

Estrutura de Dados II - 2018/2

Trabalho Prático T1

2 de setembro de 2018

Atualizado em 07 de setembro de 2018

1 Objetivo

O objetivo deste trabalho é implementar uma *heurística* para o TSP (*travelling salesman problem*) utilizando uma MST (*minimal spanning tree*).

2 Aproximações para o TSP

Imagine um caixeiro viajante que começa em alguma cidade e tem de visitar $N - 1$ outras cidades (para vender alguma coisa em cada cidade) e retornar para o início. O caixeiro gostaria de realizar essa viagem (*tour*) com o menor custo possível. Aqui, o custo pode ser tempo de viagem, gasolina, eletricidade, tanto faz.

2.1 O TSP Euclidiano – ETSP

Seja d uma matriz $N \times N$ de distâncias Euclidianas em um plano 2D, aonde $d_{i,j}$ indica o *custo* de se viajar da cidade i para a cidade j . Essa matriz define uma *métrica*, isto é, valem as seguintes propriedades:

1. $d_{i,j} \geq 0$ para todo $i, j \in \{1, \dots, N\}$, i.e., as distâncias não são negativas.
2. $d_{i,i} = 0$ para todo $i \in \{1, \dots, N\}$.
3. $d_{i,j} = d_{j,i}$ para todo $i, j \in \{1, \dots, N\}$, i.e., as distâncias são simétricas.
4. $d_{i,k} \leq d_{i,j} + d_{j,k}$ para todo $i, j, k \in \{1, \dots, N\}$, i.e., as distâncias satisfazem a *inequalidade do triângulo*.

O objetivo do problema é encontrar uma *permutação (tour)* $\pi = \pi_1, \dots, \pi_N$ de $1, \dots, N$ que minimiza a função de custo

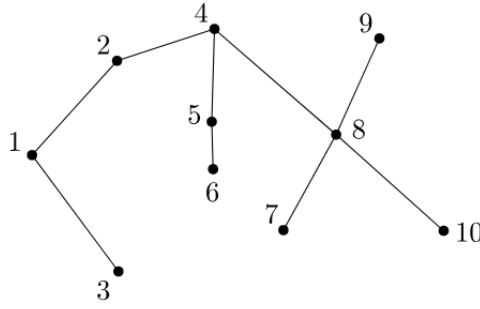
$$w(\pi) = \sum_{i=1}^{N-1} d_{\pi_i, \pi_{i+1}} + d_{\pi_N, \pi_1} \quad .$$

2.2 Uma aproximação baseada em árvores geradoras mínimas

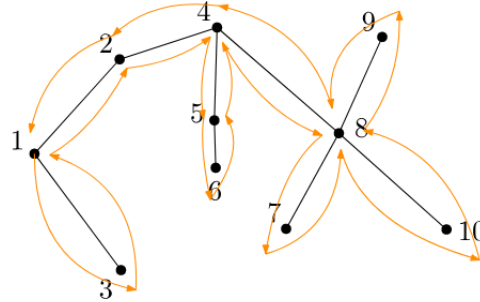
Uma árvore geradora mínima (*minimum spanning tree - MST*) é um subconjunto de arcos de um grafo não-direcionado conexo e ponderado (isto é, com peso nos arcos). Por ser uma árvore, uma MST não pode ter ciclos. Por ser mínima, a soma dos pesos dos arcos tem de ser a menor possível.

Podemos pensar que a matriz d descreve um grafo G completo e ponderado cujo conjunto de vértices é $V = \{1, \dots, N\}$, aonde o peso de um arco ij é $d_{i,j}$. Podemos computar a MST de G usando o famoso *Algoritmo de Kruskal* (veja https://en.wikipedia.org/wiki/Kruskal%27s_algorithm). A complexidade deste algoritmo é $O(|E| \log |V|)$, aonde E é o conjunto de arcos de G . Como G é um grafo completo, temos que $|E| = N(N - 1)/2$, e assim, a complexidade do algoritmo em função de N é $O(N^2 \log N)$.

