

Trabalho de TBO

Débora Zupeli Bossois
João Olavo Baião de Vasconcelos

15 de março de 2005

1 Testes comparativos

As seções 2 e 3 deste documento apresentam testes comparativos para as estruturas KD-Tree e Árvore binária de prioridade, e para as heurísticas estudadas do Problema do Caixeiro Viajante (TSP). Os testes foram executados em uma máquina Pentium 4, com processador de 2.6GHz e memória RAM de 1 Gb.

2 KD-Tree

Executamos o algoritmo KDTree para alguns conjuntos de pontos a fim de obter uma tabela com os tempos de execução de cada conjunto e a quantidade de memória usada por cada conjunto. Os resultados são apresentados na tabela 1.

| KDTree | | |
|-----------------|-----------|-------------------------|
| N.ptos | Tempo | Memória (10^5 bytes) |
| att48 | 0m0.018s | 8,363 |
| kroA100 | 0m0.027s | 17,000 |
| lin318 | 0m0.060s | 54,347 |
| pr1002 | 0m0.154s | 177,806 |
| nrw1379 | 0m0.212s | 241,071 |
| pla7397 | 0m1.255s | 1.415,285 |
| pla33810 | 0m6.165s | 6.682,207 |
| pla85900 | 0m16.313s | 17,505.565 |

Tabela 1: Tempo e uso de memória do módulo KDTree

3 Heurísticas do TSP

Nesta seção, realizamos testes comparativos para as heurísticas *Nearest Neighbor*, *FRP*, *Greedy* e *PQGreedy*. Dividimos em três tabelas distintas, que informam o tempo (Tabela 2) e a memória gastos (Tabela 3), e a qualidade do tour (Tabela 4).

| Conj. de ptos | Função | | | |
|-----------------|-----------|----------|------------------|-----------|
| | NN | FRP | Greedy | PQGreedy |
| att48 | 0m0.012s | 0m0.011s | 0m0.021s | 0m0.018s |
| kroA100 | 0m0.019s | 0m0.017s | 0m0.088s | 0m0.031s |
| lin105 | 0m0.020s | 0m0.018s | 0m0.091s | 0m0.033s |
| lin318 | 0m0.051s | 0m0.042s | 0m1.552s | 0m0.097s |
| linhp318 | 0m0.052s | 0m0.042s | 0m1.550s | 0m0.098s |
| pr1002 | 0m0.156s | 0m0.125s | 0m37.874s | 0m0.348s |
| nrw1379 | 0m0.208s | 0m0.170s | 1m37.220s | 0m0.532s |
| C1k | 0m0.208s | 0m0.170s | 0m39.417s | 0m0.348s |
| pla33810 | 0m15.481s | 0m4.978s | (stack overflow) | 2m43.592s |

Tabela 2: Tempo das heurísticas do TSP

| Conj. de ptos | Função | | | |
|-----------------|----------|----------|------------------|------------------|
| | NN | FRP | Greedy | PQGreedy |
| att48 | 1,11 | 1,11 | 2,33 | 2,11 |
| kroA100 | 2,40 | 2,40 | 10,84 | 4,80 |
| lin105 | 2,57 | 2,57 | 11,62 | 5,22 |
| lin318 | 8,38 | 8,38 | 186,83 | 17,90 |
| linhp318 | 8,38 | 8,38 | 186,83 | 17,90 |
| pr1002 | 27,97 | 27,97 | 4,629,28 | 70,61 |
| nrw1379 | 38,32 | 38,32 | 4,629,28 | 103,04 |
| Clk | 30,58 | 30,58 | 4,585,17 | 66,41 |
| pla33810 | 1.818,74 | 1.818,74 | (stack overflow) | (stack overflow) |

Tabela 3: Uso de memória das heurísticas do TSP, em 10^6 bytes

4 Conclusões sobre os testes

A partir da análise dos testes, podemos observar que as heurísticas *Nearest Neighbor* e *FRP* são as melhores em relação ao tempo e uso de memória. Seus valores são bem parecidos, pois ambas utilizam a estrutura KD-Tree, seja simplesmente como forma de otimização (no caso do *NN*), seja como estrutura básica da heurística (*FRP*).

