Trabalho 2 algoritmos 2

Davi Fraga Marques Neves

December 11, 2023

1 introdução

A ideia desse trabalho é desenvolver as técnicas de soluções para problemas matématicamente difícies(aqueles que ainda não se conhece uma solução eficiente para resolvel-lôs). O problema em questão é o problema da do caixeiro viajante onde se deve começar em um ponto de um grafo e passar por todos os pontos sem reptir nenhum e voltar no inicial. Para isso serão implementadas 3 tecnicas de soluções: branch-and-bound, twice-around-the-tree e o algoritmo de Christofides. A ideia é vizualizar cada solução e comparar a eficiencia de cada dadas as complexidades de tempo, espaço e qualidade da solução.

2 algoritmos

2.1 Twice around the tree

O algoritmo "Twice Around the Tree" é um método heurístico para resolver o Problema do Caixeiro Viajante que tenta aproximar uma solução ótima . A escolha da estimativa de custo é implícita na construção da Árvore Geradora Mínima (MST), que é feita usando o algoritmo de Prim. A soma dos pesos das arestas da MST serve como uma estimativa de custo, assumindo que o caminho ao longo da MST será um ponto de partida razoável.

Estrutura

- Grafo (Graph): Utiliza a biblioteca NetworkX para representar o grafo.
- Listas e Tuplas: As arestas da MST e do ciclo euleriano são representadas como listas e tuplas para manipulação eficiente.

Etapas

1. Construção da MST: Usa o algoritmo de Prim, que é um algoritmo best-first. Ele seleciona arestas com os menores pesos primeiro para construir a árvore geradora mínima.
2. Duplicação de Arestas: Todas as arestas da MST são duplicadas no grafo original.
3. Encontrar Ciclo Euleriano: Usa um algoritmo que, por padrão, produz um ciclo euleriano.

Analise de complexidade

1. Construção da Árvore Geradora Mínima (MST):
    - O algoritmo de Prim é usado para construir a MST.
    - A complexidade de tempo do algoritmo de Prim é geralmente $O(V\hat{2})$, onde V é o número de vértices.
    - No pior caso, onde o grafo é denso, a complexidade pode ser $O(V\hat{2} * \log(V))$ se a implementação usar uma fila de prioridade para escolher a próxima aresta.
2. Duplicação de Arestas: A duplicação das arestas da MST é uma operação que percorre todas as arestas da MST, portanto, tem uma complexidade de O(E), onde E é o número de arestas na MST.

3. Encontrar um Ciclo Euleriano: O algoritmo de Hierholzer para encontrar um ciclo Euleriano tem uma complexidade linear O(E), onde E é o número de arestas no grafo resultante.

4. Remoção de Duplicatas e Obtenção do Ciclo Hamiltoniano: Criar uma lista sem duplicatas (usando `list(dict.fromkeys(...))`) tem complexidade O(V + E), onde V é o número de vértices e E é o número de arestas no grafo resultante.

Portanto, a complexidade total do código é dominada pela construção da MST em muitos casos e é aproximadamente $O(V\hat{2} * \log(V))$, considerando um grafo denso. Note que a complexidade exata pode depender da implementação específica dos algoritmos em bibliotecas como NetworkX.

2.2 Branch and bound

algoritmo de Branch-and-Bound para o caixeiro viajante é uma abordagem sistemática que busca eficientemente uma solução exata para o problema. A escolha de uma estratégia de busca em profundidade é adequada para esse problema pois enquanto o caminho nao for descartado ele ainda será o meseficiente ate o momento sendo o maximo local. A implementação é projetada para ser eficiente e explorar apenas ramos promissores, evitando a exploração desnecessária de ramos não ferteis.

estrutua de dados

- Matriz de Custo): Uma matriz 2D para armazenar os custos acumulados de visitar cada nó a partir de qualquer outro nó.
- Lista de Visitados: Uma lista para rastrear quais nós foram visitados durante o processo de busca.
- Caminho Atual : Uma lista que armazena o caminho atual sendo explorado.
- Melhor Caminho: Uma lista que armazena o caminho ótimo encontrado até o momento.

