Documentação TP2 - Soluções para Problemas Difíceis

Isadora Alves de Salles

01/12/2019

1 Introdução

O objetivo deste trabalho é implementar soluções exatas e aproximadas para o problema do caixeiro viajante, que é um problema NP-difícil. O problema do Caixeiro Viajante, ou TSP (*Travelling Salesman Problem*) é um dos problemas mais tradicionais em Ciência da Computação e consiste na busca por um caminho de menor custo possível que passe por todas as cidades de um conjunto de n cidades e retorne ao ponto inicial.

A figura 1 mostra a representação de uma solução para um conjunto qualquer de cidades.

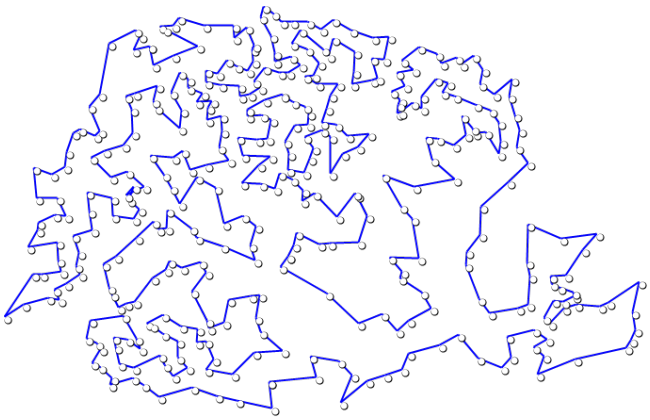

Figura 1: Exemplo TSP

2 Formalização do Problema

O mapa das cidades a serem visitadas pelo caixeiro pode ser modelado a partir de um grafo direcionado $G(V, E)$, onde $V \in G$ é o conjunto de vértices do grafo (cidades) e $E \in G$ o conjunto de arestas (trechos que ligam as cidades). A ligação entre duas cidades i e j pode ser representada pelo par (i, j).

Cada aresta possui um custo c_{ij} associado a ela, que não necessariamente é o mesmo que c_{ji}. O caminho entre uma cidade A e uma cidade B, por exemplo, pode ter diferentes valores se consideradas diferentes direções.

Existem as versões do TSP métrico e assimétrico. Neste trabalho, iremos utilizar apenas instâncias do TSP métrico, que é o caso em que as distâncias entre as cidades formam um espaço métrico (são simétricas e obedecem a desigualdade triangular), ou seja, o custo c_{ij} será o mesmo que c_{ji} e teremos um grafo completo, existe uma aresta de qualquer vértice para todos os outros.

3 Solução exata - Branch-and-bound

A adaptação do algoritmo de branch-and-bound para o problema do caixeiro viajante é direta do algoritmo de backtracking para computação do circuito hamiltoniano. Incluindo apenas uma forma de podar ramos que gerem caminhos mais custosos.

Uma forma de estimar se um ramo levará para um caminho mais custoso é a seguinte: como devemos sempre entrar e sair de um vértice podemos usar a média da soma das duas arestas de menor peso incidentes a cada vértice como uma estimativa para podar a árvore.

Tendo essa estimativa podemos avaliar se compensa ou não estender um determinado ramo. Por se tratar de um problema de minimização sabemos que o valor da estimativa sempre aumenta, então temos que avaliar apenas os ramos com as menores estimativas. Dessa forma mesclamos uma busca em largura e profundidade, nunca aprofundamos uma solução por completo se existir um outro ramo com melhor estimativa para ser avaliado. Assim, evitamos o método de backtracking que realiza várias buscas em profundidade de forma exaustiva. A estrutura de dados utilizada para isso é uma fila de prioridades, um heap, em que a prioridade é a menor estimativa. Assim, o que está no início da fila sempre será o node de menor bound (ou estimativa). Dessa forma, percorremos os nodes adicionados ao heap retirando um a um até que o heap esteja vazio. A condição para que um node sejá adicionado ao heap é que sua estimativa de custo seja inferior ao melhor custo, dessa forma podamos soluções que só tem como ser piores que a melhor encontrada. O melhor custo é inicialmente infinito e é atualizado quando encontra-se uma folha.

3.1 Análise de Complexidade

O algoritmo de branch-and-bound continua tendo complexidade exponencial no tempo, pois no pior caso temos que analisar todos os ramos, mas na prática é bem mais eficiente que o backtracking pois conseguimos devido as podas.