A figura abaixo mostra uma MST construída para 10 pontos distribuídos no plano 2D.



Seja T a MST acima, e imagine a execução de um caminhamento em T , aonde registramos a lista de vértices a medida que eles são encontrados, como por exemplo, na figura abaixo.



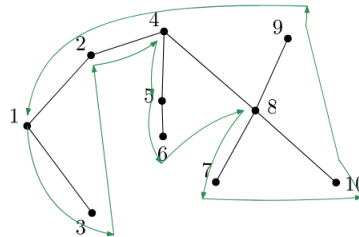
No caminhamento acima, obtemos a seguinte sequência de vértices

$$S_1 = 1, 3, 1, 2, 4, 5, 6, 5, 4, 8, 7, 8, 10, 8, 9, 8, 4, 2, 1 \quad .$$

Agora, removemos todas as ocorrências dos vértices exceto a primeira, obtendo a nova sequência

$$S_2 = 1, 3, 2, 4, 5, 6, 8, 7, 10, 9 \quad .$$

Note que S_2 é uma permutação de $1, \dots, N$, pois S_1 contém todos os elementos de $1, \dots, N$ pelo menos uma vez, e portanto S_2 contém cada elemento exatamente uma vez. Assim, S_2 é uma solução válida para o TSP. A figura abaixo mostra o *tour* descrito por S_2 .



O quão boa é a solução encontrada acima? Cada arco de T é usado exatamente duas vezes em S_1 , assim

$$w(S_1) = 2w(T) \quad .$$

Pela desigualdade do triângulo, temos que

$$w(S_2) \leq w(S_1) \quad .$$

Lembrando que T é uma MST, sabemos que T é um sub-grafo conexo de G com peso mínimo. Mas um *tour* do TSP também é um sub-grafo conexo de G , assim

$$w(T) \leq w(C^*) \quad ,$$

aonde C^* é um *tour* ótimo. Juntando todas essas relações, obtemos

$$w(S_2) \leq w(S_1) \leq 2w(T) \leq 2w(C^*) \quad .$$

Assim, encontramos uma solução, S_2 , para o TSP cujo valor é sempre *no máximo* duas vezes a solução ótima.

3 TSPLIB – Entrada e Saída

O TSP é seguramente um dos problemas de otimização mais estudados atualmente, devido a sua grande aplicabilidade nos mais variados cenários. Nessa seção nós vamos descrever brevemente os arquivos da TSPLIB (<https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/>), a biblioteca de casos de teste padrão para o problema. Se você quiser saber mais sobre o TSP, visite a página mais completa sobre o assunto: <http://www.math.uwaterloo.ca/tsp/index.html>.

O formato da TSPLIB é explicado em detalhes em <https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp95.pdf>. Vamos descrever aqui somente as informações fundamentais, então veja a documentação se quiser saber mais. (Não é necessário para fazer esse trabalho.)

MUITO IMPORTANTE: a TSPLIB convencionou que a numeração dos nós começa de 1. Vamos manter essa convenção nos arquivos de entrada e saída disponibilizados no AVA. (Veja o arquivo ED2_Trab1_io.zip.)

3.1 Uma instância do problema

Um arquivo de entrada que descreve uma instância do problema é dado abaixo. (Veja o arquivo in/berlin52.tsp contido em ED2_Trab1_io.zip.)

```
NAME: berlin52
COMMENT: 52 locations in Berlin (Groetschel)
TYPE: TSP
DIMENSION: 52
EDGE_WEIGHT_TYPE: EUC_2D
NODE_COORD_SECTION
1 565.0 575.0
2 25.0 185.0
3 345.0 750.0
4 945.0 685.0
...
52 1740.0 245.0
EOF
```

Os campos sempre estão presentes e sempre na ordem acima. São eles:

