Documentação Trabalho Prático 2

Gabriel Castelo Branco Rocha Alencar Pinto, Universidade Federal de MInas Gerais

11 de Dezembro de 2022

1 Introdução

Neste trabalho prático, foi abordado o famoso "Problema do Caixeiro Viajante"(do inglês, Travelling Salesman Problem, ou TSP). O problema, em linhas gerais, consiste no seguinte: Um vendedor, saindo de uma determinada cidade, deseja passar por todas as cidades em um local, passando apenas uma vez em cada cidade. O objetivo do problema é buscar o menor caminho possível que cumpra tais condições.
O TSP é, notávelmente, um problema *NP-Difícil*, ou seja, um algoritmo que resolva o problema terá fator de dificuldade no mínimo exponencial. O foco deste trabalho, portanto, será utilizar duas diferentes abordagens para calcular (ou estimar) o caminho mínimo. São elas:

- Algoritmo Branch-And-Bound, como solução exata para o TSP, mas com custo exponencial
- Algoritmos aproximativos (mais especificamente, os algoritmos de Christofides e Twice-Around-The-Tree).

Além disso, foi parte do trabalho a criação de um gerador de instâncias do problema. O gerador é responsável por construir instâncias de tamanho 2^i, $\forall i\ 4 \leq i \leq 10$. Uma instância é composta por i pares ordenados (x, y) estritamente diferentes, que representam pontos em um plano cartesiano.

2 Detalhes de Implementação

Para trabalhar com a maior eficiência possível, o trabalho foi implementado em C++11, com o compilador GNU g++. Para representação das estruturas lógicas (grafos e matrizes de adjacência) a biblioteca iGraph (versão 0.10.2) foi utilizada. Além disso, devido à ausência de algoritmos para cálculo do Matching de peso mínimo em grafos na biblioteca iGraph, utilizado no algoritmo de Christofides, foi utilizado um script Python, no padrão Python 3.9+. Dentro deste, foram utilizadas as Bibliotecas Networkx e Numpy.
Para Facilitar a interpretação do problema em termos de códigos, foram criadas três classes, para fazer as 3 principais funcionalidades exigidads pela situação:

- **instance-generator**: Classse responsável por gerar as instâncias, garantindo que cada ponto seja único no espaço, além de salvar em um arquivo de texto o resultado.
- **instance-processor**: Classe Responsável por processar um arquivo de texto com uma instância, gerando um vetor com os pontos da mesma.
- **solver**: Classe responsávelpor resolver a instâcia, gerando a matriz de adjacência, calculando as distâncias de Euclides e Manhattan de todos os pontos para todos os pontos, e, por fim, Aplicando os algoritmos para cada uma das 3 instâncias.

Estas classes consituem o corpo do programa. Agora, tratando especificamente dos algoritmos utilizados:

- **Branch And Bound**: A implementação utilizada para o algoritmo Branch-And-Bound foi uma transposição direta do pseudocódigo apresentado em sala de aula para código C++. Para esta implementação, não foram utilizadas as estruturas para tratar de grafos baseadas na biblioteca *iGraph*, mas sim estruturas padrão implementadas na *Standart Template Library*, a STL de C++.
  - O custo do Algoritmo Branch And Bound é, no pior caso, exponencial em essência, pois visitaria todas as soluções possíveis até encontrar a máxima. Todavia, é uma garantia de segurança para a resposta, em casos onde não se pode tolerar a margem de erro dos algoritmos aproximativos.
  - As principais estruturas de dados utilizadas foram: uma priority queue, para armazenar os pontos a serem visitados, e a estrutura **Node**, que basicamente é um container que possui 4 elementos: *Bound, cost, level e s.* O valor S é um vetor contendo id dos demais nós com os quais o elemento faz fronteira.
- **Twice Around The Tree**: A implementação do algoritmo Twice around the Tree foi feita seguindo as orientações das notas de aula, todavia, para representar as estruturas de dados e para computar algoritmos secundários, como uma *Árvore Geradora Mínima*, ou MST, foi utilizada a biblioteca *iGraph*. Contendo uma grande diversidade de algoritmos, a biblioteca permite representar rapidamente um grafo, dado sua matriz de adjacência, calcular árvore geradora mínima e as demais necessidades do algoritmo.
  - O algoritmo Twice Around the Tree é 2-aproximativo para o problema do caixeiro viajante, ou seja, enquanto que ele não provêm necessariamente a melhor solução, chegando a propor uma solução até duas vezes pior para o problema a depender da instâcia, seu custo de execução é polinomial.

