Universidade Federal
do Espírito Santo

Centro Tecnológico

Departamento de Informática

O Problema do Caixeiro Viajante

Autor:
Vinicius Arruda

Professora:
Mariella Berger

02 de Setembro de 2015

Resumo

Trabalho da disciplina de Estrutura de Dados II, que consiste na implementação de quatro soluções diferentes para o Problema do Caixeiro Viajante. As soluções são os algoritmos de solução ótima, heurística do vizinho mais próximo, heurística de melhoramento 2-opt e heurística do envoltório convexo.

1 Introdução

O Problema do Caixeiro Viajante (do inglês: *Traveling Salesman Problem*, abreviado por TSP) compõe um clássico da carreira de algoritmos, teoria dos grafos, otimização combinatória e tantas outras áreas de estudos computacionais e matemáticos. Além disso, é um problema fascinante e pertence ao seleto grupo de problemas NP-Completo.

O TSP foi inspirado no problema de um vendedor que gostaria de percorrer o menor caminho possível saindo de sua cidade atual, passando por diversas cidades vendendo seus produtos e retornando à cidade de origem. O TSP pode ser aplicado à diversas áreas, desde problemas de roteamento à criptografia.

A idéia central destas implementações de soluções para o TSP é mostrar que não é um problema trivial, pois para encontrar a solução ótima de n cidades é necessário examinar n - *1* rotas distintas.

A complexidade de tempo em busca da solução ótima é fatorial. Se essa complexidade fosse expressa em termos de um polinômio em n, os computadores atuais teriam a capacidade de suportar o aumento de n.

Das soluções, três delas implementam o Problema do Caixeiro Viajante Assimétrico (do inglês: *Asymmetric Traveling Salesman Problem*, abreviado por ATSP) que é uma variante do TSP, com a única diferença que a distância de uma cidade A para a cidade B não é a mesma distância da cidade B para a cidade A. A única solução que implementa o TSP é a heurística do envoltório convexo.

Para a implementação das soluções, foram utilizados estruturas de dados estáticas, como vetor e matriz, e estruturas de dados dinâmicas, como lista encadeada e pilha.

Por questão de simplicidade, este trabalho utilizará a abreviação TSP para se referir à ambos os problemas e quando for necessário enfatizar se há ou não simetria entre as distâncias das cidades, será utilizado *TSP simétrico* e *TSP assimétrico*.

2 Objetivo

Aplicar o conhecimento adquirido na disciplina de Estruturas de Dados II para implementar algoritmos de melhoramento e heurísticas para encontrar soluções aproximadas para o TSP. Adquirir conhecimento da complexidade que é solucionar o problema do TSP e a importância que as heurísticas possuem para a computação.

3 Ferramentas

Os algoritmos foram implementados na linguagem de programação C. Para a compilação foi utilizado o compilador GCC versão 4.7.2 em uma máquina com o sistema operacional Debian GNU/Linux 7.8 (wheezy). O código foi escrito utilizando o editor de texto gedit versão 3.4.2. Para a depuração do programa, foi utilizado a ferramenta Valgrind versão 3.7.0.

4 Algoritmo Exato

O algoritmo *exato* possui solução ótima, porém a complexidade de tempo cresce em escala fatorial em relação ao número de cidades. Sua implementação se baseia em analisar o custo de cada caminho possível, e retornar o caminho que tiver o menor custo.

4.1 Metodologia

O algoritmo *exato* foi implementado utilizando recursão. A figura 1 mostra o processo de geração de caminhos com um exemplo de 4 cidades, partindo da cidade 0.

