3 algoritmos para o problema do caixeiro viajante

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil
Leonardo de Oliveira Maia: 2019042139
lom2019@ufmg.br

1 Introdução

O problema do caixeiro viajante consiste em um problema enfrentado por varias áreas que envolve o melhor trajeto que poderia ser dado para visitar todos os pontos de um dado local e retornar ao ponto inicial sem ocorrer a repetição de arestas visitadas, ou seja, um problema de otimização de rotas utilizado, por exemplo, por uma empresa que deseja abastecer de água certas cidades em sua região com 1 caminhão e retornando ao ponto de partida sendo o mais rápido possível. Para isto foi feito 3 algoritmos que resolvessem esse problema baseando em diferentes formas de estruturar, estes algoritmos são eles: Twice Around the Tree, Christofides e Branch and Bound.

2 Ideia Central

Estamos querendo analisar os 3 algoritmos para o problema do caixeiro viajante, comparando questões de desempenho que seriam o uso de memoria, o quão próximo da solução ótima os algoritmos se aproximam e o tempo de execução de cada um deles. Abordaremos também em quais tipos de situações cada algoritmo seria melhor utilizado para resolver o problema caso tivéssemos que escolher entre os 3.

3 Estrutura de dados

Por meio do uso da biblioteca "tsplib95" e os datasets já fornecidos do site "comopt", foi feito a costrução dos grafos para cada dataset, seus vértices representam as cidades e as arestas que ligam entre elas suas rotas. No algoritmo Branch and Bound foi feito o uso da classe "Vértice" para auxiliar na construção de nos em direção a solução ótima

4 Algoritmos

4.1 Twice Around the Tree

Este algoritmo foi feito utilizando a biblioteca "networkx" que tem a capacidade de manipular grafos para gerar arvore mínima que sera definida pela função qual sera a melhor, ou seja, varia a complexidade entre $O(ElogV)$ ou $O(V^2)$, por ser uma arvore mínima utilizamos Depth-First Search (DFS) para percorrer gradualmente com pre-ordem também da biblioteca "networkx" e gerar nosso caminho mínimo, a pre-ordem inicial nos da O(V), enquanto a DFS nos da $O(V + E)$. Somamos os caminhos gerados pelo algoritmo e obtemos nosso peso solução. Este algoritmo nos da complexidade dominada pela construção da arvore mínima, variando entre $O(ElogV)$ ou $O(V^2)$.

4.2 Christofides

Este algoritmo foi feito utilizando a biblioteca "networkx" que tem capacidade de manipular grafos para gerar emparelhamento mínimo nos vértices de grau ímpar na árvore mínima do grafo, o emparelhamento em grafos com baixo grau é eficiente, tendendo a ser $O(n^3)$. Foi feito o uso de arvore geradora mínima, fazemos o pareamento utilizando a arvore, criamos um multigrafo combinamos os 2 citados anteriormente tendo complexidade $O(n)$ no multigrafo e realizamos o menor caminho de complexidade $O(V+E)$. Assim, o algoritmo fica dominado pelo emparelhamento mínimo nos vértices e tem complexidade $O(n^3)$.

4.3 Branch and Bound

O algoritmo começa calculando um limite inicial para a árvore de busca, o que envolve percorrer cada nó do grafo e encontrar os dois menores pesos das arestas conectadas a cada nó. Isso tem uma complexidade aproximada de $O(V)$, onde V é o número de vértices. Durante a busca, o algoritmo utiliza uma fila de prioridade simulando uma pilha para explorar os vértices em ordem de limites inferiores, reduzindo a complexidade de busca para O(log N) para adicionar e remover elementos da pilha. A busca continua até que todos os vértices sejam explorados, limitando a altura da árvore de busca a V e a complexidade das operações em cada nível da árvore a $O(V)$ no pior caso. Portanto, a complexidade final do algoritmo depende fortemente da eficácia dos limites superiores calculados e, em geral, no pior caso, a complexidade de tempo é exponencial e pode ser expressa como $O(2^N * N^2)$, onde N é o número de vértices no grafo. Essa complexidade é devido à natureza exponencial da busca exaustiva por todas as permutações possíveis de vértices, combinada com a complexidade quadrática para calcular o limite inicial do grafo. Foi escolhido para o algoritmo a implementação "Best-First" que prioriza os caminhos mais promissores e podando ramos desnecessários para serem calculados, assim economizamos a memoria do computador.

5 Experimentos

Os experimentos foram feitos utilizando métricas de tempo de execução, espaço utilizado e quão próximo da solução ótima cada algoritmo aproximou. Foi feito o uso de "thread" para limitar o tempo de execução dos algoritmos para 30min no máximo, sendo descartado caso não tenha concluído a tempo. Durante a realização dos experimentos, foi notado que o algoritmo de "Branch and Bound" apresentava o melhor resultado em comparação aos outros, porem, devido a sua complexidade, a medida que os vértices do grafo cresciam, este algoritmo começava a ser o mais ineficiente em questão de tempo e espaço, com a criação de vários nos e múltiplas pesquisas, seu tempo crescia exponencialmente e fazia o maior uso de memoria. Para calculo mais rápido o "Twice around the three" possuía maior velocidade, porem devido ao grafo possuir circuitos eulerianos, o algoritmo apresentava maior imprecisão em relação a solução ótima para os grafos, foi o algoritmo que menor utilizou o espaço de memoria. O algoritmo mais consistente foi o de "Christofides", apresentava uma solução com diferença máxima de 1,5 em comparação a solução ótima do problema de cada grafo, consumia memoria um pouco acima do "Twice around the three", sua velocidade ficou no meio entre os 3 algoritmos e a medida que crescia as cidades se tornava um algoritmo cada vez melhor para números de vértices altos.

6 Conclusão Final

Concluímos que para o problema do caixeiro viajante seja mais interessante utilizar o algoritmo de "Branch and Bound" nos casos em que a quantidade de cidades seja expressivamente pequena. Para o caso de quantidade de cidades extremamente altos talvez seja interessante utilizar o "Twice around the three" que mesmo apresentando um desvio da solução ótima maior, sua resposta sera mais rápida em comparação aos outros que demandariam mais tempo. Nos casos médios e se devêssemos escolher apenas um algoritmo para utilizar sempre, escolheríamos o "Christofides" que seria o meio entre uso de memoria, tempo e aproximação da solução ótima.