

# TRABALHO 0: ÁRVORES E ESTRUTURAS DE PRIORIDADE DRAFT 1

RAUL H.C. LOPES

## 1. PRELIMINARES

Neste trabalho você exercitará seus conhecimentos de algumas estruturas fundamentais da arte de programação de computadores: seqüências, árvores binárias de pesquisa e de prioridade, ordenação. O trabalho consiste de exercícios de implementação sobre as essas estruturas e algoritmos a elas associados.

A seção 2 apresenta os fundamentos de teoria de grafos que serão usados no trabalho. A seção 3 apresenta os exercícios a realizar. A seção 4 define os critérios que serão utilizados na correção do trabalho e atribuição de pontos. A seção 5 define o que deve ser entregue e a forma de submissão do trabalho para correção. Finalmente, a seção 7 descreve resumidamente diferenças entre a versão atual deste documento e versões anteriores.

Algumas dicas importantes sobre a execução do trabalho seguem.

- (1) Não deixe de ler este documento por completo antes de iniciar o trabalho.
- (2) Siga estritamente as especificações deste documento: qualquer desvio delas pode significar a anulação de um exercício ou de todo o trabalho.
- (3) Comece a trabalhar de imediato: o trabalho foi concebido assumindo que você usaria, em período de aulas, cerca de três dias em cada seção de exercícios e um dia na seção final.

O trabalho pode ser executado em grupos, mas, como estabelecido na seção 4, grupos maiores recebem menos créditos. Um elemento importante da avaliação do trabalho trata da detecção de mutualismo parasitário: situação em que um aluno atua de vampiro parasita sugando o resultado do trabalho do grupo. Fato: plágio em qualquer item de um trabalho implica na anulação completa do trabalho do(s) grupo(s) envolvidos. Qualquer elemento de um grupo pode ser, durante a execução do trabalho ou após a entrega do mesmo, avaliado

em relação ao conhecimento do que foi feito. Essa avaliação poderá ser feita de forma oral ou por escrito.

## 2. O PROBLEMA DO CAIXEIRO VIAJANTE

Os exercícios deste trabalho lidam com o *Problema do Caixeiro Viajante*, *Traveling Salesman Problem*, de agora em diante chamado **TSP**. Este problema demanda encontrar o caminho mais curto para visitar cada cidade de um conjunto dado exatamente uma vez e retornar ao ponto de partida. Para representar cidades, usaremos uma abstração: um grafo não dirigido.

Um grafo não dirigido é um par  $(V, E)$ , onde  $V$  é um conjunto de pontos e  $E$  é um conjunto de pares de pontos, cada par de pontos sendo chamado de arestas. Para nosso trabalho valem as seguintes restrições:

- uma aresta  $(u, v)$  indica que existe uma conexão direta do ponto  $u$  ao ponto  $v$ ;
- o grafo é geométrico: os pontos são elementos de um espaço Euclidiano de duas dimensões;
- as coordenadas dos pontos são pares de inteiros positivos;
- para qualquer par de pontos  $u$  e  $v$  do grafo existe uma aresta  $(u, v)$ , que tem um custo associado dado pela distância entre  $u$  e  $v$ ;
- o grafo é não direcionado e a existência de uma aresta  $(u, v)$  indica a existência de outra aresta  $(v, u)$  com mesmo custo que a primeira.

Um caminho em um grafo é uma seqüência de pontos do mesmo. Um circuito Hamiltoniano  $C$  é uma seqüência de pontos com as seguintes restrições:

- cada ponto do grafo inicial figura exatamente uma vez no circuito;
- o circuito indica a seqüência de pontos a visitar para sair de um vértice inicial, o primeiro da seqüência dado, ir até o último vértice da seqüência, visitando todos os outros na ordem dada por  $C$ , e voltar ao ponto de partida.

Um circuito Hamiltoniano, que chamaremos simplesmente de **tour**, consiste em uma permutação do conjunto de pontos de grafo dado. Assumindo que  $C!i$  denote o  $i$ -ésimo elemento do tour  $C$ , o custo do tour é dado por:

$$cost.C = (+i : 0 \leq i < n - 1 : dist(C!i, C!(i + 1))) + dist(C!(n-1), C!0)$$

onde  $dist(u, v)$  é a distância geométrica de  $u$  a  $v$ .

O problema do **TSP**, como colocado neste trabalho, consiste, então, em determinar o **tour** de menor custo.

### 3. OS EXERCÍCIOS

Esta seção contém exercícios sobre estruturas de dados e algoritmos usados freqüentemente em bancos de dados espaciais e problemas sobre grafos como o **TSP**.

Em todos os exercícios represente o **tour** resultante como uma lista de pontos sem repetição. Por exemplo, o **tour** que parte do ponto  $p_0$ , vai para  $p_2$ ,  $p_1$  e volta a  $p_0$  seria representado pela lista:

$$[p_0, p_2, p_1]$$

e não pela lista

$$[p_0, p_2, p_1, p_0]$$

Note que essa restrição refere-se apenas ao **tour** resultante. Sua representação interna dos diversos algoritmos fica a cargo de sua criatividade.

