Trabalho Prático II
Soluções para problemas difíceis

Heitor Henrique dos Santos[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

Resumo. *O trabalho tem como propósito implementar e avaliar o desempenho de algoritmos voltados para a resolução do Problema do Caixeiro Viajante Euclidiano que é NP-Difícil. Especificamente, optou-se pela técnica de branch and bound para obter a solução ótima, além das estratégias Twice Around The Tree e a heurística de Christofides para alcançar soluções aproximadas.*

1. Introdução

O **Problema do Caixeiro Viajante (TSP)**[1] é um dos problemas mais conhecidos na teoria dos grafos e otimização combinatória. No TSP, dado um grafo $G(V, E)$, um caixeiro viajante deseja encontrar o menor circuito que passe por todas os vértices exatamente uma vez.

Nesse trabalho, estaremos interessados apenas no caso particular do problema em que a função de custo das arestas do grafo é uma métrica:

$$c(u, v) \geq 0 \quad \text{(Positivamente Definida)}$$
$$c(u, v) = 0 \quad \text{sse } u = v \quad \text{(Identidade)}$$
$$c(u, v) = c(v, u) \quad \text{(Simetria)}$$
$$c(u, v) \leq c(u, w) + c(w, v) \quad \text{(Desigualdade Triangular)}$$

Essa versão do problema é conhecida como **Problema do Caixeiro Viajante Euclidiano**. Dessa forma, quando os vértices são cidades representadas por pontos no plano, o peso da aresta entre eles é determinado pela distância euclidiana entre os dois pontos que representam cada vértice no plano dada pela fórmula:

$$c(u, v) = \sqrt{(x_u - x_v)^2 + (y_u - y_v)^2}$$

Assim como o problema original, o Problema do Caixeiro Viajante Euclidiano também é NP-Difícil. Diante disso, foi implementado[1] algoritmos aproximativos e exatos, como o **Twice Around The Tree** e **Christofides**, para obter soluções próximas da ótima em tempo polinomial. E um algoritmo utilizando a técnica de **Branch and Bound**, a fim de obter a solução ótima.

2. Implementação

Os algoritmos foram implementados em Python 3.12.0, executados no sistema operacional Windows 10. A construção de grafos e algoritmos relacionados a grafos, especialmente nos algoritmos aproximativos, foi facilitada pelo uso da biblioteca Networkx[2]. No caso específico do algoritmo Branch and Bound, o grafo é tratado como uma matriz de distâncias.

[1]`https://github.com/heitoeu/TP2_Algoritmos_2`

[2]`https://networkx.org/documentation/latest/index.html#`

2.1. Branch and Bound

Foi implementado um algoritmo com a técnica de Branch and Bound a fim de podar alguns ramos da simulação de uma Máquina de Turing não-deterministica na tentativa de obter a solução ótima de forma mais eficaz do que a solução ingênua, que testa todos os ramos.

Para isso, foi criado uma função que calcula a lowerbound de cada nó da computação com o objetivo de fazer uma busca best first, explorando os nós frutiferos como prioritariamente. Essa função usa como critério de ordem o seguinte. O primeiro nó possui apenas o vértice de partida e sua lower bound é dada pela soma das duas menores arestas de todos os vértices do grafo divido por dois. Em seguida, os nós filhos serão todos os possíveis primeiros caminhos a partir do primeiro vértice, a estimativa deles será a estimativa do nó pai mas agora forçando essa aresta vinculada ao novo vértice adicionado como no exemplo a seguir[2].

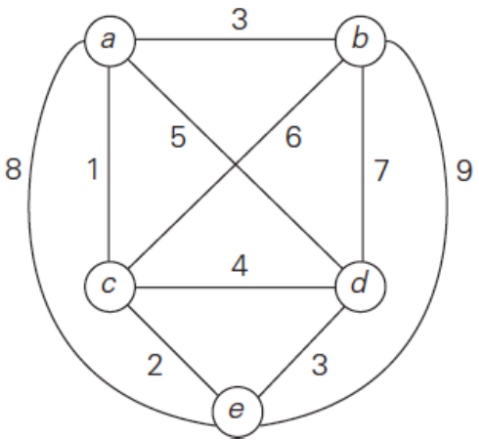

Figure 1. Grafo utilizado para o exemplo BnB

Começando pelo vértice a sua estimativa será de 14, enquanto o nó fillho de vértices ad terá estimativa de 16 com o peso da aresta $a - d$ forçado nas arestas de peso 3 de a e de peso 4 de b:

$$\text{ceil}\left(\frac{(1+3)+(3+6)+(1+2)+(4+3)+(2+3)}{2}\right) = 14$$

$$\text{ceil}\left(\frac{(1+5)+(3+6)+(1+2)+(5+3)+(2+3)}{2}\right) = 16$$

No algoritmo, isso é feito da seguinte forma, se a aresta forçada não faz parte das duas menores arestas de a a nova estimativa será de:

$$\frac{2 \times \text{estimativa\_pai} + \text{aresta\_forçada} - \max(\text{menores\_arestas}(a))}{2}$$

Essa mesma verificação é feita em relação a b mas agora sobre o valor resultante da equação anterior. E então retornamos o valor que passar por essas duas condições. Isso é o mesmo que forçar as arestas.