- NAME: indica o nome do problema. Por convenção, os nomes sempre incluem o número de pontos (dimensão).
- COMMENT: uma ou mais linhas de comentários que podem ser ignoradas.
- TYPE: descreve o tipo do grafo descrito no arquivo. A única opção válida é TSP, que indica dados de um problema de TSP simétrico.
- DIMENSION: indica o valor de N (o número de cidades).
- EDGE_WEIGHT_TYPE: descreve o tipo de peso dos arcos. A única opção válida é EUC_2D, que indica distância Euclidiana em um plano 2D.
- NODE_COORD_SECTION marca o início da seção das coordenadas dos nós. Cada nó (cidade) é descrito em uma linha como abaixo:

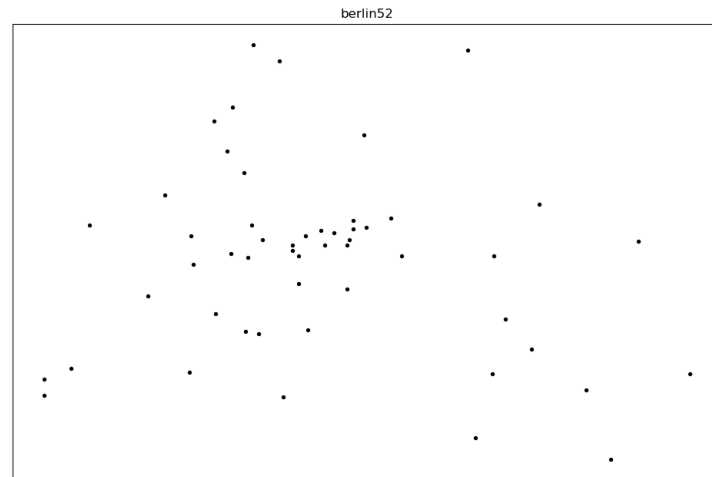
`node_id x y`

aonde `node_id` é um inteiro único (≥ 1) e (x, y) são coordenadas Cartesianas do plano. Os valores de x e y são números reais. **IMPORTANTE:** como é possível ver nos arquivos de entrada, os campos dessa seção podem estar separados por mais de um espaço em branco.

- EOF indica o fim do arquivo.

MUITO IMPORTANTE: Você pode sempre assumir que a entrada do seu programa será um arquivo válido como descrito acima. Utilize os arquivos de entrada disponibilizados no diretório `in/`. Cuidado se for utilizar arquivos diretamente da TSPLIB pois eles nem sempre seguem precisamente as convenções descritas acima. Os arquivos disponibilizados no AVA foram todos uniformizados para simplificar o I/O.

O arquivo `tsp_plot.py.zip` disponibilizado no AVA contém um programa Python para visualização do TSP. O comando `./tsp_plot.py in/berlin52.tsp` gera a figura abaixo.



IMPORTANTE: para executar o programa Python você precisa da biblioteca `matplotlib`, que pode ser facilmente instalada como um pacote na maioria das distribuições Linux. **Se você tiver *qualquer* problema em executar o programa Python ou tiver qualquer outra dificuldade com I/O durante o desenvolvimento do trabalho, fala imediatamente com o professor.**

O seu trabalho deve ler como entrada um arquivo contendo uma instância do problema do TSP como descrita acima, e deve gerar como saída dois arquivos, um contendo a MST e outro contendo o *tour*, cujos formatos são descritos nas próximas duas seções.