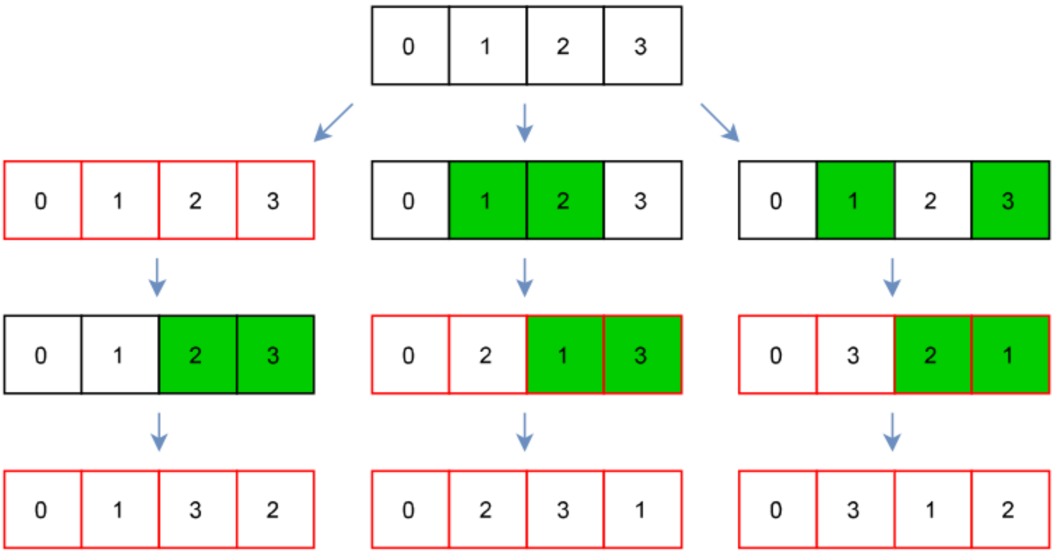

Figura 1: Exemplo de execução da implementação para 4 cidades.

Na figura 1, está marcado em verde as posições do vetor de caminho que irão realizar a troca. Em vermelho estão os diferentes caminhos que foram gerados ao longo da execução. Para cada um destes caminhos, é calculado seu custo, e se for o menor custo encontrado até o momento, este se torna o custo mínimo atual e o menor caminho atual.

Ao executar o algoritmo *exato*, a pilha de execução irá crescer até n chamadas recursivas, onde n é o número de cidades do TSP. Um problema possível de ocorrer é o estouro da pilha de execução caso n seja grande demais, porém, se torna inviável calcular o TSP com um grande número de cidades, pois como já dito anteriormente, o tempo de execução cresce em escala fatorial.

4.2 Avaliação e Resultados

Para a avaliação do algoritmo *exato* foi utilizado a base de dados *br17*. Esta base de dados foi reduzida para as dimensões de 2 à 16, eliminando a última linha e a última coluna para reduzir uma dimensão da base. A avaliação foi feita utilizando a função *time* disponível por linha de comando nos sitemas linux. A tabela 1 mostra o resultado obtido.

Tabela 1: Resultado da execução do algoritmo exato.

Número de cidades	Tempo	Custo do caminho
2	0.412s	10
3	0.432s	11
4	0.424s	104
5	0.424s	104
6	0.416s	70
7	0.416s	36
8	0.416s	39
9	0.468s	39
10	0.668s	39
11	2.888s	39
12	26.470s	39
13	5m 24.864s	39
14	67m 31.385s	39
15	982m 12.383s	39

5 Algoritmo NN

O algoritmo do vizinho mais próximo ou *NN* (abreviação do inglês: *Nearest Neighbor*) pertence à classe dos algoritmos gulosos, e tem como funcionamento incluir ao caminho a cidade mais próxima da atual.

5.1 Metodologia

O algoritmo *NN*, partindo de uma cidade inicial, verifica a distância da cidade atual em relação à todas as cidades ainda não visitadas e inclui em seu caminho a cidade que houver a menor distância. Isso se repete até que todas as cidades estejam no caminho. O algoritmo 1 descreve o algoritmo *NN*.

Algorithm 1 Algoritmo NN

procedure NN
  $j \leftarrow 0$
  $P \leftarrow \varnothing$
  $P_j \leftarrow initialcity$
  $C \leftarrow C - initialcity$
  while $C \neq \varnothing$ **do**
    $min \leftarrow max(distance(P_j, C))$
    for all $e \in C$ **do**
      if $distance(P_j, e) < min$ **then**
        $min \leftarrow distance(P_j, e)$
        $temp \leftarrow e$
      end if
    end for
    $j \leftarrow j + 1$
    $P_j \leftarrow e$
    $C \leftarrow C - e$
  end while
end procedure

5.2 Avaliação e Resultados

Para a avaliação do algoritmo *NN* foi utilizado as bases *ft53.atsp*, *ft70.atsp*, *rbg358.atsp* e *rbg443.atsp*. A avaliação foi feita utilizando a função *time* disponível por linha de comando nos sitemas linux. A tabela 2 mostra o resultado obtido.

