UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

Trabalho de Linguagens de Programação II

Soluções para o TSP

Autor: André Ambrósio Boechat

boechat107@gmail.com
Engenharia de Computação

02 de Junho de 2008

1 Introdução

O problema tratado neste trabalho é o TSP (*Travelling Salesman Problem*). Deseja-se, através da solução desse problema, estudar as variações de *binary tree* e suas aplicações como estruturas de indexação e *heap*. Dois algoritmos serão usados para a construção do *tour*: o *Furthest Insertion* e o *Double Minimum Spanning Tree* (*DMST*).

2 Descrição do Problema

Dado um conjunto de pontos no plano Euclidiano de duas dimensões, o problema consiste em encontrar o menor *tour* possível que passe por todos os pontos apenas uma vez e volte ao ponto inicial (veja a Figura 1). A distância entre dois pontos quaisquer é dada pela distância Euclidiana.

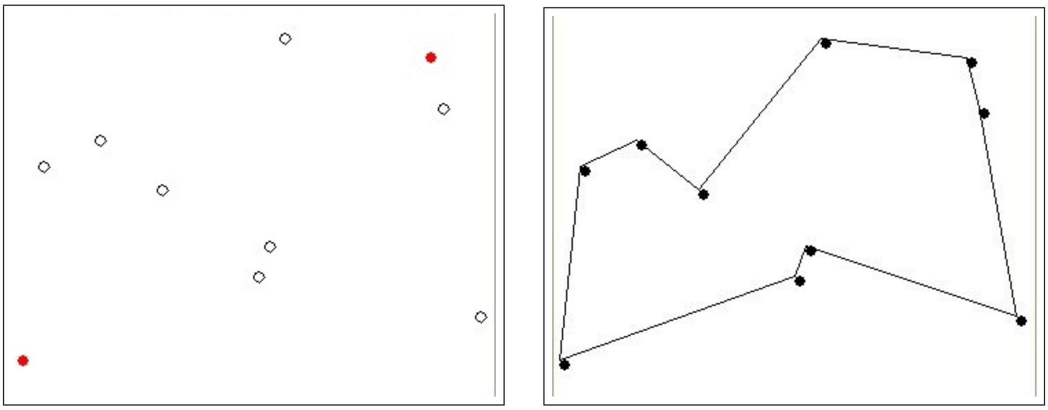

Figura 1: Construção de um *tour* que visita todos os pontos do plano Euclidiano com o menor comprimento possível.

3 Propostas de Implementação e Conjecturas Sobre Eficiência

A primeira etapa do trabalho consiste em propor as estruturas de dados e os algoritmos a serem utilizados no desenvolvimento das duas soluções, *Furthest Insertion* e *DMST*, além conjecturar sobre a eficiência de tais estruturas.

3.1 Furthest Insertion

A heurística do *Furthest Insertion* consiste em duas ações básicas:

- Procura do vértice mais distante ao *tour*;
- Inserção desse vértice no *tour* de forma que gere o menor caminho possível.

A distância de um vértice livre ao *tour* é a distância entre esse vértice e o ponto do *tour* mais próximo dele. Para isso, um algoritmo semelhante ao *Nearest Neighbour* pode ser usado. Assim, deve-se calcular a distância ao *tour* para cada vértice livre e selecionar aquele que apresentar a maior distância.

De acordo com a heurística, o vértice de maior distância deve ser inserido no *tour* da seguinte forma: supondo C a distância entre dois vértices, i e j como vértices já pertencentes ao *tour* e r como o vértice livre selecionado anteriormente, deve-se inserir r no *tour* de forma que minimize a expressão $C_{ir} + C_{jr} - C_{ir}$.

Com o objetivo de dominar integralmente o problema e comparar o desempenho entre diferentes estrutura de dados, pretende-se implementar duas versões diferentes para o *Furthest Insertion*, cada uma utilizando uma estrutura de dados diferente.

3.1.1 Versão Base

A primeira versão, que também pode ser considerada a versão base, é implementada utilizando-se apenas listas simples como estruturas de indexação espacial e *heap*, ou seja, tanto os vértices livres[1] quanto os pertencentes ao *tour* são armazenados em listas.

Essa implementação apresenta baixa eficiência, pois quase todas as operações sobre as listas implicam na necessidade varrê-las totalmente, aumentando consideravelmente a ordem de complexidade do algoritmo.

A procura pelo vértice mais distante ao *tour*, utilizando lista, apresenta o custo de $O(n^2)$. Então, como o *loop* principal do *Furthest Insertion* é executado até que não existam mais vértices livres, o que o torna $O(n^3)$. Porém, se, ao procurar o vértice mais distante, as distâncias entre os vértices livres ao *tour* for atualizada levando em consideração apenas o último vértice inserido no *tour*, o custo do algoritmo cai para $O(n^2)$.

3.1.2 Versão Mais Eficiente

A segunda versão utiliza uma *B-tree* como fila de prioridade e uma *Kd-tree* como estrutura de indexação espacial, ou seja, os vértices livres são armazenados na *B-tree* e o *tour* é armazenado na *Kd-tree*.

A vantagem da utilização de uma *B-tree* para armazenamento dos vértices livres é possibilidade de indexá-los, em relação às suas respectivas distâncias ao *tour*, em uma árvore que se mantem sempre balanceada, o que agiliza a busca pelo vértice mais distante. Isso vem do fato de que, sempre que for criada uma nova aresta do *tour*, a distância entre os vértices livres e o *tour*só pode permanecer a mesma ou diminuir. Assim, pode-se verificar se a distância do vértice mais distante (operação que custa $O(\log n)$) sofreu alguma alteração; caso isso aconteça, a distância vértice deve ser recalculada, o vértice reinserido na árvore e repetir a busca pelo mais distante;caso contrário, pode-se afirmar que esse vértice continua sendo o mais distante, o que evita a atualização da distância dos outros vértices.

[1] Vértice que ainda não pertence ao *tour*.

Pelas suas características, a *Kd-tree* apresenta-se como uma estrutura eficiente para agilizar a procura pela melhor posição de inserção no *tour*. Visualmente, pode-se dizer que a *Kd-tree* organiza os vértices nela inseridos em retângulos, utilizando a coordenada x para separá-los em direita-esquerda e a coordenada y em abaixo-acima. Assim, a organização da árvore possibilita resgatar facilmente as posições relativas entre os vértices.

Assim como a versão básica, o *loop* principal da segunda versão também é executado até que não exsitam mais vértices livres. A diferença de eficiência entre as duas implementações ocorre, obviamente, na busca e inserção do vértice mais distante. Se a atualização da distância de um vértice livre ao *tour* resumir-se a verificar apenas a distância ao último vértice inserido no *tour*, a *B-tree* possibilita que a busca pelo vértice mais distante despenda $O(\log n)$. Já a *Kd-tree*, por dividir o plano Euclidiano em subdomínios, possibilita que uma inserção no *tour* seja executada em $O(\log n)$. Dessa forma, o algoritmo do *Furthest Insertion* seria executado com a ordem de complexidade $O(n \log n)$.

3.2 DMST

Para esse algoritmo, são implementados duas versões diferentes: uma variação do algoritmo de Kruskall e uma variação do algoritmo de Djikstra.

3.2.1 Variação do Algoritmo de Kruskall

A estrutura de dados a ser usada para armazenar o conjunto de arestas é uma *B-tree*.

3.2.2 Variação do Algoritmo de Djikstra

A estrutura de dados a ser utilizada para os vértices livres é uma *B-tree*, enquanto que os vértices já visitados são armazenados em uma *Kd-tree*.