Em relação ao custo de espaço, temos que a estrutura do heap que armazena a estimativa, o nível do node na árvore, o custo da solução e o vetor com a solução. O custo de armazenar o heap irá variar de acordo com a quantidade de podas realizadas pelo algoritmo. Mas considerando o pior caso em que analisamos todas as possibilidades, o custo do heap será exponencial. Além disso, temos que armazenar um grafo completo e isso terá custo $O(n^2)$, sendo n o número de vértices. Assim, o custo espacial será limitado pelo custo do heap, logo é exponencial.

4 Soluções aproximadas

4.1 Algoritmo Twice-around-the-tree

Uma boa aproximação para o custo de um caminho mínimo é o custo da árvore geradora mínima do grafo. Uma MST é composta pelas arestas de menor peso do grafo, sabemos que pelo menos algumas dessas arestas devem estar no nosso conjunto solução para o problema do TSP. Então, computar a MST seria uma boa aproximação para o TSP, principalmente ao se comparar com a heurística gulosa do vizinho mais próximo. Esse algoritmo terá um fator de aproximação igual a 2 e a seguir podemos ver o pseudocódigo para ele:

Algorithm 1: Twice-around-the-tree

select a vertex $v \in G$ to be a root vertex;
compute a minimum spanning tree T for G from root v;
$H \leftarrow$ list of vertices in pre order walk of T;
return hamiltonian cycle(H);

O algoritmo acima é bem simples fazendo o uso de dois algoritmos em grafos: Prim, para encontrar a árvore geradora mínima, e caminhamento em pré-ordem que nos da um ciclo hamiltoniano válido no grafo. Foi usada a biblioteca networkx do Python que possui essas duas funções já implementadas, bem como um tipo abstrato de dados pronto para manipular o grafo. Após computar a MST, obtemos o circuito hamiltoniano a partir do circuito euleriano no multigrafo com as arestas da árvore geradora mínima duplicadas, fazendo o caminhamento em pre-ordem na árvore.

4.1.1 Análise de Complexidade

Seja m o número de arestas e n o número de vértices, temos que:

A complexidade assintótica de tempo do algoritmo descrito é limitada pela complexidade do algoritmo de Prim, que é $O(mlogn)$, visto que o caminhamento em pré-ordem tem custo $O(n)$. No caso como sempre estamos testando em grafos completos, a complexidade passa a ser $O(n^2)$. Isso será demonstrado na análise experimental.

Já a complexidade assintótica de espaço do algoritmo será dada pelo espaço gasto para armazenar o grafo, que como dito anteriormente, será um grafo completo, então temos que o custo será $O(n^2)$.

4.2 Algoritmo de Christofides

O algortimo de Christofides é uma heurística ainda melhor que o twice-around-the-tree para o caixeiro viajante, possuindo fator de aproximação de 1.5. Ele parte da ideia de transformar um ciclo euleriano em um ciclo hamiltoniano, porém ao invés de utilizar apenas as arestas da árvore geradora mínima para isso fazemos o seguinte:

1. Encontramos a árvore geradora mínima e selecionamos os vértices de grau ímpar dessa árvore;
2. Encontramos um matching perfeito de peso mínimo com os vértices selecionados;
3. E construímos um multigrafo com as arestas da MST e do matching, a partir disso iremos montar o circuito euleriano.

Sabemos que o número de vértices de grau ímpar em uma árvore tem que ser par, dessa forma garantimos que o matching perfeito existe. Assim, será possível percorrer todas as arestas do grafo e retornar para o ponto de início pois formaremos ciclos. Nenhum vértice terá grau ímpar, pois adicionamos uma aresta a cada dois vértices de grau ímpar. O pseudocódigo para esse algoritmo está a seguir:

Algorithm 2: Algoritmo de Christofides

$T \leftarrow$ minimum spanning tree T for G;
$O \leftarrow$ set of vertices of odd degree of T;
$S \leftarrow$ subgraph(G, O);
$M \leftarrow$ minimum weight perfect matching (S);
$G' \leftarrow$ multigraph(T, M);
$E \leftarrow$ compute eulerian cycle (G');
return hamiltonian cycle(E);

Ter implementado esse algoritmo em Python foi bastante prático por poder utilizar as funções da biblioteca networkx. A questão é que não existe o algoritmo que calcula o matching perfeito de peso mínimo na networkx, porém existe um que calcula o matching perfeito de peso máximo. Sendo assim, encontramos a aresta de maior peso no grafo, depois subtraímos essa aresta de todas as outras, e calculamos o matching perfeito de peso máximo sobre esse grafo manipulado. Isso irá retornar as arestas que compom, no grafo original, o matching perfeito de peso mínimo. Todas as outras funções foram utilizadas diretamente da biblioteca.