Já em relação ao tour produzido, a heurística *Greedy* foi relativamente melhor do que as outras. Apesar disso, é a pior delas em relação ao tempo e uso de memória, uma vez que possui como estrutura básica a ordenação de todas as arestas, produzindo um tempo quadrático.

Ou seja, não foi possível obtermos em uma só heurística o melhor tempo e o melhor tour. Talvez, dentre as destacadas, a melhor seja o *PQGreedy*, já que possui tempo e uso de memória acessíveis, e ainda assim, produzem um tour relativamente bom.

| Conj. de ptos | Função | | | |
|----------------|--------|--------|--------|----------|
| | NN | FRP | Greedy | PQGreedy |
| att48 | 1.2089 | 1.8216 | 1.1980 | 1.2801 |
| kroA100 | 1.2617 | 1.8874 | 1.1368 | 1.3600 |
| lin105 | 1.4158 | 1.9016 | 1.1661 | 1.1363 |
| pr1002 | 2.9926 | 5.0329 | 2.9218 | 2.8999 |

Tabela 4: Qualidade do tour das heurísticas do TSP (razão entre o melhor tour possível e o tour da função correspondente)

5 Testes de correção do Módulo Binq

Esta seção mostra testes de correção dos operadores utilizados no módulo *Binq*. Para isso, realizamos diversos testes, de forma a cobrir todas as possibilidades de entrada do programa.

Operadores:

- findmin
- delmin
- merge
- insertBinq

5.1 Operador *findmin*

Para o teste do operador *findmin*, utilizamos um conjunto de 5 árvores de prioridade:

- (NodePQ 2 (NodePQ 3 LeafPQ LeafPQ) (NodePQ 4 LeafPQ LeafPQ))
- (NodePQ 2 (NodePQ 3 LeafPQ LeafPQ) LeafPQ)
- (NodePQ 2 LeafPQ (NodePQ 4 LeafPQ LeafPQ))
- (NodePQ 2 LeafPQ LeafPQ)
- LeafPQ

Os resultados foram:

```

Binq> findmin (NodePQ 2 (NodePQ 3 LeafPQ LeafPQ) (NodePQ 4 LeafPQ LeafPQ))
Just 2
Binq> findmin (NodePQ 2 (NodePQ 3 LeafPQ LeafPQ) LeafPQ)
Just 2
Binq> findmin (NodePQ 2 LeafPQ (NodePQ 4 LeafPQ LeafPQ))
Just 2
Binq> findmin (NodePQ 2 LeafPQ LeafPQ)
Just 2
Binq> findmin LeafPQ
Nothing

```

Os resultados foram os previstos.

5.2 Operador *delmin*

Os mesmos parâmetros utilizados acima também cobrem todas as possibilidades de entrada do operador *delmin*. Vamos aos testes:

```
*Binpq> delmin (NodePQ 2 (NodePQ 3 LeafPQ LeafPQ) (NodePQ 4 LeafPQ LeafPQ))
NodePQ 3 (NodePQ 4 LeafPQ LeafPQ) LeafPQ
*Binpq> delmin (NodePQ 2 (NodePQ 3 LeafPQ LeafPQ) LeafPQ)
NodePQ 3 LeafPQ LeafPQ
*Binpq> delmin (NodePQ 2 LeafPQ (NodePQ 4 LeafPQ LeafPQ))
NodePQ 4 LeafPQ LeafPQ
*Binpq> delmin (NodePQ 2 LeafPQ LeafPQ)
LeafPQ
*Binpq> delmin LeafPQ
LeafPQ
```

Obtemos os resultados corretos.

