Soluções para Problemas Difíceis

Traveling Salesperson Problem

Vinícius Alves de Faria Resende

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brasil

Resumo: O trabalho se configura como uma avaliação sistemática acerca das estratégias para lidar com problemas difíceis, em especial, o problema *Traveling Salesperson Problem*. Com a utilização da linguagem de programação C++ foram desenvolvidos algoritmos exatos e aproximativos que foram submetidos a testes com as instâncias de problemas TSP euclidianos provindos da TSPLIB. Uma avaliação sistemática dos resultados foi desenvolvida para conferir os comportamentos esperados confrontados com os observados. Por fim é feita uma dissertação sobre todo o corpo do que foi desenvolvido trazendo *insights* valiosos sobre o conteúdo do trabalho.

1. Introdução

O trabalho realizado é uma exploração sobre as diferentes abordagens disponíveis para lidar com o Problema do Caixeiro Viajante (TSP). Este é um dos problemas difíceis mais clássicos da ciência da computação. Em suma, a ideia do problema é que dada uma lista de cidades e distâncias entre elas, o objetivo é encontrar o menor caminho possível que visite cada cidade exatamente uma vez e retorne à cidade original.

Em termos matemáticos, podemos dizer que se temos um conjunto de n cidades, queremos encontrar uma permutação que minimize a distância percorrida. Disto, podemos derivar a seguinte formulação matemática do problema: Considere que d_{ij} representa a distância entre duas cidades arbitrárias i e j, e considere também que teremos x_{ij} como uma variável binária, que dirá se o Viajante realmente usou o caminho direto de i para j. A função objetiva do problema é minimizar a distância total:

$$Min(\sum_{i=1}^{n} \sum_{j=1,\, j\neq i}^{n} d_{ij} \cdot x_{ij})$$

Essa minimização é sujeita à algumas restrições:

1. Toda cidade deve ser visitada, e somente uma vez:

$$\sum_{j=1,\, j\neq i}^{n} x_{ij} = 1,\ \forall i$$

2. O Viajante deve sair de cada cidade exatamente uma vez:

$$\sum_{i=1,\, i\neq j}^{n} x_{ij} = 1,\ \forall j$$

3. Além disso, precisamos de colocar restrições em sub-caminhos que são geradores de ciclos:

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \geq 2,\ \forall S \subset \{2, 3, \ldots, n\},\ |S| \geq 2$$

O problema TSP se classifica como NP-Difícil, o que significa que não há algoritmo polinomial conhecido que possa resolver este problema para qualquer instância de forma ótima. Portanto, existem heurísticas e aproximações que são utilizadas para encontrar uma solução próxima da ótima para grandes instâncias do problema. Esse tipo de solução mostra-se ainda mais aderente ao caso uma vez que, para problemas reais, muitas vezes há problema na computação/coleta de dados, por isso, mesmo uma solução algorítmica ótima pode não representar com perfeição a realidade.

No contexto do trabalho desenvolvido, a ideia é que sejam abordados os aspectos práticos ao tratar com um problema TSP. Para tal, será apresentada uma solução exata que tira proveito da técnica *Branch-and-Bound*, que busca otimizar o algoritmo exaustivo, mesmo que não diminua sua complexidade assintótica. Além disso, para problemas de TSP que se enquadram nas características de um problema TSP euclidiano, existem algoritmos aproximativos com fatores de aproximação constante que também serão explorados. Os algoritmos aproximativos a serem explorados no trabalho são o *Twice-Around-the-Tree*(2-aproximado) e o Algoritmo de *Christofides*(1.5-aproximado).

De forma a realizar uma exploração empírica dos algoritmos mencionados defronte à diversas bases de dados do problema, foi escolhida uma implementação utilizando a linguagem de programação C++, a escolha da linguagem se deu pela familiaridade do programador com a linguagem, além de que uma linguagem compilada sem coletor de lixo como C++ tende a ter um bom desempenho, de forma que, para obter a mesma eficiência em *Python*, é necessário fazer muito uso de bibliotecas como *numpy* e *pandas* o que afasta a implementação do código dos pseudo-códigos estudados, segmentando o aprendizado à uma linguagem, o que não considerei desejável.