3.2 Formato da MST

O arquivo de saída contendo a MST gerada pelo seu trabalho deve ter o formato como a seguir. (Veja o arquivo `mst/berlin52.mst`.)

```
NAME: berlin52
TYPE: MST
DIMENSION: 52
MST_SECTION
35 36
24 48
34 35
...
33 43
EOF
```

Observações sobre os campos:

- NAME e DIMENSION devem ser copiados da entrada.
- o campo TYPE deve ser sempre MST.
- MST_SECTION marca o início da seção de arcos da MST. Cada arco é descrito em uma linha como abaixo:

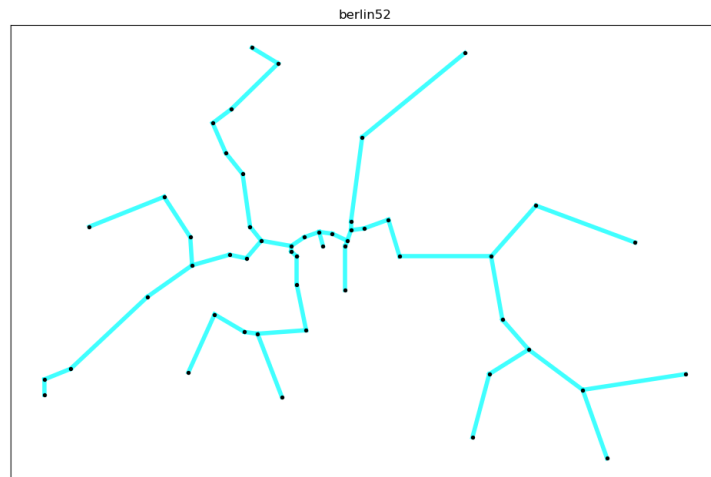
```
i j
```

aonde i e j são as cidades conectadas pelo arco. **MUITO-IMPORTANTE:** os arcos dessa seção devem ser listados em ordem crescente de peso. Os arcos dessa seção podem ser listados em QUALQUER ORDEM.

- EOF indica o fim do arquivo.

A MST também pode ser visualizada pelo programa `./tsp_plot.py`, exceto para `in/d18512.tsp`, que é muito grande.

O comando `./tsp_plot.py in/berlin52.tsp mst/berlin52.mst` gera a figura abaixo.



3.3 Formato do *tour*

O arquivo de saída contendo o *tour* gerado pelo seu trabalho deve ter o formato como a seguir. (Veja o arquivo `tour/berlin52.tour`.)

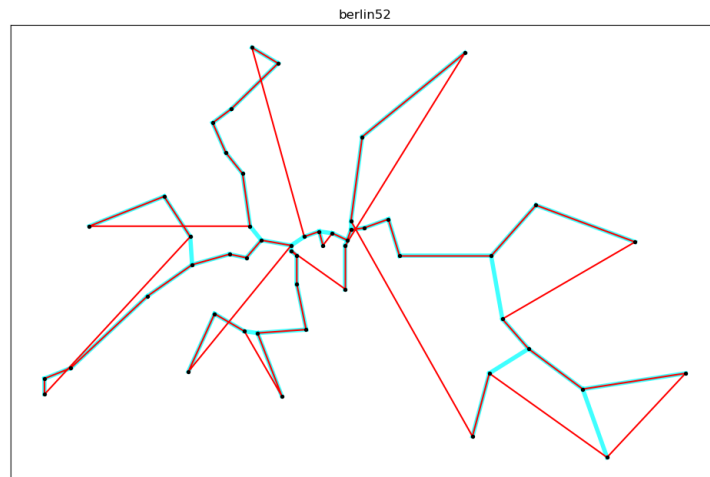
```
NAME: berlin52
TYPE: TOUR
DIMENSION: 52
TOUR_SECTION
1
22
31
...
49
EOF
```

Observações sobre os campos:

- NAME e DIMENSION devem ser copiados da entrada.
- o campo TYPE deve ser sempre TOUR.
- TOUR_SECTION marca o início da seção das cidades no *tour*. Cada linha contém uma cidade no *tour*. **MUITO-IMPORTANTE:** o *tour* deve sempre começar pela cidade 1. O *tour* pode começar por QUALQUER CIDADE.
- EOF indica o fim do arquivo.

O *tour* também pode ser visualizado pelo programa `./tsp_plot.py`.

O comando `./tsp_plot.py in/berlin52.tsp mst/berlin52.mst tour/berlin52.tour` gera a figura abaixo.