5.3 Operador *merge*

Aplicamos as diversas variações possíveis para *merge* e, como visto no resultado abaixo, as operações foram concluídas com êxito.

```
*Binpq> merge LeafPQ LeafPQ
LeafPQ
*Binpq> merge LeafPQ (NodePQ 2 LeafPQ LeafPQ)
NodePQ 2 LeafPQ LeafPQ
*Binpq> merge (NodePQ 2 LeafPQ LeafPQ) LeafPQ
NodePQ 2 LeafPQ LeafPQ
*Binpq> merge (NodePQ 2 LeafPQ LeafPQ) (NodePQ 10 LeafPQ LeafPQ)
NodePQ 2 (NodePQ 10 LeafPQ LeafPQ) LeafPQ
*Binpq> merge (NodePQ 10 LeafPQ LeafPQ) (NodePQ 2 LeafPQ LeafPQ)
NodePQ 2 (NodePQ 10 LeafPQ LeafPQ) LeafPQ
*Binpq> merge (NodePQ 2 (NodePQ 3 LeafPQ LeafPQ) LeafPQ) (NodePQ 10 LeafPQ LeafPQ)
NodePQ 2 (NodePQ 10 LeafPQ LeafPQ) (NodePQ 3 LeafPQ LeafPQ)
*Binpq> merge (NodePQ 2 LeafPQ (NodePQ 3 LeafPQ LeafPQ)) (NodePQ 10 LeafPQ LeafPQ)
NodePQ 2 (NodePQ 3 (NodePQ 10 LeafPQ LeafPQ) LeafPQ) LeafPQ
*Binpq> merge (NodePQ 2 (NodePQ 3 LeafPQ LeafPQ) (NodePQ 4 LeafPQ LeafPQ))
(NodePQ 10 LeafPQ LeafPQ)
NodePQ 2 (NodePQ 4 (NodePQ 10 LeafPQ LeafPQ) LeafPQ) (NodePQ 3 LeafPQ LeafPQ)
*Binpq> merge (NodePQ 10 (NodePQ 13 LeafPQ LeafPQ) LeafPQ) (NodePQ 2 LeafPQ LeafPQ)
NodePQ 2 (NodePQ 10 (NodePQ 13 LeafPQ LeafPQ) LeafPQ) LeafPQ
*Binpq> merge (NodePQ 10 LeafPQ (NodePQ 13 LeafPQ LeafPQ)) (NodePQ 2 LeafPQ LeafPQ)
NodePQ 2 (NodePQ 10 LeafPQ (NodePQ 13 LeafPQ LeafPQ)) LeafPQ
*Binpq> merge (NodePQ 10 (NodePQ 13 LeafPQ LeafPQ) (NodePQ 14 LeafPQ LeafPQ))
(NodePQ 2 LeafPQ LeafPQ)
NodePQ 2 (NodePQ 10 (NodePQ 13 LeafPQ LeafPQ) (NodePQ 14 LeafPQ LeafPQ)) LeafPQ
*Binpq> merge (NodePQ 2 LeafPQ LeafPQ) (NodePQ 10 (NodePQ 3 LeafPQ LeafPQ) LeafPQ)
NodePQ 2 (NodePQ 10 (NodePQ 3 LeafPQ LeafPQ) LeafPQ) LeafPQ
*Binpq> merge (NodePQ 2 LeafPQ LeafPQ) (NodePQ 10 LeafPQ (NodePQ 3 LeafPQ LeafPQ))
```

```

NodePQ 2 (NodePQ 10 LeafPQ (NodePQ 3 LeafPQ LeafPQ)) LeafPQ
*Binqp> merge (NodePQ 2 LeafPQ LeafPQ) (NodePQ 10 (NodePQ 3 LeafPQ LeafPQ)
(NodePQ 4 LeafPQ LeafPQ))
NodePQ 2 (NodePQ 10 (NodePQ 3 LeafPQ LeafPQ) (NodePQ 4 LeafPQ LeafPQ)) LeafPQ
*Binqp> merge (NodePQ 10 LeafPQ LeafPQ) (NodePQ 2 (NodePQ 3 LeafPQ LeafPQ) LeafPQ)
NodePQ 2 (NodePQ 10 LeafPQ LeafPQ) (NodePQ 3 LeafPQ LeafPQ)
*Binqp> merge (NodePQ 10 LeafPQ LeafPQ) (NodePQ 2 LeafPQ (NodePQ 3 LeafPQ LeafPQ))
NodePQ 2 (NodePQ 3 (NodePQ 10 LeafPQ LeafPQ) LeafPQ) LeafPQ
*Binqp> merge (NodePQ 10 LeafPQ LeafPQ) (NodePQ 2 (NodePQ 3 LeafPQ LeafPQ)
(NodePQ 4 LeafPQ LeafPQ))
NodePQ 2 (NodePQ 4 (NodePQ 10 LeafPQ LeafPQ) LeafPQ) (NodePQ 3 LeafPQ LeafPQ)
*Binqp> merge (NodePQ 2 (NodePQ 3 LeafPQ LeafPQ) (NodePQ 4 LeafPQ LeafPQ))
(NodePQ 10 (NodePQ 11 LeafPQ LeafPQ) (NodePQ 12 LeafPQ LeafPQ))
NodePQ 2 (NodePQ 4 (NodePQ 10 (NodePQ 11 LeafPQ LeafPQ) (NodePQ 12 LeafPQ LeafPQ))
LeafPQ) (NodePQ 3 LeafPQ LeafPQ)
*Binqp> merge (NodePQ 10 (NodePQ 13 LeafPQ LeafPQ) (NodePQ 14 LeafPQ LeafPQ))
(NodePQ 2 (NodePQ 11 LeafPQ LeafPQ) (NodePQ 12 LeafPQ LeafPQ))
NodePQ 2 (NodePQ 10 (NodePQ 12 (NodePQ 14 LeafPQ LeafPQ) LeafPQ) LeafPQ)
(NodePQ 13 LeafPQ LeafPQ)) (NodePQ 11 LeafPQ LeafPQ)

