Lucas Jacone - 2019006922

Trabalho Prático 2 – Soluções para problemas difíceis

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Professor: Renato Vimieiro

Belo Horizonte, Minas Gerais
2023/2

1 Objetivo e Definição do Problema do Caixeiro Viajante Euclidiano

O objetivo central deste trabalho é abordar de forma prática os algoritmos para a solução de problemascomputacionais considerados desafiadores, com foco específico no Problema do Caixeiro Viajante euclidiano. São avaliadas e implementadas três abordagens distintas para a solução do PCV: uma exata, por meio do algoritmo branch-and-bound, e duas heurísticas, sendo elas o método twice-around-the-tree e o algoritmo de Christofides.

Dado um conjunto finito de pontos (ou "cidades") no plano euclidiano, o objetivo é encontrar a menor rota possível que visite cada ponto exatamente uma vez e retorne ao ponto de partida. A distância entre quaisquer dois pontos é a distância euclidiana, que é a distância "em linha reta"entre eles no plano.

1.1 Detalhes Matemáticos

Conjunto de Pontos

Seja P um conjunto de n pontos no plano euclidiano, onde cada ponto p_i representa uma localização ou cidade. Aqui, $P = \{p_1, p_2, \ldots, p_n\}$.

Distância Euclidiana

A distância entre dois pontos p_i e p_j é calculada pela fórmula euclidiana $d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$, onde (x_i, y_i) e (x_j, y_j) são as coordenadas cartesianas de p_i e p_j, respectivamente.

Caminho Hamiltoniano

Uma trajetória que passa por cada ponto exatamente uma vez.

Ciclo Hamiltoniano

Um caminho Hamiltoniano que retorna ao ponto inicial.

Objetivo

Encontrar o ciclo Hamiltoniano com o menor comprimento total, onde o comprimento é a soma das distâncias euclidianas entre pontos consecutivos no ciclo.

2 Algoritmos

2.1 Branch and Bound

O algoritmo `branchAndBoundTSP` representa uma adaptação sofisticada do método Branch and Bound para resolver o Problema do Caixeiro Viajante Euclidiano (PCVE). Este algoritmo é estruturado para explorar eficientemente o espaço de soluções, podando rotas que não conduzem à solução ótima e mantendo o foco nas rotas mais promissoras.

Utilizei o DFS devido à sua maior eficiência de memória, explorando uma rota de cada vez até o fim antes de retroceder, além disso o DFS pode encontrar soluções ótimas ou próximas mais rapidamente, crucial para podar eficientemente o espaço de busca assim que uma solução promissora é identificada. Isso permite descartar ramos improdutivos sem exploração completa, otimizando o tempo de processamento.

Estratégia do Algoritmo

O algoritmo inicia calculando um limite inferior e as arestas associadas para a rota inicial, criando um nó raiz que é inserido em um Min-Heap. Ele explora rotas usando essa estrutura de heap, avaliando cada nó pelo seu custo e limite inferior. Durante as iterações, o algoritmo extrai o nó com o menor limite, considerando sua solução parcial como a melhor atual se esta ultrapassar o número total de nós com um custo inferior ao melhor encontrado até então. Ele otimiza a busca com condições específicas, como evitar repetições de cidades e tratamentos especiais para as últimas cidades. Finalmente, retorna o custo da melhor rota e a sequência das cidades visitadas.

Características do Algoritmo

A função findInitialBound é usada para calcular o limite inicial, e a função findBound é usada para calcular o novo limite a cada expansão de nó. A estrutura do nó (Node) é central para manter o estado da solução parcial, incluindo o custo acumulado, as arestas selecionadas e a sequência de cidades visitadas.

Complexidade

O algoritmo branchAndBoundTSP para o Problema do Caixeiro Viajante tem uma complexidade assintótica de tempo e espaço que, no pior caso, pode aproximar-se de O(n!) devido à necessidade de explorar todas as permutações dos nós. A eficiência prática varia com a eficácia das heurísticas de poda aplicadas.