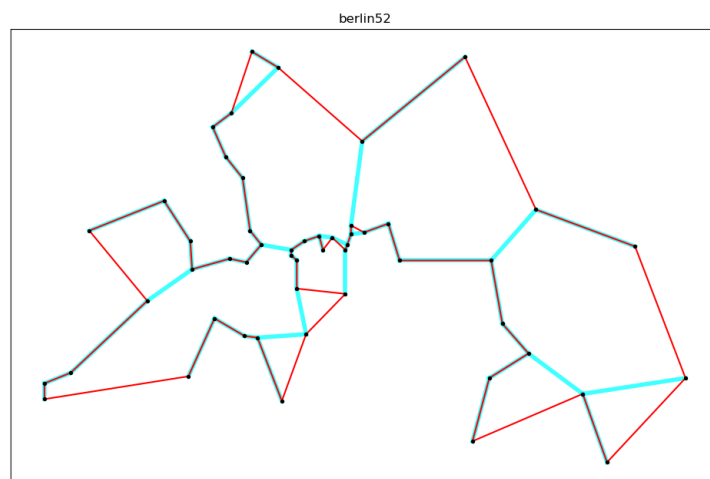


3.4 Tours ótimos

A título de ilustração, alguns *tours* ótimos foram disponibilizados no diretório `opt/`.

AVISO: Você não precisa deles para esse trabalho.

O comando `./tsp_plot.py in/berlin52.tsp mst/berlin52.mst opt/berlin52.opt.tour` gera a figura abaixo.



3.5 Execução do trabalho

O seu trabalho será executado da seguinte maneira:

```
./trab1 <problema>.tsp
```

Isto é, o seu programa recebe como entrada um arquivo de uma instância do TSP e gera como saída *dois* arquivos, de nome `<problema>.mst` e `<problema>.tour`, contendo respectivamente a MST e o *tour* calculados, seguindo os formatos descritos anteriormente. (Substitua `<problema>` pelo nome real do arquivo passado como argumento.)

4 Detalhes de implementação

ABSOLUTAMENTE ESSENCIAL: A saída do seu programa deve ser exatamente (no mesmo formato) como nos arquivos de saída disponibilizados no AVA. Garanta que seu programa gerará exatamente o mesmo resultado do gabarito usando o utilitário `diff` (no Linux). Um exemplo de comandos:

\$ // REMOVIDO

~~Aonde assumimos que os diretórios `mst/` e `tour/` contém os arquivos de saída para teste disponibilizados pelo professor. O resultado dos `diff` para os casos de teste devem sempre ser vazio, indicando que os arquivos comparados são idênticos. Esse deve ser o resultado do seu trabalho para todos os casos de teste no AVA. Trabalhos que não atendam esse requisito serão severamente prejudicados na nota. Isto é absolutamente necessário porque será usada uma bateria de testes para validação de seu trabalho, que verifica se sua resposta está correta baseado na saída do seu programa.~~

~~É possível existirem dois ou mais arcos com o mesmo peso (distância). Isso implica que a MST não é única, isto é, é possível construir duas ou mais árvores distintas com o peso mínimo. Como pode existir mais de uma MST, também pode existir mais de um *tour*. Como não faz sentido colocar restrições arbitrárias para a saída ficar igual à do professor, essa parte da especificação foi retirada.~~

~~Utilize os arquivos de MST e *tour* fornecidos pelo professor como REFERÊNCIA para construir as suas soluções. O resultado do seu programa deve ser uma MST e um *tour* CORRETOS, embora eles NÃO precisem ser iguais aos do professor. O seu trabalho será corrigido manualmente... :- (~~

Seu programa deve ser, obrigatoriamente, compilado com o utilitário `make`. Crie um arquivo `Makefile` que gera como executável para o seu programa um arquivo de nome `trab1`.

