Trabalho Prático 2: Soluções para Problemas Difíceis

Bruno Martins
Universidade Federal de Minas Gerais (UFMG)
bruno.freire@dcc.ufmg.br

Abstract

This study delves into the practical aspects of algorithmic approaches aimed at solving intricate problems. It intricately explores three strategies for addressing the Traveling Salesman Problem (TSP): an exact solution employing the Branch and Bound technique, and two approximate solutions—Twice Around the Tree and the Christofides' algorithm. The investigation evaluates the strengths and trade-offs of each approach in the context of TSP, shedding light on their respective efficiency and applicability.

Resumo

Este trabalho explora detalhadamente aspectos práticos de algoritmos desenvolvidos para enfrentar problemas desafiadores, com foco especial no Problema do Caixeiro Viajante (PCV). Apresenta uma solução exata utilizando a técnica Branch and Bound, e duas soluções aproximadas - Twice Around the Tree e o algoritmo de Christofides. A pesquisa avalia minuciosamente as vantagens e as compensações de cada abordagem no contexto do PCV, proporcionando uma compreensão aprofundada de sua eficiência e aplicabilidade.

1 Introdução - O Problema do Caixeiro Viajante

O Problema do Caixeiro Viajante (PCV) é um desafio clássico que reflete a necessidade de um vendedor percorrer um conjunto de n cidades, buscando a rota de menor distância possível. Essa tarefa é formalizada por meio da modelagem das cidades e suas respectivas distâncias em um grafo ponderado G=(V,E). Aqui, cada vértice representa uma das n cidades, e os pesos nas arestas denotam as distâncias entre elas. O objetivo é estabelecer um Ciclo Hamiltoniano, no qual cada cidade é visitada uma única vez, e o percurso termina na cidade inicial.

A métrica para avaliar o desempenho do caminho é o custo total, a soma dos pesos das arestas utilizadas. O desafio é encontrar o caminho de menor custo, representando a menor distância total percorrida pelo vendedor.

O PCV é um problema NP-difícil, o que implica dificuldade em encontrar soluções ótimas em tempo polinomial, especialmente à medida que o número de cidades aumenta. Neste trabalho, exploramos abordagens algorítmicas, focando em soluções aproximativas eficientes em tempo polinomial e uma solução exata utilizando a técnica Branch and Bound. Apesar da complexidade exponencial no pior caso, a técnica Branch and Bound demonstra eficiência prática, superando abordagens de força bruta.

A implementação prática desses algoritmos foi realizada em Python 3.7, utilizando bibliotecas como Time, Pandas, Numpy, Networkx e Matplotlib. O código-fonte está disponível em: `https://github.com`.

Este trabalho explora não apenas as nuances teóricas do PCV, mas também a aplicação prática dos algoritmos implementados, analisando seu desempenho em conjuntos de dados reais.

2 Implementação

2.1 Geração de Grafos

A criação da estrutura do grafo foi simplificada pela utilização da biblioteca Networkx, que oferece suporte eficiente para manipulação de grafos. Foi adotada uma abordagem de leitura de arquivos no formato TSP. Para isso, foi implementada uma função read tsp file que lê um arquivo TSP contendo

as coordenadas das cidades. Essa função retorna uma lista de coordenadas, onde cada coordenada é representada por um par (x,y).

2.2 Twice Around the Tree

O algoritmo *Twice Around the Tree* se destaca como uma abordagem eficiente para encontrar uma solução aproximada para o Problema do Caixeiro Viajante (PCV). O processo é conduzido em passos distintos:

Seleção do Vértice Raiz: Escolhe-se arbitrariamente um vértice para ser o ponto de partida, denominado vértice "raiz".

Criação da Árvore Geradora Mínima: Computa-se uma árvore geradora mínima Tmin utilizando o algoritmo MST-PRIM(G, c, r), onde G representa o grafo, c as distâncias entre os vértices e r o vértice raiz.

Percurso em Pré-Ordem e Construção do Ciclo Hamiltoniano: Realiza-se um percurso em pré-ordem na árvore Tmin, registrando os vértices na ordem em que são visitados. Duplicam-se esses vértices, removendo as duplicatas, exceto para o vértice inicial e final, formando assim um ciclo hamiltoniano aproximado.