```

5.4 Operador *insertBinq*

Este operador insere um elemento em uma árvore de prioridade. Seja o seguinte conjunto de entrada:

- LeafPQ 3
- (NodePQ 3 LeafPQ LeafPQ) 10
- (NodePQ 3 (NodePQ 4 LeafPQ LeafPQ) LeafPQ) 10
- (NodePQ 3 LeafPQ (NodePQ 4 LeafPQ LeafPQ)) 10
- (NodePQ 3 (NodePQ 4 LeafPQ LeafPQ) (NodePQ 5 LeafPQ LeafPQ)) 10

Veja os resultados obtidos a partir dessas entradas:

```

*Binqp> insertBinq LeafPQ 3
NodePQ 3 LeafPQ LeafPQ
*Binqp> insertBinq (NodePQ 3 LeafPQ LeafPQ) 10
NodePQ 3 (NodePQ 10 LeafPQ LeafPQ) LeafPQ
*Binqp> insertBinq (NodePQ 3 (NodePQ 4 LeafPQ LeafPQ) LeafPQ) 10
NodePQ 3 (NodePQ 10 LeafPQ LeafPQ) (NodePQ 4 LeafPQ LeafPQ)
*Binqp> insertBinq (NodePQ 3 LeafPQ (NodePQ 4 LeafPQ LeafPQ)) 10
NodePQ 3 (NodePQ 4 (NodePQ 10 LeafPQ LeafPQ) LeafPQ) LeafPQ
*Binqp> insertBinq (NodePQ 3 (NodePQ 4 LeafPQ LeafPQ) (NodePQ 5 LeafPQ LeafPQ)) 10
NodePQ 3 (NodePQ 5 (NodePQ 10 LeafPQ LeafPQ) LeafPQ) (NodePQ 4 LeafPQ LeafPQ)

```

Os resultados foram os esperados.