**3.1. Árvores binárias espaciais.** Neste trabalho, serão apenas considerados problemas em duas dimensões. As árvore binárias espaciais usadas armazenarão pontos de duas coordenadas. Uma árvore binária espacial, ver K-d trees for semidynamic point sets, apresenta as seguintes características:

- cada folha armazena uma seqüência de ao menos um ponto e até oito pontos.
- um nó interno particiona o conjunto de pontos de subárvore de que ele é raiz e define ao menos:
  - a coordenada usada no particionamento;
  - o valor da linha de particionamento;
  - a subárvore da esquerda contendo os pontos que têm na coordenada de particionamento valor menor do que (ou igual a) o valor da linha de particionamento;
  - a subárvore da direita contendo os pontos que têm na coordenada de particionamento valor maior do que o valor da linha de particionamento
- no caminho da raiz para as folhas os nós internos alternam o particionamento dos pontos pelas coordenadas dadas.

Defina os seguintes tipos em Haskell:

- *Point a* para representar pontos de um grafo, cujas coordenadas são do tipo *a*.

- *KDtree*  $a$  para representar tipo das *kd-tree* de pontos do tipo *Point a*.

**Exercício 1.** Defina em Haskell um algoritmo com o seguintes tipo:

$$\text{build} :: [\text{Point } a] \rightarrow \text{KDtree } a$$

que, dada uma seqüência de pontos  $S$ , constrói uma *kd-tree* dos pontos de  $S$ .

**Exercício 2.** Apresente em Haskell um algoritmo com o seguinte tipo:

$$\text{isKDtree} :: \text{KDtree } a \rightarrow \text{Bool}$$

que testa a correção do algoritmo apresentado no exercício 1.

**Exercício 3.** Apresente testes comparativos de correção, desempenho e uso de memória, usando dados dos arquivos da seção 3.7.

### 3.2. Árvores binárias de prioridade.

**Exercício 4.** Uma *Árvore binária de prioridade* é uma *árvore binária ordenada da raiz para as folhas*: a raiz é menor (ou igual) a qualquer elemento da *árvore* e cada uma das suas *subárvores* é uma *Árvore binária de prioridade*.

Considere a seguinte definição de operador para realizar o merge de duas *árvore de prioridade*.

- (1)  $\text{merge } \perp y = y$
- (2)  $\text{merge } x \perp = x$
- (3)  $a < b \Rightarrow \text{merge } \langle x, a, y \rangle \langle u, b, v \rangle = \langle \text{merge } y \langle u, b, v \rangle, a, x \rangle$
- (4)  $a \geq b \Rightarrow \text{merge } \langle x, a, y \rangle \langle u, b, v \rangle = \langle \text{merge } \langle x, a, y \rangle v, b, u \rangle$

Defina em Haskell os seguintes operadores e tipos:

- *Binpq a*  
Para *Árvores binárias de prioridade*.
- $\text{merge} :: \text{Binpq } a \rightarrow \text{Binpq } a \rightarrow \text{Binpq}$   
que constrói o merge de duas *Árvores binárias de prioridade*.
- $\text{insert} :: \text{Binpq } a \rightarrow a \rightarrow \text{Binpq } a$   
que, dados *Árvore binária de prioridade*  $h$  e item  $x$ , constrói uma *Árvore binária de prioridade* resultante da inserção de  $x$  em  $h$ .
- $\text{min} : \text{Binpq } a \rightarrow a$   
que retorna o valor mínimo de uma *Árvore binária de prioridade*.

- $delmin : Binq\ a \rightarrow Binq\ a$   
que, dada uma Árvore binária de prioridade  $h$ , exclui dela o valor mínimo.

**Exercício 5.** Defina em Haskell um operador com tipo

$$isbinq :: Binq\ a \rightarrow Bool$$

que testa se  $h$  é uma Árvore binária de prioridade.

**Exercício 6.** Apresente testes de correção, desempenho e uso de memória para cada um dos operadores definidos no exercício 4.

**3.3. TSP e heurística NN.** Nos exercícios a seguir, você trabalhará com a heurística **Nearest Neighbor (NN)**, descrita na página 23 de *Experimental Analysis of Heuristics for the STSP*.

**Exercício 7.** Implemente um algoritmo para encontrar um **tour** para o **TSP**, usando a heurística **NN**. Esta heurística começa com um **tour** parcial contendo um ponto escolhido aleatoriamente do conjunto dado. A cada passo, um novo ponto é escolhido para ser agregado ao **tour**: o ponto escolhido é aquele que se encontra mais próximo do último ponto adicionado ao **tour**.

Seu algoritmo terá o seguinte tipo:

$$nn :: [Point\ a] \rightarrow [Point\ a]$$

**Exercício 8.** Defina um algoritmo

$$istour :: [Point\ a] \rightarrow Bool$$

que testa se seqüência de cidades é um **tour** e use-o para testar a correção da solução do exercício 7.