4.2.1 Análise de Complexidade

Seja n o número de vértices e m o número de arestas, temos que:

A complexidade assintótica de tempo será limitada pelo algortimo de Blossom, que é o algoritmo para calcular o matching perfeito de peso máximo, que tem custo $O(n^3)$. Todos os outros algoritmos utilizados tem complexidade inferior a esta.

Já a complexidade assintótica de espaço, será equivalente a do algoritmo twice around the tree. Mesmo que no algortimo de Christofides criamos um novo subgrafo, e até mesmo um multigrafo, todas as estruturas criadas armazenam menos elementos no espaço, seja menos vértices ou menos arestas, assim o custo de espaço será dado por $O(m^2)$ que é o custo do grafo original.

5 Avaliação Experimental

5.1 Instâncias

Neste trabalho estamos tratando especificamente do TSP métrico, que é um caso especial do TSP, em que os pesos das arestas satisfazem a desigualdade triangular. Assim, para gerar as instâncias, foram gerados pontos no plano e o peso das arestas foi dado pelo cálculo da distância Euclidiana e da distância de Manhattan.

O gerador de instâncias receberá três parâmetros: lower, upper e size, sendo lower e upper os limites para gerar os pontos, e size variando de [4, 10], as instâncias geradas terão tamnaho 2^{size}.

5.2 Ambiente Computacional

A implementação do algoritmo foi feita utilizando-se a linguagem Python3 e o tempo de execução foi computado com a função time() da biblioteca time do Python. Os experimentos foram executados em um computador de 12Gb de RAM, processador intel core i7, 1.80GHz com 8 núcleos num sistema operacional Ubuntu 16.04 LTS.

5.3 Resultados

Para avaliação experimental foram geradas instâncias de tamanho 2^4 até 2^{10}, e todos os algoritmos foram executados, para as duas métricas de distância, para todas as instâncias. A seguir temos os resultados dos experimentos realizados e algumas análises.

O tempo de execução dos algoritmos foi limitado a 20 minutos, ao passar de 20 minutos a execução do algoritmo é abortada e os dados referentes ao custo da solução são listados como 'NA' na tabela, e o tempo como 'Limite_exceed'. Abaixo segue um exemplo de um pedaço da tabela gerada, não será mostrada toda a tabela neste relatório, as análises serão apresentadas através de gráficos para facilitar a visualização.

Algorithm	Length	Cost	Time(s)	Metric
Twice around the tree	16	70.77212	0.00072	Euclidean
Twice around the tree	16	114	0.00059	Manhattan
Christofides	16	66.40269	0.00189	Euclidean
Christofides	16	94	0.00229	Manhattan
Branch-and-Bound	16	66.40269	7.04398	Euclidean
Branch-and-Bound	16	84	22.9330	Manhattan
Twice around the tree	32	131.22910	0.00226	Euclidean
Twice around the tree	32	158	0.00213	Manhattan
Christofides	32	110.95226	0.02930	Euclidean
Christofides	32	140	0.01844	Manhattan
Branch-and-Bound	32	NA	Limit_exceed	Euclidean
Branch-and-Bound	32	NA	Limit_exceed	Manhattan

Tabela 1: Exemplo dos resultados

Na Tabela 1 podemos perceber que a execução do branch-and-bound excedeu ao limite de tempo para uma instância de 32 vértices. Isso ocorreu para todas as instâncias com número de vértices maior ou igual a 32 vértices. O motivo disso é o fato de que o branch-and-bound tem custo exponencial, e um aumento pequeno no tamanho da instância pode gerar um custo de tempo muito grande. Sendo assim, não é possível analisar muita coisa para o algoritmo de branch-and-bound com esses primeiros resultados. Porém, esses resultados foram plotados em gráficos para demonstrar a complexidade de tempo dos outros dois algoritmos. Os gráficos podem ser vistos a seguir:

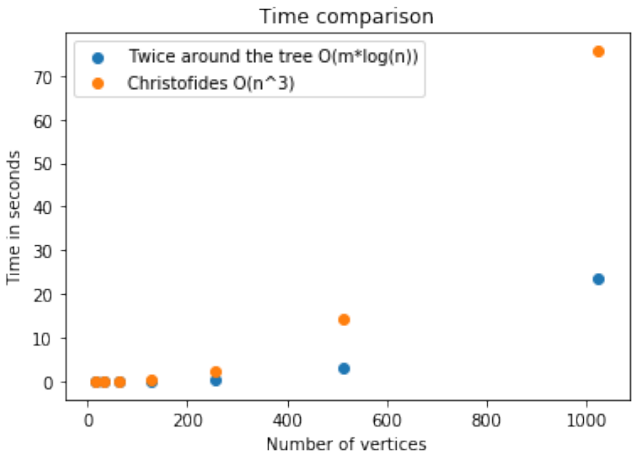

Figura 2: Complexidade assintótica para a distância euclidiana

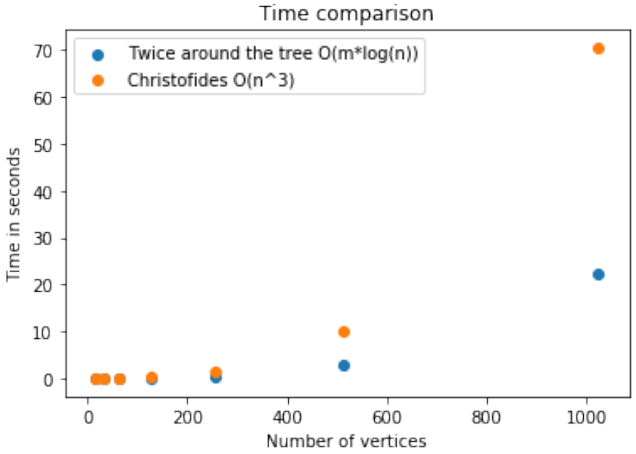

Figura 3: Complexidade assintótica para a distância de manhattan

Podemos perceber com os gráficos 2 e 3 que a complexidade relatada anteriormente no relatório é comprovada. Ou seja, que o Christofides tem complexidade $O(V^3)$ e o twice around the tree tem complexidade $O(E * logV)$. Note que o fato de alterarmos a métrica de distância não muda o caráter assintótico dos algoritmos, e o tempo de execução para as duas métricas foi bastante similar.

Como o branch-and-bound só conseguiu encontrar uma solução em até 20 minutos para instâncias de 16 vértices, decidi gerar várias instâncias de 16 vértices e analisar o comportamento dos três algoritmos nessas instâncias. Assim, foram geradas 15 instâncias aleatórias com 16 vértices, e extraímos os seguintes dados em relação ao tempo de execução:

Algorithm	Mean	Std	Metric
Branch-and-Bound	333.44858	242.07144	Euclidean
Twice around the tree	0.00071	0.00016	Euclidean
Christofides	0.00456	0.00189	Euclidean
Branch-and-Bound	380.13437	371.82645	Manhattan
Twice around the tree	0.00064	0.00011	Manhattan
Christofides	0.00437	0.00137	Manhattan

Tabela 2: Média e desvio padrão do tempo de execução

Note que para a métrica Manhattan o branch-and-bound demora mais tempo para executar, na média. já para os outros dois algoritmos a diferença é insignificante.

Além disso, foi calculada a média e o desvio padrão do custo encontrado pelos algoritmos aproximativos em relação a solução ótima dada pelo branch-and bound. Esses resultados podem ser vistos na tabela a seguir:

Algorithm	Mean	Std	Metric
Twice around the tree	12.97305	7.18464	Euclidean
Christofides	4.36902	2.81242	Euclidean
Twice around the tree	18.13333	7.94505	Manhattan
Christofides	6.0	5.23723	Manhattan

Tabela 3: Média e desvio padrão da solução em relação ao custo ótimo

Note que na Tabela 3 comprovamos que a solução encontrada pelo algoritmo de Christofides está mais próxima, na média, que a solução encontrada pelo twice around the tree. Esse resultado era esperado devido ao fator de aproximação dos algoritmos. Também podemos notar na tabela que a solução está mais distante do ótimo para a métrica de Manhattan.

6 Conclusão

Com este trabalho foi possível analisar o comportamento assintótico de três algoritmos para resolver o problema do caixeiro viajante. Com a primeira análise experimental conseguimos provar a complexidade dos algoritmos. Já na segunda análise, vimos que o branch-and-bound tem um comportamento exponencial, assim não é possível definir o quanto uma modificação no tamanho da instância ou métrica utilizada pode interferir no desempenho final. Nota-se também que na segunda análise temos o desvio das soluções encontradas pelos algoritmos aproximativos para a solução ótima dada pelo branch-and-bound.

O branch-and-bound implementado foi comparado com o algoritmo de branch-and-cut implementado para a disciplina Pesquisa Operacional, e assim foi possível saber se o algoritmo estava correto e que os valores encontrados eram realmente o ótimo.

Além disso, com esse trabalho foi possível aprender maneiras de resolver problemas difíceis de forma aproximada e ótima, e determinar para quais condições é possível resolver problema desse tipo na otimalidade.