Assim é possível criar um algoritmo Branch and Bound para o TSP que funciona da seguinte forma. Primeiramente, é fixado um nó de partida, como a solução se trata de um circuito essa ação poda os nós que levam para soluções equivalentes. De forma análoga é fixado que o vértice 1 sempre deve vir primeiro que o vértice 2, dessa forma, os caso de circuitos equivalentes em relação a orientação também são podados.

Após isso, os nós são explorados segundo a profundidade e em caso de empate o nó de menor lower bound terá prioridade isso tudo monitorado por uma fila de prioridades. Além disso, serão explorados apenas os nós que tiverem uma estimativa menor do que o atual melhor caminho. Sempre que um nó folha é detectado temos um circuito completo, se o custo dele for menor do que o custo do atual melhor caminho, então o atual melhor caminho será subsituido por esse novo custo. O algoritmo para quando todos os possíveis nós frutiferos foram explorados, retornando a solução ótima para o TSP.

2.2. Twice Around The Tree

A ideia desse algoritmo é usar o custo da árvore geradora mínima do grafo como uma aproximação. Ele funciona da seguinte forma, inicialmente o algoritmo usa uma função pronta da Networkx para obter a árvore geradora mínima a partir do grafo de entrada. Em seguida, basta computar o caminho preorder na árvore obtida a partir da raiz, aqui escolhemos o vértice 1 como raiz já que a identação do dataset utilizado começa por esse mesmo índice. Esse caminho na árvore retornará um caminho que passa por todos os vértices restando apenas o caminho que liga o último vértice ao vértice de partida.

É importante ressaltar que o preorder não computa os retornos a vértices que já foram visitados, então, considerando a árvore geradora mínima na Figura 2 do grafo mostrado anteriormente, o caminhamento preorder passa pelos vértices na seguinte sequência $abcbdedba$. Mas note que ele apenas computa a passagem por $abcde$ que é o mesmo que usar uma rota que corta caminho usando as arestas cd e ea. Como o custo das arestas do grafo é uma métrica, esse caminho é mais barato do que retornar ao vértice pai antes de ir para o próximo vértice. Em seguida o algoritmo computa o custo da rota obtida adicionado o vértice de partida oa final do caminho.

Esse algoritmo foi demonstrado ser polinomial e 2-aproximado para o TSP Euclidiano, o que pode ser bastante útil para usos práticos, já que vários problemas do mundo real atendem as restrições dessa versão do problema.

2.3. Christofides

Esse algoritmo é semelhante ao algoritmo usando Twice Around the Tree. Assim como nele, uma árvore geradora mínima do grafo de entrada é computada. Em seguida é utilizada uma função da Networkx para obter o matching perfeito de peso mínimo no subgrafo induzido pelos vértices de grau ímpar da árvore. As arestas desse matching são adicionadas a árvore geradora mínima. Por fim, como no algoritmo anterior computamos o caminho utilizando uma DFS preorder fornecida também pela Networkx. Essa forma de caminhar serve para transformar o circuito Euleriano em um circuito Hamiltoniano de forma implícita.

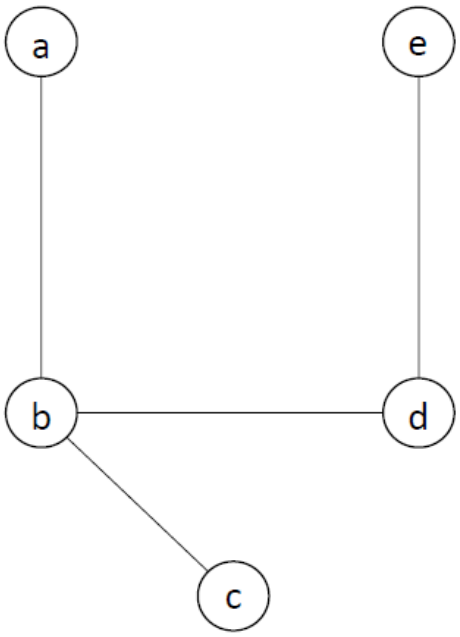

Figure 2. Árvore Geradora Mínima do Grafo da Figura 1

Esse algoritmo foi demosntrado ser polinomial e 1.5-aproximado para o TSP-Euclidiano. Uma melhora significativa proposta por Nicos Christofides que notou que as arestas problemáticas para o algoritmo anterior eram aquelas vinculadas aos vértices de grau ímpar, que levam a caminhos sem entrada ou sem saída. Por esse motivo o matching é feito no grafo induzido por esses vértices, proporcionando novas arestas no grafo final onde o circuito será computado.