4.1 Passos para o desenvolvimento do trabalho

Você deve realizar todos os passos abaixo, na ordem indicada, para concluir o seu trabalho.

1. Leia atentamente **todo** esse documento de especificação. Certifique-se de que você entendeu tudo que está escrito aqui. Havendo dúvidas ou problemas, fale com o professor o quanto antes. As dúvidas podem ser sanadas pessoalmente com o professor ou por email.
2. Instale as bibliotecas Python necessárias para fazer o programa `tsp_plot.py` funcionar.
3. Utilizando o programa `tsp_plot.py`, visualize os arquivos de teste (`.tsp`) disponibilizados para se familiarizar com o formato dos problemas.
4. Desenvolva uma estrutura de dados para armazenar as informações do problema do TSP. Você é livre para usar a(s) estrutura(s) que preferir.
5. Crie código que lê o arquivo de entrada fornecido e armazena todos os dados na estrutura que você projetou no item anterior. **IMPORTANTE: teste adequadamente essa parte do código antes de continuar para se certificar que todos os dados foram lidos corretamente.**
6. Construa a MST do problema de entrada utilizando o algoritmo de Kruskal. Algumas observações:
 - Veja o link da seção 2.2 para uma explicação do algoritmo na Wikipedia.
 - A estrutura de dados *disjoint-set* mencionada no pseudo-código da Wikipedia é a estrutura de *union-find* vista em sala. O código dessa estrutura está disponível no AVA juntamente com os slides da Aula 01. Adapte a versão mais eficiente do algoritmo para as suas necessidades e use-a no seu trabalho.
 - O algoritmo de Kruskal requer que os arcos do grafo do TSP sejam ordenados pelo peso. Alguns comentários específicos sobre isso:
 - É imediato notar que os arcos não são fornecidos explicitamente na entrada. No entanto, como as cidades são pontos no plano, você pode calcular o peso do arco como a distância Euclidiana. **MUITO IMPORTANTE: para evitar erros de arredondamento e outros problemas, as distâncias (pesos dos arcos) sempre devem ser números INTEIROS.** Utilize algo como a função abaixo para calcular a distância entre duas cidades i e j .

```
int compute_dist(int i, int j) {
    int xd = x[i] - x[j];
    int yd = y[i] - y[j];
    return round(sqrt(xd*xd + yd*yd));
}
```

No código acima as funções `round` e `sqrt` vêm da biblioteca `math.h`.

- Crie uma estrutura de dados para representar um arco da MST e crie um *array* de arcos, usando os dados do problema de entrada para preencher o *array*. Note que é simples saber a quantidade de arcos, uma vez que o grafo de entrada é completo (veja seção 2).
 - Ordene o *array* de arcos de forma não-decrescente, utilizando o peso do arco como a chave de ordenação. Você deve usar um algoritmo de ordenação eficiente, como *quicksort* ou *mergesort*. Esse é o ponto crítico do tempo de execução de todo o trabalho. *Obs.: Existem outras formas de construir a MST sem gerar todos os arcos e ordená-los mas isso requer estruturas de dados e algoritmos mais elaborados que não cabem nesse curso.*
7. Implemente o algoritmo de Kruskal segundo descrito acima e gere a MST do problema de entrada.
 8. Gere o arquivo de saída da MST no formato descrito anteriormente.
 9. Implemente um caminhamento na MST que gera o *tour* conforme descrito na seção 2.2. Pode ser necessário utilizar estruturas de dados adicionais/auxiliares. Você é livre projetar essa parte de código da forma que preferir, mas note que um código mal projetado aqui pode ter um impacto negativo considerável no desempenho do seu programa.
 10. Execute o caminhamento na MST para calcular o *tour*, gerando o seu arquivo de saída no formato descrito anteriormente.

