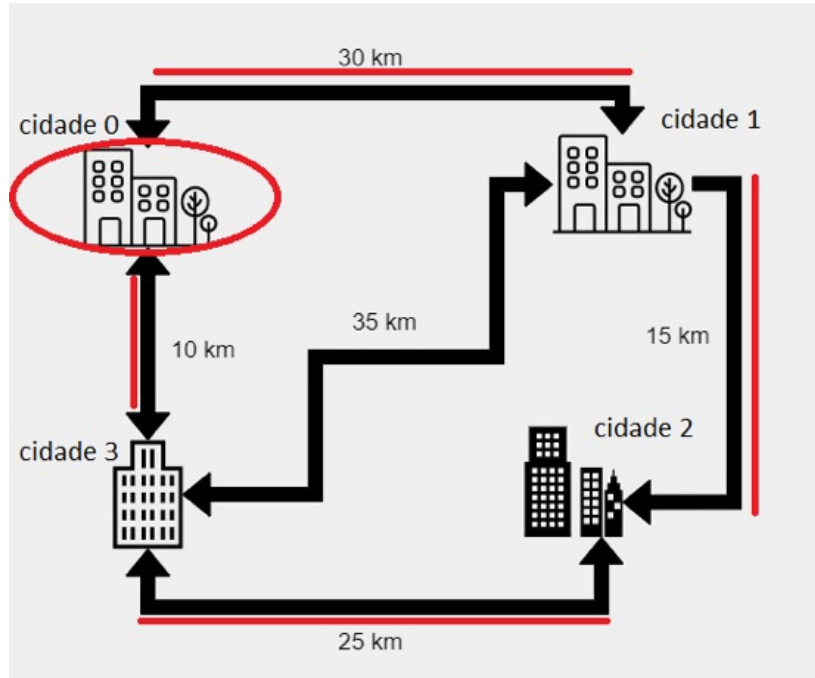


Figura 1: Cidades e a distância entre elas.

A Figura 2 apresenta o menor caminho, partindo da cidade 0, que passa por todas as cidades da Figura 1.

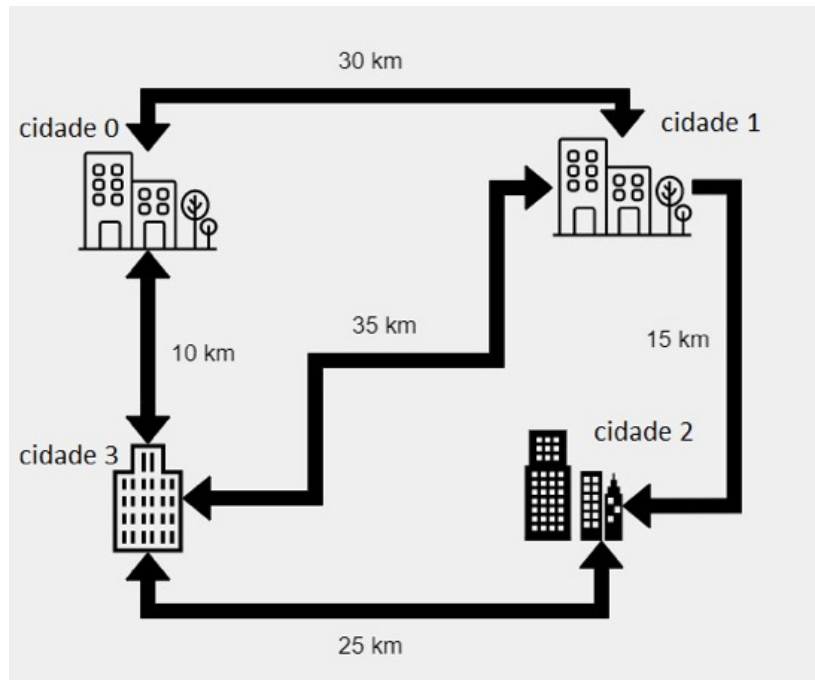


Figura 2: Menor caminho entre as cidades.

Partindo disso, sua tarefa será encontrar o menor caminho possível, passando por todas as cidades, partindo da cidade inicial (**cidade 0**). Para isso, você irá utilizar **recursividade**.

Imposições e comentários gerais

Neste trabalho, as seguintes regras devem ser seguidas:

- Seu programa não pode ter *memory leaks*, ou seja, toda memória alocada pelo seu código deve ser corretamente liberada antes do final da execução. (Dica: utilize a ferramenta *valgrind* para se certificar de que seu código libera toda a memória alocada).
- Um grande número de *Warnings* ocasionará a redução na nota final.

O que deve ser entregue

- Código fonte do programa em C (**bem indentado e comentado**).
- Documentação do trabalho (relatório¹). A documentação deve conter:
 1. **Introdução:** descrição sucinta do problema a ser resolvido e visão geral sobre o funcionamento do programa.
 2. **Implementação:** descrição sobre a implementação do programa. **Não faça** “*print screens*” de telas. Ao contrário, procure resumir ao máximo a documentação, fazendo referência ao que julgar mais relevante. É importante, no entanto, que seja descrito o funcionamento das principais funções e procedimentos utilizados, bem como decisões tomadas relativas aos casos e detalhes de especificação que porventura estejam omissos no enunciado. Muito importante: os códigos utilizados na implementação devem ser inseridos na documentação.
 3. **Estudo de Complexidade:** estudo da complexidade do tempo de execução dos procedimentos implementados e do programa como um todo (notação O), considerando entradas de tamanho n .
 4. **Testes:** descrição dos testes realizados e listagem da saída (não edite os resultados).
 5. **Análise:** deve ser feita uma análise dos resultados obtidos com este trabalho. Por exemplo, avaliar o tempo gasto de acordo com o tamanho do problema.
 6. **Conclusão:** comentários gerais sobre o trabalho e as principais dificuldades encontradas em sua implementação.
 7. **Bibliografia:** bibliografia utilizada para o desenvolvimento do trabalho, incluindo sites da Internet se for o caso.
 8. **Formato:** PDF ou HTML.