3. Experimentos

Os experimentos foram realizados sobre as instâncias cuja função de custo é a distância euclidiana em 2D fornecidos pela bilioteca TSPLIB[3]. Os critérios utilizados para a avaliação de cada algoritmo foram tempo, espaço e qualidade da solução.

Caso o algoritmo demore mais de 30 min para parar, é retornado NA para dados não coletados. Além disso um instância própria $a0.tsp$ foi usada para conseguir coletar dados do Algoritmo Branch and Bound, que não para em tempo hábil para nenhuma das instâncias da TSPLIB. Os resultados dos experimentos para algumas instâncias foram exportados de CSV para a tabela a seguir para avaliarmos os desempenhos.

3.1. Branch and Bound

Ficou claro que mesmo a técnica de Branch and Bound ainda tem um custo considerável, e o pior dos casos continua sendo $O(n!)$ (o caso em que as estimativas aparecem em ordem decrescente). Com os experimentos o algoritmo rodou em tempo hábil somente para um grafo de 12 vértices criado separadamente. O algoritmo não retorna em menos de uma hora para nenhuma das instâncias da TSPLIB.

[3] `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/`

3.2. Twice Around the Tree e Christofides

Em termos de memória, de acordo com as medições, ambos os algoritmos gastaram uma quantidade semelhante. Já em tempo o algoritmo de Christofides superou o Twice Around the Tree, calcular o matching perfeito de peso mínimo atrasou o algoritmo ao ponto que para a instância $fl3795$ ele não parou em menos de uma meia hora de espera. O preço desse tempo é justificado por uma solução com a qualidade mais próxima da solução ótima do que a obtida pelo Twice Around the Tree para a maioria das instâncias.

Outro ponto interessante de se notar é que a qualidade de nenhum deles ultrapassou $1.5 \times$ Solução_Ótima. Esse resultado era esperado para o algoritmo de Christofides, visto que já foi demonstrado que ele é 1.5-aproximado. No entanto, para o Twice Around The Tree era esperado uma qualidade um pouco mais próxima de $2 \times$ Solução_Ótima,o que não foi observado para as instâncias fornecidas.

Algorithm	Instance	Nodes	Limiar	Cost	Quality	Time	Mb
Branch and Bound	a0	12	4057	4057	1.0	1.03	0.02
Twice Around the Tree	a0	12	4057	4558	1.12	0.00	0.04
Christofides	a0	12	4057	4400	1.08	0.00	0.06
Branch and Bound	eil51	51	426	NA	NA	NA	0.39
Twice Around the Tree	eil51	51	426	612	1.44	0.01	0.14
Christofides	eil51	51	426	589	1.38	0.03	0.14
Branch and Bound	rd400	400	15281	NA	NA	NA	96.64
Twice Around the Tree	rd400	400	15281	20289	1.33	0.53	8.07
Christofides	rd400	400	15281	20249	1.33	10.25	7.93
Branch and Bound	fl417	417	11861	NA	NA	NA	109.04
Twice Around the Tree	fl417	417	11861	16226	1.37	0.54	8.67
Christofides	fl417	417	11861	17086	1.44	3.99	8.53
Branch and Bound	u2319	2319	234256	NA	NA	NA	37.35
Twice Around the Tree	u2319	2319	234256	320610	1.37	24.72	267.01
Christofides	u2319	2319	234256	317707	1.36	1010.35	267.01
Branch and Bound	pr2392	2392	378032	NA	NA	NA	33.63
Twice Around the Tree	pr2392	2392	378032	523832	1.39	26.17	285.36
Christofides	pr2392	2392	378032	514075	1.36	962.62	285.36
Branch and Bound	pcb3038	3038	137694	NA	NA	NA	0.74
Twice Around the Tree	pcb3038	3038	137694	196554	1.43	44.85	460.12
Christofides	pcb3038	3038	137694	188048	1.37	2845.15	460.12
Branch and Bound	fl3795	3795	28772	NA	NA	NA	0.96
Twice Around the Tree	fl3795	3795	28772	39110	1.36	108.45	719.38

4. Conclusão

Dessa forma, é possível concluir que o custo para se obter a solução ótima para esse problema é muito alto, mesmo com a técninca de Branch and Bound. Por isso, se faz necessário a exploração de algoritmos aproximativos como o de Christofides e o Twice Around the Tree para obter uma solução relativamente boa para fins práticos e em tempo polinomial. Também foi observado que para obter soluções cada vez melhores

é necessário trocar o tempo por qualidade de solução, como foi observado nos experimentos com vários vértices comparando os dois algoritmos aproximativos. Essa trabalho ajudou a entender melhor como lidar com problemas NP-Difíceis na prática e também a desenvolver maior familiaridade com Python.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. *Introduction to Algorithms*. MIT Press, 2009.

[2] Levitin, A. *Introduction to the Design and Analysis of Algorithms*. Pearson, 2011.