Tabela 2: Resultado da execução do algoritmo NN.

Número de cidades	Tempo	Custo do caminho
53	0.000s	9514
70	0.000s	43186
358	0.012s	1812
443	0.024s	3922

6 Algoritmo 2-opt

A heurística de melhoramento *2-opt*, é um algoritmo de otimização que faz uma busca local apenas verificando possíveis trocas de caminhos entre 4 das n cidades do problema.

Neste trabalho, o algoritmo *2-opt* é aplicado sobre um caminho gerado pelo algoritmo *NN*, visando otimizar o caminho encontrado, reduzindo seu custo.

6.1 Metodologia

O algoritmo *2-opt* percorre o caminho das cidades e para cada par de caminho entre uma cidade e outra, ou seja, para cada par de arestas, que não sejam adjacentes, é feita a inversão destes caminhos e verifica se o custo do caminho diminuiu. Ao final de todos os pares de arestas possíveis, realiza-se a inversão de caminhos entre as cidades que obtiveram menor custo.

6.2 Avaliação e Resultados

Para a avaliação do algoritmo *2-opt* foi utilizada as mesmas bases utilizada para o algoritmo *NN*. A avaliação foi feita utilizando a função *time* disponível por linha de comando nos sitemas linux. A tabela 3 mostra o resultado obtido.

Tabela 3: Resultado da execução do algoritmo 2-opt.

Número de cidades	Tempo	Custo do caminho
53	0.008s	8833
70	0.016s	42490
358	3.596s	1774
443	3.460s	3892

Comparando a tabela 2 com a 3, notamos que houve uma melhora do custo do caminho encontrado aplicando o algoritmo *2-opt* em uma solução do algoritmo *NN* para as bases de testes acima.

7 Algoritmo Convex Hull

O algoritmo *Convex Hull*, é um algoritmo de aproximação que gera uma solução para o TSP em dois passos. O primeiro passo é gerar o envoltório convexo, que é um caminho entre as cidades mais externas de maneira que englobe todas as que não estejam no caminho do envoltório. O segundo passo, é apartir do caminho gerado pelo envoltório convexo, incluir as cidades que não estão no envoltório de maneira que o custo do desvio do caminho para passar nesta cidade seja mínimo.

7.1 Metodologia

O algoritmo *Convex Hull* possui várias maneiras de ser implementado. A implementação deste trabalho segue os seguintes passos:

1. Considerando as cidades como pontos no plano cartesiano, ordena esses pontos em ordem crescente em relação ao eixo x.

2. Cria a parte superior do envoltório convexo.

3. Cria a parte inferior do envoltório convexo.

Com o envoltório pronto, segue a construção do caminho completo:

1. Coloca as cidades que estão no envoltório convexo no caminho.

2. Para cada cidade que não esteja no caminho (*out*), calcula-se o custo do desvio: *dist(in[i], out) + dist(out, in[i+1]) - dist(in[i], in[i+1])*, onde *in[i]* é a *iésima* cidade pertencente ao caminho.

3. A cidade que possuir o menor custo de desvio é incluída no caminho.

4. Enquanto houver cidade fora do caminho, repete o passo 2 e 3.

7.2 Avaliação e Resultados

Para a avaliação do algoritmo *Convex Hull* foi utilizado a base berlin52.tsp. A avaliação foi feita utilizando a função *time* disponível por linha de comando nos sitemas linux. A tabela 4 mostra o resultado obtido.

Tabela 4: Resultado da execução do algoritmo Convex Hull.

Número de cidades	Tempo	Custo do caminho
52	0.004s	8430.44

8 Referências Bibliográficas

1. http://www.mat.ufrgs.br/~portosil/caixeiro.html
2. https://pt.wikipedia.org/wiki/Complexidade_de_tempo
3. http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/atsp/
4. http://tex.stackexchange.com
5. http://www.inf.ufes.br/~mberger/Disciplinas/2015_2/EDII/trab1.pdf
6. http://www.inf.ufes.br/~mberger/Disciplinas/2015_2/EDII/03.pdf
7. https://en.wikibooks.org/wiki/LaTeX/Algorithms