Trabalho Prático 2: TSP, Qual as melhores abordagens?

Tomas Lacerda Muniz
Universidade Federal de Minas Gerais (UFMG)
tomas.muniz@dcc.ufmg.br

Introdução

Fomos desafiados a comparar três algorítimos para a resolução do famoso problema do caixeiro viajante. Estas são compostas por o algorítimo de branch and bound, uma solução mais precisa porém custosa e de outros dois algorítimos dotados de métodos heurísticos para conseguir soluções aproximadas mas não exatas, porém muito mais rápidas.

1 Introdução - O Problema do Caixeiro Viajante

O Problema do Caixeiro Viajante consiste no desafio de um vendedor que precisa visitar um conjunto de cidades, buscando a rota mais curta possível. Este problema é modelado através de um grafo ponderado G = (v, e), onde cada vértice representa uma cidade e as arestas, com seus respectivos pesos, indicam as distâncias entre elas. O objetivo do vendedor é encontrar um Ciclo Hamiltoniano, que permita visitar cada cidade uma única vez, retornando ao ponto de partida.

O custo total do percurso é calculado pela soma dos pesos das arestas utilizadas na rota. O vendedor almeja, portanto, encontrar a rota de menor custo. O Problema do Caixeiro Viajante é classificado como NP-difícil em termos de otimização. Este trabalho explora algoritmos aproximativos que fornecem soluções próximas em tempo polinomial.

2 Implementação

2.1 Disclaimer

Não consegui implementar própriamente os algorítimos devido a falta de tempo, confesso que administrei muito mal e por isso não consegui fazer uma ampla comparação usando os diversos algorítimos propostos, por isso percorrerei aqui como implementei estes algorítimos que estão executando em grafos aleatórios e não nos datasets propostos pelo trabalho.

2.2 Twice Around the Tree

O código implementa o algoritmo "Twice Around the Tree" para aproximar um ciclo Hamiltoniano em um grafo. Este método compreende várias etapas chave:

1. **Construção da Árvore Geradora Mínima (MST):** Utiliza-se a função `nx.minimum_spanning_tree` da biblioteca NetworkX para encontrar a MST do grafo de entrada. Utilizei o método de Prim.

2. **Duplicação de Arestas e Criação de Multigrafo:** Todas as arestas da MST são duplicadas, formando um conjunto de arestas que é utilizado para construir um multigrafo. Isso assegura a existência de um ciclo Euleriano.

3. **Encontrando um Ciclo Euleriano:** Através da função `nx.eulerian_circuit`, encontra-se um ciclo Euleriano no multigrafo e percorro cada aresta uma só vez.

4. **Conversão para Ciclo Hamiltoniano Aproximado:** O ciclo Euleriano é convertido em um ciclo Hamiltoniano, que visita cada vértice uma vez, retornando ao ponto inicial. A conversão é feita verificando os vértices já visitados e evitando repetições.

2.3 Christofides

O código apresentado implementa o Algoritmo de Christofides, uma abordagem heurística para resolver o Problema do Caixeiro Viajante (TSP) em grafos ponderados. O algoritmo é dividido nas seguintes etapas principais:

1. **Árvore Geradora Mínima (MST):** Utiliza-se a função `nx.minimum_spanning_tree` da biblioteca NetworkX para construir a MST do grafo, baseando-se no peso das arestas. Utilzei também o método de Prim.

2. **Emparelhamento Mínimo Perfeito:** Identificam-se os vértices de grau ímpar na MST. Em seguida, realiza-se o emparelhamento mínimo perfeito no subgrafo induzido por esses vértices. A função `find_minimum_weight_perfect_mat` calcula esse emparelhamento usando o algoritmo de aproximação, visto que o NetworkX não possui uma função integrada para isso.

3. **Construção de um Multigrafo:** As arestas do emparelhamento são adicionadas à MST, resultando em um multigrafo.

4. **Ciclo Euleriano e Conversão para Ciclo Hamiltoniano:** Utilizando a função `nx.eulerian_circuit`, encontra-se um ciclo Euleriano no multigrafo. Esse ciclo é então convertido em um ciclo Hamiltoniano, visitando cada vértice uma vez e evitando repetições.

2.4 Branch and Bound

Este segmento de código implementa o algoritmo Branch and Bound para resolver o Problema do Caixeiro Viajante (TSP) em um grafo ponderado completo. O algoritmo é estruturado nas seguintes etapas principais:

1. **Preprocessamento dos Edges:** A função `precomputeEdges` organiza as arestas de cada nó do grafo em ordem crescente de peso e seleciona as duas menores para futuros cálculos de limites.

2. **Cálculo do Limite Inferior (Bound):** A função `bound` calcula o limite inferior para um dado nó do Branch and Bound, considerando os pesos das arestas e os nós já incluídos na solução.

3. **Cálculo do Limite Base:** A função `calculateBaseBound` estabelece um limite base inicial para a busca, somando os dois menores pesos de arestas de cada nó.

4. **Branch and Bound:** A função `branchAndBound` executa o algoritmo propriamente dito. Ela usa uma fila de prioridades para explorar os nós com menores limites primeiro, expandindo-os e atualizando o melhor custo encontrado quando uma solução completa é alcançada.

A performance do algoritmo é medida em termos de tempo de execução. O código calcula e imprime o custo da melhor solução encontrada e o tempo total de execução, fornecendo uma avaliação eficaz da eficiência do algoritmo Branch and Bound para o TSP.

3 Resultados

Tive como conclusão que os algorítimos aproximativos geram resultados muito mais rapidamente, utilizando de métodos heurísticos, porém não apresentam a precisão que o Branch and Bound apresenta... entretanto ao subirmos os números de nós, o Branch and Bound gasta muito tempo para rodar, sendo incapaz de rodar com mais do que 30 nós na forma que eu programei. Ainda que existam muitos espaços para otimizações.

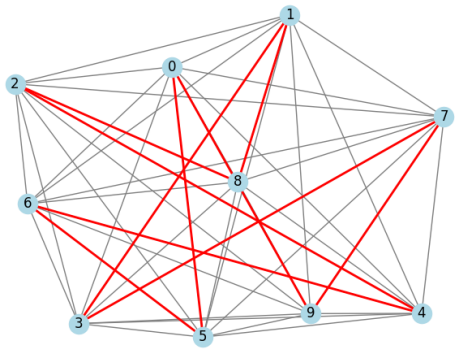

Figure 1: Grafo de Christofides.

Estes grafos são exemplos aleatórios gerados por grafos aleatórios com 10 nós a fim de exemplificar visualmente a execução do algorítimo.

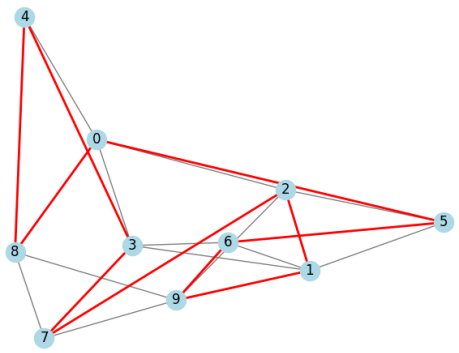

Figure 2: Grafo do Twice Around the Tree.