Como deve ser feita a entrega

Verifique se seu programa compila e executa na linha de comando antes de efetuar a entrega. Quando o resultado for correto, entregue via *Moodle* até 29/10/2023 até 23:55, um arquivo **.ZIP** com o nome e sobrenome do aluno. Esse arquivo deve conter: (i) os arquivos *.c* e *.h* utilizados na implementação, (ii) instruções de como compilar e executar o programa no terminal, e (iii) o relatório em **PDF**.

Detalhes da implementação

Para atingir o seu objetivo, você deverá construir um Tipo Abstrato de Dados (TAD) **GrafoPonderado** como representação do caminho que você quer analisar. Ele possui os seguintes atributos: número de cidades e a matriz de adjacências. O TAD deverá implementar, pelo menos, as seguintes operações:

1. **alocarGrafo:** aloca um (ou mais) TAD **GrafoPonderado**.
2. **desalocarGrafo:** desaloca um TAD **GrafoPonderado**.
3. **leGrafo:** inicializa o TAD **GrafoPonderado** a partir de dados do terminal.
4. **encontraCaminho:** função recursiva que retorna o menor caminho no grafo fornecido.

¹Exemplo de relatório: <https://www.overleaf.com/latex/templates/modelo-relatorio/vprmcsgmcdg>.

5. `imprimeCaminho`: imprime na tela o menor caminho e a distância total percorrida.

O TAD deve ser implementado utilizando a separação interface no `.h` e implementação `.c` bem como as convenções de tradução.

Considerações

O código-fonte deve ser modularizado corretamente em três arquivos: `main.c`, `grafo.h` e `grafo.c`. O arquivo `main.c` deve apenas invocar e tratar as respostas das funções e procedimentos definidos no arquivo `grafo.h`. A separação das operações em funções e procedimentos está a cargo do aluno, porém, não deve haver acúmulo de operações dentro de uma mesma função/procedimento.

O limite de tempo para solução de cada caso de teste é de apenas **um segundo**. Além disso, o seu programa não pode ter *memory leaks*, ou seja, toda memória alocada pelo seu código deve ser corretamente liberada antes do final da execução. (Dica: utilize a ferramenta *valgrind* para se certificar de que seu código libera toda a memória alocada). *Warnings* ocasionará a redução pela metade da nota final. Assim sendo, utilize suas habilidades de programação e de análise de algoritmos para desenvolver um algoritmo correto e rápido!

Entrada

A entrada é dada por meio do **terminal**. Para facilitar, a entrada (e a saída esperada) será fornecida por meio de arquivos.² A primeira linha especifica o número de cidades a serem visitadas. A seguir são fornecidas as cidades, suas cidades vizinhas e a distância entre elas, em cada linha. Abaixo um exemplo de entrada.

Saída

A saída consiste no menor caminho encontrado (deve haver **sempre** um espaço após o número da cidade), e a distância total percorrida, na segunda linha.

Exemplo de casos de teste

Exemplos de saídas esperadas dada uma entrada:

Entrada	Saída
4 0 0 0 0 1 30 0 2 0 0 3 10 1 0 30 1 1 0 1 2 15 1 3 35 2 0 0 2 1 15 2 2 0 2 3 25 3 0 10 3 1 35 3 2 25 3 3 0	0 1 2 3 0 80

²Para usar o arquivo como entrada no terminal, utilize `./executavel < nome_do_arquivo_de_teste`.

A SAÍDA DA SUA IMPLEMENTAÇÃO DEVE SEGUIR EXATAMENTE A SAÍDA PROPOSTA.

Diretivas de Compilação

As seguintes diretivas de compilação devem ser usadas (essas são as mesmas usadas no *Moodle*).

```
$ gcc -c grafo.c -Wall
$ gcc -c main.c -Wall
$ gcc grafo.o main.o -o exe -lm
```

Avaliação de *leaks* de memória

Uma forma de avaliar se não há *leaks* de memória é usando a ferramenta *valgrind*. O *valgrind* é um *framework* de instrumentação para análise dinâmica de um código e é muito útil para resolver dois problemas em seus programas: **vazamento de memória e acesso a posições inválidas de memória** (o que pode levar a *segmentation fault*). Um exemplo de uso é:

```
1 gcc -g -o exe arquivo1.c arquivo2.c -Wall
2 valgrind --leak-check=full -s ./exe < casoteste.in
```

Espera-se uma saída com o fim semelhante a:

```
1 ==xxxxxx== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Para instalar no Linux, basta usar: `sudo apt install valgrind`.

O SEU CÓDIGO SERÁ TESTADO NOS COMPUTADORES DO LABORATÓRIO
(AMBIENTE LINUX)

PONTO EXTRA

Será concedido 0,1 extra para quem gerar o relatório em \LaTeX (Latex). Deve ser enviado os *.tex* e o arquivo *PDF* (o não envio anula o ponto extra).