Considerando o problema TSP, uma representação de grafos se faz necessária, para tal escolhi uma representação por matriz de adjacência, uma vez que não haveria ganho significativo em uma representação por listas de adjacência, dado que para problemas TSP todas as arestas possíveis existem. Além disso esta implementação permite acesso direto ao peso de uma arestas e_{ij}, o que é desejável considerando a natureza do problema.

De forma geral, os algoritmos mencionados foram implementados para operar sobre uma implementação de matriz de adjacência para o grafo equivalente ao problema TSP em questão, maiores detalhes sobre cada algoritmo serão descritos em seções posteriores, bem como suas estimativas de custo de tempo e espaço.

2. Detalhes de Implementação

No que tange aos detalhes de implementação, estes estarão descritos nesta seção, de forma que cada algoritmo supracitado será individualmente analisado. Isso porque cada algoritmo possui características únicas, variando sua implementação, qualidade de resposta e complexidade assintótica de tempo e espaço.

2.1. Estruturas de Dados

No que se refere às estruturas de dados, não há grande complexidade de estruturas utilizadas, sendo a maioria delas específica para cada algoritmo e serão descritas no contexto do algoritmo.

Estruturas simples auxiliares foram utilizadas em diversas partes do código, como vetores, tuplas, *strings, stacks* e *streams*, porém todos já são disponibilizados pela biblioteca STL, e não precisaram de personalização.

Contudo, há uma estrutura que norteia todas as implementações, que é o Grafo. A implementação do Grafo se deu por uma simples utilização de matriz de adjacência, uma vez que esta implementação apresenta um custo constante para consulta de arestas, o que é muito proveitoso para o problema TSP. Existe um problema com a implementação que é o fato de que ela ocupa o espaço das arestas que talvez não existam no Grafo original, porém as entradas do problema TSP possuem arestas entre todos os vértices, mais uma vez motivando a escolha de uma representação por matriz de adjacência. A matriz é $n\ x\ n$, e cada posição $(i,\ j)$ da matriz contém o custo da aresta que liga i a j.

2.2. Algoritmos Aproximativos:

Twice Around the Tree

O algoritmo *Twice Around the Tree* é consideravelmente popular ao lidar com problemas do tipo TSP, e apresenta um fator de aproximação igual a 2. A algoritmo se baseia em uma heurística construtiva onde uma Árvore Geradora Mínima (MST) é computada, então caminha-se nessa árvore considerando arestas bidirecionais até que forme-se um ciclo fechado. Repetições de vértices (voltas) podem ser substituídas por um caminho direto, onde $u \rightarrow v \rightarrow w$ pode ser substituído por $u \rightarrow w$ caso o vértice v já tenha sido visitado. Isto só é possível dado um problema TSP euclidiano (desigualdade triangular).

Como o algoritmo é aproximativo, ele se caracteriza por ser polinomial, por isso, seus passos são bem definidos e podem ser descritos de forma imperativa. O primeiro passo do algoritmo consiste em computar a MST a partir do grafo da instância TSP, para computar a árvore foi usado o Algoritmo de Prim, este algoritmo é bem conhecido e estudado na academia, nesse contexto é conhecido que sua complexidade assintótica de tempo é igual a $O(|V|^2)$, onde $|V|$ é o total de vértices (cidades). Para a computação da árvore, foi utilizada uma implementação iterativa com o auxílio de uma estrutura de dados de *Minimum Heap*, que receberá uma tupla para cada aresta, contendo sua origem, destino e peso, o *Heap* irá ordenar as arestas recebidas por aquelas de menor peso. Desta forma, adicionam-se arestas ao *Heap* e marcam-se os vértices como visitados, adicionando ao Heap cada uma das arestas ainda não computadas. Ao final, têm-se um novo grafo como árvore, seguindo também uma representação de matriz de adjacência, porém apenas as arestas da MST terão valores válidos.

Uma vez que a MST foi computada, é necessário seguir para o próximo passo do algoritmo aproximativo, que se caracteriza por fazer um caminhamento do tipo pré-ordem na árvore geradora mínima. Esta operação pode ser feita iterativamente com o auxílio de uma estrutura de dados *Stack*, de forma que os vizinhos não visitados de um vértice da árvore serão adicionados ao caminhamento antes mesmo do vértice em questão. O caminhamento é polinomial e, por ser uma árvore, passa por todos os vértices e arestas, dando um custo de $O(|V| + |E|)$ onde $|E|$ é o total de arestas da árvore.