2.2 Christofides

Fornece uma solução cujo comprimento é, no máximo, 1,5 vezes maior que o da solução ótima. Particularmente eficaz para instâncias onde as distâncias entre as cidades satisfazem a desigualdade triangular.

Estratégia do Algoritmo

O algoritmo inicia construindo uma árvore geradora mínima (MST) do grafo, conectando todas as cidades com o menor custo sem formar ciclos. Em seguida, identifica nós de grau ímpar na árvore e cria um subgrafo, onde um emparelhamento mínimo de peso é realizado para conectar esses nós, minimizando o custo adicional. Um multigrafo é formado ao combinar a MST com o emparelhamento, garantindo que todos os nós tenham grau par, o que possibilita a existência de um circuito euleriano. O algoritmo utiliza o circuito euleriano para formar um ciclo hamiltoniano, removendo repetições de visitas às cidades. Finalmente, o peso total do caminho é calculado, resultando na solução do problema.

Características do Algoritmo

Este método é particularmente adequado para o problema, pois leva em consideração as propriedades geométricas das distâncias entre as cidades.

Complexidade

O algoritmo christofidesTSP tem uma complexidade assintótica primariamente dominada pela construção da árvore geradora mínima (MST) e pelo emparelhamento mínimo, ambos operando em $O(n^2)$ para grafos gerais, onde n é o número de nós. A conversão do circuito euleriano para um ciclo hamiltoniano é linear, O(n).

2.3 Twice Around The Tree

Este método é conhecido por sua simplicidade e eficiência, oferecendo uma solução aproximada que, embora não seja ótima, pode ser computada rapidamente.

Estratégia do Algoritmo

O algoritmo Twice Around the Tree começa construindo uma árvore geradora mínima do grafo, conectando todas as cidades para minimizar o custo total sem formar ciclos. Em seguida, realiza uma travessia em profundidade (DFS) a partir do nó raiz, gerando uma sequência de visitas aos nós da MST. O caminho do DFS é percorrido duas vezes, formando um ciclo que passa por cada nó pelo menos uma vez e ajustado para criar um

ciclo hamiltoniano. Por fim, o peso total do caminho é calculado, somando os pesos das arestas entre cidades consecutivas.

Características do Algoritmo

Este método se beneficia da eficiência da construção da árvore geradora mínima e da simplicidade da travessia DFS para formar um caminho que aproxima a solução. Embora a solução gerada não seja garantidamente ótima, em muitos casos, ela é suficientemente próxima do ótimo, especialmente considerando a economia de tempo computacional.

Complexidade

O algoritmo twiceAroundTheTreeTSP tem uma complexidade assintótica de tempo principalmente O(n), onde n é o número de nós, devido à travessia em profundidade (DFS) da árvore geradora mínima. O cálculo do peso do caminho é linear em relação ao número de arestas, que é O(n) para uma árvore.

3 Experimentos realizados

Os testes foram conduzidos utilizando instâncias selecionadas da biblioteca TSPLIB, com ênfase em instâncias que utilizam a distância euclidiana em 2D como função de custo. Para cada execução de um algoritmo em uma instância específica, medimos o tempo de execução e avaliamos a qualidade da solução gerada. Estabelecemos um limite de tempo de 1800 segundos (30 minutos) para cada instância. Instâncias que excederam este limite de tempo foram excluídas da análise, considerando os resultados como não-disponíveis (NA).

A análise considerou as seguintes métricas

- **Expected**: Métrica referente ao limiar do valor ótimo (solução) daquela instância, oferecido pelo professor por meio do Teams. Essa métrica é importante pois, com ela, será possível fazer o cálculo de checagem do fator de aproximação de cada algoritmo.
- Obtained: Solução obtida pelo algoritmo para cada instância.
- Aproximação Obtida: Métrica obtida ao realizar o cálculo de aproximação utilizando o valor ótimo de "Expected" e o valor obtido na execução do algoritmo "Obtained" (Expected/Obtained)
- Validade da Aproximação: É a métrica que avalia se o algoritmo conseguiu desempenhar em cada instância o fator de aproximação proposto. No caso dos algoritmos em questão, temos um fator de aproximação de 1.5 para o Christofides e 2 para o Twice Around The Tree.
- Tempo: Tempo de execução do algoritmo para cada instância (em segundos).