Algumas considerações finais:

- Ao longo do desenvolvimento do trabalho, certifique-se que o seu código não está vazando memória testando-o com o *valgrind*. Não espere terminar o código para usar o *valgrind*, incorpore-o no seu ciclo de desenvolvimento. Ele é uma ferramenta excelente para se detectar erros sutis de acesso à memória que são muito comuns em C. Idealmente o seu programa deve sempre executar sem nenhum erro no *valgrind*.
- **ATENÇÃO:** Não é necessária nenhuma estrutura de dados muito elaborada para o desenvolvimento deste trabalho. Todas as estruturas que você vai precisar foram discutidas em aula ou no laboratório. Veja os códigos disponibilizados pelo professor para ter ideias. Prefira estruturas simples a coisas muito complexas. Pense bem sobre as suas estruturas e algoritmos antes de implementá-los: quanto mais tempo projetando adequadamente, menos tempo depurando o código depois. Em resumo: se está complicado demais pare e pense no que você está fazendo porque não deveria ser assim. Aproveite e converse com o professor para tirar suas dúvidas. Resumo do resumo: é possível implementar esse trabalho tranquilamente em menos de mil linhas de código C. Se você estiver usando muito mais do que isso, algo pode estar errado.

5 Regras para desenvolvimento e entrega do trabalho

- **Data da Entrega:** O trabalho deve ser entregue até às 23:55 h do dia 23/09/2018 (domingo). Não serão aceitos trabalhos após essa data.
- **Prazo para tirar dúvidas:** Para evitar atropelos de última hora, você deverá tirar todas as suas dúvidas sobre o trabalho até o dia 21/09/2018. A resposta para dúvidas que surgirem após essa data fica a critério do professor.
- **Grupo:** O trabalho deve ser feito em **dupla**.
- **Linguagem de Programação e Ferramentas:** Você deve desenvolver o trabalho na linguagem C, sem usar ferramentas proprietárias (e.g., Visual Studio). O seu trabalho será corrigido no Linux.
- **Como entregar:** Pela atividade criada no AVA. Envie um arquivo compactado com todo o seu trabalho. A sua submissão deve incluir todos os arquivos de código e um *Makefile*, como especificado anteriormente. Além disso, inclua um arquivo de relatório (descrito adiante). **Somente uma pessoa da dupla deve enviar o trabalho no AVA. Coloque o nome completo dos dois integrantes da dupla no nome do arquivo do trabalho.**
- **Recomendações:** Modularize o seu código adequadamente. Crie códigos claros e organizados. Utilize um estilo de programação consistente. Comente o seu código extensivamente. Não deixe para começar o trabalho na última hora.

6 Relatório de resultados

Após terminar de implementar e testar o seu programa, você deve avaliar os resultados obtidos para os casos de teste e preencher uma tabela de resultados experimentais que também deve ser entregue junto com o código do trabalho.

Veja o arquivo `ED2_Trabl_info.txt` disponibilizado no AVA com os resultados obtidos pelo professor. É interessante notar que os custos dos *tours* nunca ultrapassam 2 vezes o ótimo, como já era de se esperar pela a discussão teórica no início desse documento.

Idealmente, o seu programa deve dar a resposta correta em menos de um minuto para todos os casos de teste.

O relatório deve ser entregue em formato `.txt`.

7 Avaliação

- O trabalho vale 2.0 pontos na média parcial do semestre.
- Trabalhos com erros de compilação receberão nota zero.
- Trabalhos que gerem *segmentation fault* para algum dos casos de teste disponibilizados no AVA serão severamente penalizados na nota.
- Trabalhos com *memory leak* (vazamento de memória) sofrerão desconto na nota.
- Caso seja detectado plágio (entre alunos ou da internet), todos os envolvidos receberão nota zero.
- Serão levadas em conta, além da correção da saída do seu programa, a clareza e simplicidade de seu código.
- A critério do professor, poderão ser realizadas entrevistas com os alunos, sobre o conteúdo do trabalho entregue. Caso algum aluno seja convocado para uma entrevista, a nota do trabalho será dependente do desempenho na entrevista. (Vide item sobre plágio, acima.)