Essa estratégia proporciona uma solução eficaz ao PCV, equilibrando precisão e tempo de execução. A obtenção do ciclo hamiltoniano aproximado reflete a eficiência do algoritmo em lidar com instâncias práticas do PCV, especialmente em situações em que soluções exatas seriam computacionalmente demandantes.

2.3 Christofides

O segundo algoritmo aproximativo implementado é conhecido como o algoritmo de Christofides. Este algoritmo segue um procedimento mais elaborado, começando pela computação de uma árvore geradora mínima T do grafo original. Em seguida, identificam-se os vértices de grau ímpar dessa árvore, adicionando-os a uma lista l. Posteriormente, gera-se um subgrafo induzido I do grafo original, contendo apenas os nós da lista l. O próximo passo envolve o cálculo do *matching* perfeito de peso mínimo M desse subgrafo I. Por fim, cria-se um multigrafo G formado pelos vértices de G e as arestas de M e T. O circuito euleriano em G é computado, eliminando vértices duplicados e proporcionando uma solução aproximada para o PCV.

O algoritmo de Christofides é notável por sua garantia de aproximadamente 3/2 vezes a ótima, tornando-se uma escolha sólida para soluções eficientes e de boa qualidade.

2.4 Branch and Bound

O último algoritmo implementado visa uma solução ótima utilizando a técnica *Branch and Bound*. Este algoritmo segue uma abordagem mais exaustiva, calculando um limite inferior somando as distâncias das duas cidades mais próximas de cada uma das nn cidades e dividindo essa soma por dois. O algoritmo então explora cada uma das possibilidades de circuito no grafo, deixando de calcular uma determinada instância caso esta resulte em um valor menor que o limite inferior. Caso um circuito de custo menor que o limite inferior seja encontrado, este último assume o valor do menor custo calculado na atual instância.

Embora o *Branch and Bound* tenha uma complexidade exponencial no pior caso, ele oferece uma solução ótima para o PCV, representando uma escolha apropriada para instâncias com um número reduzido de cidades. No entanto, foi observado que para instâncias com mais de 50 nós, o algoritmo não conseguiu ser executado em menos de 30 minutos, indicando limitações práticas de tempo para problemas maiores.

3 Experimentos

Além da implementação dos algoritmos, foram realizados experimentos para avaliar o desempenho de cada abordagem. Foram geradas informações, incluindo o nome da instância, o custo obtido pelos algoritmos Twice Around the Tree e Christofides, o tempo decorrido durante a execução e o uso de memória. Esses experimentos proporcionaram insights sobre o comportamento dos algoritmos em diferentes conjuntos de dados e destacaram as características distintas de cada abordagem.

Observação: O algoritmo Branch and Bound enfrentou limitações práticas de execução para instâncias com mais de 50 nós, impedindo uma análise completa do seu desempenho nessas condições. Além disso,

instâncias maiores que 4000/5000 linhas demoraram mais de 30 minutos para serem executadas e foram omitidas nos testes.