4 Execução dos Algoritmos

Essa seção discorrerá sobre os resultados obtidos após as execuções dos algoritmos nas 78 entradas da biblioteca.

Note que não estão dispostos na planilha todos os resultados. Isso se da ao fato da extensão dos experimentos. Os arquivos .csv contendo todos os resultados estarão dispostos jutamente com o restante da documentação no repositório de entrega.

Branch And Bound

Durante os experimentos realizados com o algoritmo branch and bound no contexto do Problema, observou-se que até mesmo na menor instância da biblioteca TSPLIB, o algoritmo ultrapassou o limite estabelecido de 30 minutos.

O resultado quando foi feita a execução na menor instância da biblioteca:

Runing: eil51
Expected: 426
TimeoutError: A execução excedeu o tempo limite de 1800 segundos.

Branch And Bound REDUCED

Foram realizados uma série de testes adicionais com o algoritmo branch and bound, foram criadas instâncias específicas com 12, 16 e 25 entradas para avaliar a escalabilidade e eficiência do algoritmo no Problema do Caixeiro Viajante.

Os resultados mostraram que o algoritmo foi capaz de completar as instâncias de 12 e 16 entradas dentro de um tempo razoável. No entanto, ao enfrentar a instância com 25 entradas, o algoritmo alcançou o limite de tempo estabelecido, sem produzir uma solução.

Runing: eil12
Expected: 169.1603
Branch and Bound: 169.1603
Aproximação obtida: 1.0
Validade da aproximação:True
Tempo: 0.03906

Runing: eil16
Expected: 213.2029
Branch and Bound: 213.2029
Aproximação obtida: 1.0
Validade da aproximação:True
Tempo: 0.7207

Runing: eil25
Expected: 233.7375
TimeoutError: A execução excedeu o tempo limite de 1800 segundos.

Esta observação reitera a natureza exponencial do branch and bound em problemas NP-difíceis, onde um aumento relativamente pequeno no tamanho da instância pode levar a um crescimento substancial no tempo de processamento, ultrapassando os limites práticos de execução.

Christofides

Nos testes realizados com o algoritmo de Christofides, foi observado um desempenho notável até mesmo em instâncias de grande escala. O algoritmo conseguiu processar dentro do limite do tempo, até a instância fnl4461, que possui 4461 entradas, mantendo-se dentro do fator de aproximação esperado, não excedendo o limite de até 1.5 vezes "pior"do que a solução ótima em todas as instâncias testadas.

Running	Expected	Obtained	Aprox Obtida	Validade	Tempo
eil51	426	"486.82"	"0.8750"	TRUE	"0.0156"
berlin52	7542	"8594.0"	"0.8775"	TRUE	"0.0156"
kroB100	22141	"24009."	"0.9221"	TRUE	"0.0625"
kroA200	29368	"33602."	"0.8739"	TRUE	"0.6355"
kroB200	29437	"33118."	"0.8888"	TRUE	"0.5026"
ts225	126643	"133282"	"0.9501"	TRUE	"0.0829"
tsp225	3919	"4376.0"	"0.8955"	TRUE	"0.4498"
linhp318	41345	"47259."	"0.8748"	TRUE	"1.3497"
rd400	15281	"17551."	"0.8706"	TRUE	"4.0302"
fl417	11861	"13424."	"0.8835"	TRUE	"1.5801"
u574	36905	"41337."	"0.8927"	TRUE	"8.2349"
u1060	224094	"249108"	"0.8995"	TRUE	"49.381"
u1432	152970	"171640"	"0.8912"	TRUE	"49.312"
u2152	64253	"73917."	"0.8692"	TRUE	"109.17"
fl3795	28772	"31490."	"0.9136"	TRUE	"750.63"
fnl4461	182566	N/A	N/A	N/A	N/A