**3.4. TSP e heurística FRP.**

**Exercício 9.** Use uma kd-tree para particionar o conjunto de cidades dado. Em seguida, percorra a árvore das folhas para a raiz, construindo **tours**:

- em uma folha construa um **tour** dos pontos presentes, usando **NN**;
- em um nó interno, use procedimento similar ao merge do Merge Hull unir os **tours** obtidos nas suas duas subárvores.

Seu algoritmo terá o seguinte tipo:

$$frp :: [Point\ a] \rightarrow [Point\ a]$$

**Exercício 10.** Teste correção, desempenho, uso de memória e qualidade do **tour** da sua solução.

**3.5. TSP e heurística Greedy.** Na heurística **Greedy**, descrita na página 24 de *Experimental Analysis of Heuristics for the STSP*, o **tour** é mantido como uma seqüência de arestas (pares de pontos.) Começando com a aresta de menor peso, a cada passo adiciona-se a aresta  $(u, v)$  que atende às restrições:

- é a aresta de menor peso dentre as candidatas elegíveis;
- $(u, v)$  é elegível se  $u$  e  $v$  têm cada um grau zero ou um;
- a adição de  $(u, v)$  não completa um ciclo.

**Exercício 11.** *Implemente em Haskell uma versão simples da heurística Greedy, usando apenas ordenação de arestas por peso como estrutura básica. Seu algoritmo terá o seguinte tipo:*

$$greedy :: [Point\ a] \rightarrow [Point\ a]$$

**Exercício 12.** *Teste a correção, desempenho, uso de memória e qualidade de **tour** da sua implementação da heurística Greedy.*

### 3.6. Árvore binária espacial semi-dinâmica.

**Exercício 13.** *Implemente na sua árvore binária espacial, conforme descrição de *K-d trees for semidynamic point sets*:*

- operador de exclusão de ponto com tipo

$$delete :: Point\ a \rightarrow (KDtree\ a) \rightarrow (KDtree\ a)$$

- operador com tipo

$$nn :: (Point\ a) \rightarrow (KDtree\ ) \rightarrow (Point\ a)$$

*para encontrar, dentre os pontos não excluídos de uma árvore, o vizinho mais próximo de um ponto dado.*

**Exercício 14.** *Use uma árvore binária espacial e uma fila de prioridade, como descrito na página 24 de *Experimental Analysis of Heuristics for the STSP*, para produzir uma nova versão da heurística **Greedy**, com tipo:*

$$pqgreedy :: [Point\ a] \rightarrow [Point\ a]$$

**3.7. Testes.** O *tarball test.data.tar.bz2* contém arquivos de dados que você poderá utilizar para testar seus programas. Interessa avaliar neste trabalho para cada arquivo:

- correção do **tour** calculado;
- tempo de execução;
- qualidade do **tour** quando comparado com o melhor **tour** possível.

Note que os arquivos de dados estão no formato TSPLIB<sup>1</sup>. Use o programa **fromtsplib.pl** para convertê-los para o formato do programa **tourlen**, apresentado na seção 6.

Abre esse diretório com o comando:

```
tar -jxf test.data.tar.bz2
```

Leia o arquivo **README**, no diretório **test.data** criado, para mais informações.

#### 4. CRÉDITOS

A definição dos critérios de avaliação estará na nova versão deste documento disponível no dia 17/01/05.

#### 5. A SUBMISSÃO PARA CORREÇÃO

A definição do formato de submissão do trabalho estará na versão deste documento do dia 17/01/05.

A data provisória de entrega deste trabalho é 26/01/05.

#### 6. EXEMPLO

O *tarball* *point.io.tar.bz2* contém um exemplo de programa para leitura e escrita de pontos. Abra o arquivo usando

```
tar -jxf point.io.tar.bz2
```

Isso criará um diretório de nome **point.io**. Leia o arquivo **README** desse diretório para instruções adicionais.

#### 7. CORREÇÕES

Esta versão da especificação do trabalho corrige em relação à versão de 10/01/05:

- Seção 3: adicionada restrição sobre representação do **tour** resultante de qualquer heurística.
- seção 3.1: o tipo usado para representar pontos passa a ser parametrizado em relação ao tipo das coordenadas. Isso vale para toda a especificação do trabalho, tendo sido ajustadas as especificações de tipo das funções que implementarão heurísticas para a solução do **TSP**.
- Seção 3.1: o tipo da árvore *kd-tree* passa a ser parametrizado em relação ao tipo das coordenadas dos pontos a armazenar.
- Seção 3.2: ajustes de espaçamento na definição do operador *merge*, em 1.

---

<sup>1</sup> Não deixe de visitar o *site* TSPLIB.

- Seção 3.2: correção no tipo do operador *isbinpq*, no exercício 5.
- Seção 3.6: correção do tipo da função de exclusão de pontos do exercício 13.
- Seção 6 descreve exemplo de *leitura/escrita* de pontos em Haskell.