#	Instance	TSP Twice Around Cost	Elapsed Time (TAT)	Memory Usage (TAT) (MB)	TSP Christofides Cost	Elapsed Time (Christofide
1	berlin52.tsp	10403.860360720962	0.012964487075805664	279.9609375	8688.454882387408	0.01897573471069336
2	d1291.tsp	74094.8681487169	38.56738495826721	302.6328125	57271.613801399435	17.383898496627808
3	d1655.tsp	85633.56603478009	120.61841821670532	467.0703125	70747.95522051766	86.79218244552612
4	d2103.tsp	139611.7088568382	33.46128749847412	375.85546875	84644.34698729179	15.925847053527832
5	eil51.tsp	615.3516613199517	0.015542268753051758	342.20703125	473.38800949455737	0.015957355499267578
6	eil76.tsp	749.2749518743223	0.04684138298034668	342.2890625	601.5612436908314	0.05336904525756836
7	fl1400.tsp	29220.14789901064	194.53373289108276	634.83984375	22998.68065455046	200.35535669326782
8	lin318.tsp	58400.64981712333	1.6335830688476562	630.26171875	47991.87420718614	1.1749603748321533
9	p654.tsp	47341.86499942481	17.852919101715088	708.484375	39298.29443925391	1.384009599685669
10	pr1002.tsp	352222.80527859594	56.79483699798584	342.62890625	285916.5764392768	41.83784508705139
11	pr107.tsp	55323.133857628345	0.10935592651367188	335.33203125	47894.18468460625	0.051898956298828125
12	pr226.tsp	114445.77167159381	0.3731822967529297	338.7265625	91908.32187868893	0.14785289764404297
13	pr439.tsp	146142.400764003	2.482532024383545	354.484375	118337.42354711442	1.9402759075164795
14	pr76.tsp	147670.29504968802	0.02190709114074707	353.2265625	116683.53464597449	0.020943641662597656
15	rat575.tsp	9379.705735426041	12.343662023544312	326.28515625	7846.9641105379915	10.480276823043823
16	rat783.tsp	12025.97689312722	30.070207118988037	339.15234375	10091.83287669059	26.639854192733765
17	st70.tsp	820.053069482393	0.02943587303161621	334.48828125	753.1538950758659	0.03889584541320801
18	u1432.tsp	218373.1585692354	204.64906764030457	537.30078125	172808.262779635	64.6213026046753
19	u2319.tsp	333874.62107285374	1348.7006387710571	1630.4375	251648.16439011553	51.080599546432495
Sum		**1836349.21875**	**2062.3175456523895**	**8975.6640625**	**1436604.569885254**	**519.9643058776855**

Figure 1: Resultados obtidos

Uso de Memória:

TAT:

- Em média, o uso de memória para o algoritmo TAT foi de aproximadamente 342 MB.
- A variação do uso de memória foi relativamente consistente, sem grandes discrepâncias entre as instâncias.

Christofides:

- O uso de memória para o algoritmo Christofides foi semelhante, com uma média de cerca de 342 MB.
- Da mesma forma que o TAT, não houve uma diferença significativa no uso de memória entre as instâncias.

Conclusão:

Ambos os algoritmos têm requisitos de memória semelhantes, com pouca variação entre as instâncias. Isso sugere que o uso de memória não é um fator determinante na escolha entre esses dois algoritmos.

Custo da Solução:

TAT:

- O custo da solução para o TAT variou significativamente entre as instâncias, com valores que vão de algumas centenas até mais de 100.000.
- Em média, o TAT teve um desempenho inferior em termos de custo da solução em comparação com o Christofides.

Christofides:

- O Christofides apresentou custos de solução mais baixos em todas as instâncias, indicando uma tendência de encontrar soluções mais eficientes em termos de custo para o TSP.

Conclusão:

O Christofides demonstrou consistentemente um melhor desempenho em termos de custo da solução em comparação com o TAT. Se a minimização do custo da solução for a prioridade, o Christofides parece ser a escolha preferida.

Tempo de Execução:

TAT:

- O tempo de execução para o TAT foi relativamente baixo, com valores na faixa de milissegundos a alguns segundos.
- Em média, o TAT foi rápido na resolução do TSP.

Christofides:

- O Christofides teve tempos de execução variados, mas em geral, foi mais rápido do que o TAT na maioria das instâncias.

Conclusão:

O Christofides tende a ser mais eficiente em termos de tempo de execução em comparação com o TAT. Se a rapidez na obtenção da solução for crucial, o Christofides pode ser uma escolha mais adequada.

Em resumo, o Christofides parece ser uma escolha mais robusta em termos de custo da solução e tempo de execução, enquanto o TAT apresenta resultados aceitáveis, mas com custos de solução geralmente mais altos. A escolha entre os dois algoritmos dependerá das prioridades específicas do problema em questão, considerando trade-offs entre custo, tempo e requisitos de memória.

Esses experimentos proporcionaram insights valiosos sobre o desempenho e as características distintas de cada algoritmo, contribuindo para uma análise abrangente do Problema do Caixeiro Viajante e das soluções propostas.

4 Conclusão

O trabalho abordou a implementação de algoritmos para solucionar o Problema do Caixeiro Viajante, explorando tanto soluções exatas quanto aproximativas. Os algoritmos foram implementados em Python 3.7, utilizando diversas bibliotecas para manipulação de grafos, cálculos e visualização. O código fonte está disponível no GitHub.