Depth-First: A implementação usa uma abordagem de busca em profundidade. A função recursiva `branch_and_bound_helper` explora profundamente os ramos do espaço de busca, marcando nós visitados e desfazendo as marcações para realizar backtracking.

analise de complexidade

1. Inicialização e Preprocessamento: Inicialização da matriz de custo e da lista de visitados tem uma complexidade de O(n), onde n é o número de nós.

2. Função `branch_and_bound_helper`:
   - A função é chamada recursivamente para explorar todas as possíveis soluções.
   - Para cada nó, a função faz uma verificação e chama a si mesma recursivamente para os nós restantes.
   - O pior caso ocorre quando todos os ramos são explorados, resultando em uma complexidade exponencial.
   - A complexidade é geralmente O(n!), onde n é o número de nós.

3. Iteração Inicial para Iniciar o Branch-and-Bound:
   - Inicializa o algoritmo a partir de cada nó.
   - No total, inicia o algoritmo n vezes.
   - A complexidade para cada chamada é O(n!), então o custo total é O(n * n!).

4. Desenho do Grafo e do Ciclo Hamiltoniano: O desenho do grafo tem uma complexidade linear em relação ao número de nós (O(n)).

Em resumo, a complexidade do algoritmo Branch and Bound para o TSP implementado neste código é principalmente exponencial (O(n!)), o que o torna impraticável para instâncias grandes do problema. Apesar de ser um método exato, o Branch and Bound pode ser ineficiente para problemas do TSP em grafos completos, e algoritmos heurísticos e aproximados são frequentemente preferidos para instâncias mais extensas.

2.3 christofildes

A ideia desse algoritmo é achar uma solução aproximada para o problema do caixeiro viajante que seja mais eficaz em termos de tempo em comparação com os seus concorrentes. A ideia é que ele vai sacrificar precisão em função de tempo, o que pode ser muito útil para instâncias muito grandes.

Analise de complexidade

1. Construção do Grafo Completo Ponderado:
   - O código cria um grafo completo ponderado, considerando as distâncias euclidianas entre os pontos.
   - Se `n` é o número de pontos, a construção do grafo completo tem uma complexidade de $O(n\hat{2})$.
2. Construção da Árvore Geradora Mínima (MST):
   - O algoritmo de Kruskal ou Prim é usado para construir a MST.
   - A complexidade do algoritmo de Kruskal é $O(E * \log(V))$, onde E é o número de arestas e V é o número de vértices.
   - A complexidade do algoritmo de Prim é geralmente $O(V\hat{2})$, mas pode ser otimizada para $O(E + V * \log(V))$ se uma fila de prioridade é utilizada.
   - Portanto, a complexidade dominante é geralmente a construção da MST, que é $O(E + V * \log(V))$.
3. Encontrar Vértices de Grau Ímpar: Iterar sobre os vértices da MST tem uma complexidade de $O(V)$.
4. Resolver o Problema do Matching Perfeito Mínimo:
   - O algoritmo de emparelhamento perfeito é usado para encontrar um conjunto de arestas de peso máximo em um grafo bipartido.
   - A complexidade do algoritmo de emparelhamento perfeito é geralmente $O(V\hat{3})$.
   - A criação do grafo bipartido tem uma complexidade de $O(V\hat{2})$.
   - Portanto, a complexidade dominante é a do algoritmo de emparelhamento perfeito, que é $O(V\hat{3})$.
5. Combinação das Soluções e Encontrar o Ciclo Hamiltoniano:
   - A combinação das arestas da MST e do emparelhamento perfeito tem uma complexidade de $O(E + V)$ devido à criação da lista combinada.
   - Encontrar o ciclo euleriano e o ciclo hamiltoniano tem complexidade linear $O(E)$.

Em resumo, a complexidade total do algoritmo de Christofides geralmente é dominada pela construção da MST, resultando em uma complexidade aproximada de $O(E + V * \log(V))$. Note que a complexidade exata pode variar dependendo da implementação específica das bibliotecas utilizadas.

2.4 comparação

o algoritmo branch-and-boud traz uma solução otima porém o custo de tempo é muito grande. O twice-around é no maximo duas vezes pior e o christofildes é no maximo 1,5 vezes pior, além disso ele vai ficando melhor conforme a instância aumenta.