- **Algoritmo de Christofides**: O algoritmo de Christofides, ao contrário dos Demais, não foi implementado em C++, devido às limitações da biblioteca *iGraph*. foi definido, portanto, um script auxiliar em Python para a implementação do mesmo. As estruturas utilizadas para execução do programa são aquelas providas pela biblioteca *Networkx*.

  – O algoritmo de Christofides é um algoritmo 1.5 aproximativo, ou seja, um algoritmo que, no pior caso, dará uma solução 1.5x pior do que aquela considerada a ótima. Nota-se, portanto, que é um algoritmo estritamente melhor do que o Algoritmo Twice Around the Tree, ou seja, apresenta resultados sempre menores. Tal assertiva é confirmada mais adiante na sessão 3 deste estudo.

3 Experimentos

Aqui, serão demonstrados os resultados dos testes com as instâncias geradas conforme explicado na sessão 1. Nota-se que há diferença considerável no tempo de execução entre o algoritmo de Christofides e o Twice Around The Tree, todavia tal atraso se deve à implementação deste ser em Python[1], enquanto que os outros foram escritos em C++, compilados com otimização máxima permitida pelo compilador (-O3)[2].
O algoritmo Branch And Bound, devido ao custo de execução, acabou por não produzir resultados em tempo hábil para esta demonstração.[3]

[1]Por ser uma Linguagem Interpretada, o custo de tempo de tokenização é relevante no que tange a algoritmos custosos.

[2]As configurações da máquina utilizadas para o teste são: Processador Ryzen 5 5600G, Memória: 16GB DDR4 3000MHz, GPU AMD RX 6700XT.

[3]Respeitando o limite de tempo de 30 minutos.

Resultados do Experimento			
Algortimo	Instância	Tempo de Execução	Caminho Mínimo
Christofides	2^4	0.244575 s	261.689980
Christofides	2^5	0.136870 s	357.067330
Christofides	2^6	0.140664 s	549.668480
Christofides	2^7	0.196704 s	769.458530
Christofides	2^8	0.333948 s	1032.053680
Christofides	2^9	1.010247 s	1498.529060
Christofides	2^10	4.053145 s	2135.295680
Twice Around the Tree	2^4	0.000044 s	932.646238
Twice Around the Tree	2^5	0.000091 s	1579.482045
Twice Around the Tree	2^6	0.000308 s	4162.018635
Twice Around the Tree	2^7	0.001305 s	7806.920033
Twice Around the Tree	2^8	0.006007 s	17041.619342
Twice Around the Tree	2^9	0.028102 s	30895.474565
Twice Around the Tree	2^10	0.154774 s	66224.750684
Branch And Bound	2^4	N/A	N/A
Branch And Bound	2^5	N/A	N/A
Branch And Bound	2^6	N/A	N/A
Branch And Bound	2^7	N/A	N/A
Branch And Bound	2^8	N/A	N/A
Branch And Bound	2^9	N/A	N/A
Branch And Bound	2^10	N/A	N/A

4 Conclusões Finais

Após a execução dos algoritmos e subsequente análise de resultados, diversos fatos ficam evidentes. A priori, é possível perceber a diferença de eficiência de cada algoritmo aproximativo com relação aos valores de caminho mínimo encontrados. É evidente que o algoritmo de Christofides é bem mais eficiente no que tange à precisão do resultado, encontrando inexoravelmente soluções menores do que o Twice Around the Tree. Todavia, mesmo considerando as diferenças de plataformas que separam Python e C++, fica evidente que o custo do algoritmo de Christofides, que envolve não somente o cálculo de uma MST, mas também um Matching de Peso mínimo, pesa no quesito temporal.
Destas evidências, fica demonstrado como o custo teórico se aplica ao em situações práticas, ou seja, em termos de tempo de execução e acuidade de resultados.

5 Instruções de Compilação

O programa foi desenhado para funcionar especificamente em máquinas Linux, executando comandos que não funcionariam em outros sistemas operacionais. Os testes foram realizados em um Ubuntu 20.04. Faz-se necessária, também, a

existência da biblioteca iGraph instalada, compilada estaticamente. Por fim, o interpretador Python3 e a Biblioteca Networkx devem ambos estarem instalados.
Para Compilar o programa, basta acessar o diretório da pasta já descompactada e esecutar o comando *make*.
As instruções para execução do programa podem ser obtidas através de *./main -h*.