Twice Around The Tree

O algoritmo foi capaz de processar com sucesso a instância rl11849, que contém 11849 entradas. Esse desempenho superior, em comparação com outros algoritmos testados, pode ser atribuído ao fator de aproximação de 2 do Twice Around the Tree. Embora menos preciso do que algoritmos com fatores de aproximação mais baixos (como 1.5 do Christofides), permite ao algoritmo ser mais flexível e eficiente em termos computacionais, lidando melhor com instâncias de grande escala.

Running	**Expected**	**Obtained**	**Aprox Obtida**	**Validade**	**Tempo**
eil51	426	"0.6646"	"0.6646"	TRUE	"0.0030"
kroB100	22141	"0.8554"	"0.7820"	TRUE	"0.0095"
lin105	14379	"0.7374"	"0.7374"	TRUE	"0.0105"
ch130	6110	"0.7516"	"0.7516"	TRUE	"0.0220"
d198	15780	"0.8210"	"0.8210"	TRUE	"0.0400"
pr264	49135	"0.7392"	"0.7392"	TRUE	"0.1044"
rl1323	270199	"0.7102"	"0.7102"	TRUE	"2.7853"
vm1748	336556	"0.7568"	"0.7568"	TRUE	"4.7918"
pr2392	378032	"0.7226"	"0.7308"	TRUE	"9.4336"
usa13509	19982889	N/A	N/A	N/A	N/A

5 Conclusão

Embora o Branch and Bound ofereça soluções exatas, demonstrou limitações significativas em termos de tempo de processamento. Em instâncias de maior tamanho, mostrou-se impraticável devido ao seu crescimento exponencial no tempo de execução. Esta característica o torna adequado apenas para instâncias menores, onde a precisão é indispensável e o tempo de execução é gerenciável.

Em contraste, o Twice Around the Tree mostrou-se capaz de processar instâncias muito maiores em tempo razoável. No entanto, este desempenho vem ao custo de um uso intensivo de memória RAM, devido à sua abordagem de manter múltiplas estruturas de dados em memória para agilizar a busca. Apesar de sua eficiência temporal, o algoritmo sacrifica a precisão, entregando soluções que podem ser até duas vezes piores que a ótima.

O algoritmo de Christofides, por sua vez, apresentou um meio-termo entre os outros dois. Ele foi capaz de processar instâncias de tamanho considerável, embora tenha demorado mais do que o Twice Around the Tree. No entanto, sua vantagem reside no uso mais eficiente da memória RAM, o que o torna uma opção viável para sistemas com recursos limitados. O Christofides oferece uma boa combinação de precisão e eficiência, com soluções que não excedem 1.5 vezes o valor da solução ótima.

Em resumo, podemos dizer que o Branch and Bound é ideal para precisão máxima em instâncias pequenas, o Twice Around the Tree para rapidez em instâncias grandes com recursos de memória suficientes, e o Christofides para um equilíbrio entre precisão e eficiência em sistemas com recursos mais limitados.

Referências

[1] Blossom Algorithm, *Tufts University*, https://www.eecs.tufts.edu/~gdicks02/Blossom/Blossom_Algorithm.html.

[2] NetworkX Documentation, *NetworkX*, https://networkx.github.io/documentation/latest/index.html.

[3] iGraph Library, *iGraph*, https://igraph.org.

[4] Dicas para Escrita Científica, *Universidade Federal de Minas Gerais*, https://homepages.dcc.ufmg.br/~mirella/doku.php?id=escrita.

[5] TSPLIB, *University of Heidelberg*, http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.

Para a implementação dos algoritmos foram utilizados os pseudocódigos e informações encontradas nos slides disponibilizados via teams. Também foi utilizado o livro do Levitin.