Após computar o caminhamento, basta colocar ao final do caminhamento o primeiro vértice novamente, fechando assim o ciclo necessário para o *tour* do TSP. Tendo o *tour* e o Grafo da instância TSP, é possível iterar pelos vértices na ordem que aparecem e calcular o peso das arestas que os ligam, computando assim o peso total do caminho. Isso pode ser feito em tempo $O(|V|)$.

Por fim, o algoritmo apresenta uma solução com fator de aproximação constante igual a 2, conforme a prova demonstrada durante as aulas expositivas. Além disso, vimos que sua complexidade assintótica de tempo é dominada pelo Algoritmo de Prim, que apresenta complexidade assintótica de

tempo igual a $O(|V|^2)$. Do ponto de vista de complexidade assintótica de espaço, vemos que o grafo apresenta complexidade de espaço da ordem de $O(|V|^2)$, além disso são usadas várias estruturas auxiliares para armazenar o *tour*, pilhas e *heaps*, porém todos com complexidade de espaço igual a $O(|V|)$, portanto a complexidade de espaço é dominada pelo grafo e é igual a $O(|V|^2)$.

Christofides

O Algoritmo de *Christofides* foi derivado de uma observação feita por Nicos Christofides sob a ótica apresentada pelo algoritmo *Twice Around the Tree*, e ficou popular uma vez que pode lidar com problemas do tipo TSP euclidiano e apresenta um fator de aproximação constante igual a 1,5. A abordagem do algoritmo também se baseia na computação da MST, porém para a construção das arestas necessárias para fechar-se o ciclo do *tour* para o problema, utiliza-se uma técnica de *Perfect Matching*, e, com isso, é possível computar um Circuito Euleriano usando os vértices ímpares da árvore MST. Além disso, este algoritmo também apresenta a otimização para repetições de vértices, uma vez que também trata com instâncias do problema TSP que estão submetidas à desigualdade triangular.

Dada a natureza aproximativa do algoritmo, ele se caracteriza por executar em tempo polinomial, por isso, seus passos são bem definidos e podem ser descritos de forma imperativa. O primeiro passo do algoritmo consiste em computar a MST a partir do grafo da instância TSP, para o algoritmo de Christofides, também utilizou-se o Algoritmo de Prim, que já foi descrito em detalhes anteriormente.

Após a computação da árvore geradora mínima, faz-se necessário computar os vértices da árvore que possuem grau ímpar, isso se faz necessário pois o próximo passo do algoritmo irá calcular o *Perfect Matching* considerando apenas estes vértices. Como a árvore possui todos os vértices da cidade, computar os vértices de grau ímpar pode ser feito de forma trivial, passando por cada linha da matriz que representa a MST e contabilizando a quantidade de vizinhos (colunas com peso $\neq 0$), caso esta quantidade de vizinho seja ímpar o vértice representado pela linha será adicionado ao vetor de vértices ímpares. Este passo apresenta um custo assintótico de tempo igual a $\theta(|V|^2)$.

Dada a construção da Árvore Geradora Mínima e da computação dos vértices da árvore que possuem grau ímpar, precisamos agora computar um *Minimum Perfect Matching* considerando os vértices de grau ímpar. Esta etapa pode ser feito utilizando o algoritmo conhecido como *Edmonds' Blossom Algorithm*, e apresenta uma complexidade assintótica de tempo igual a $O(|V|^3)$, nesse caso $|V|$ será a quantidade de vértices de grau ímpar. Este algoritmo é implementado de forma iterativa tirando proveito de um vetor que armazena os vértices que já foram emparelhados.

Tendo computado o emparelhamento perfeito do subgrafo induzido e também a MST, é possível formar um novo multigrafo unindo as arestas da árvore e também do *matching*, formando então um novo grafo (devido a propósitos de otimização a implementação do código edita o grafo da mst incluindo as arestas já durante a execução do algoritmo de *matching*). Portanto, este novo grafo pode ser submetido ao algoritmo do Circuito Euleriano, que tira proveito de uma estrutura de dados de pilha para conseguir implementar uma abordagem iterativa, este algoritmo é polinomial e conhecido, apresenta uma complexidade assintótica de tempo igual a $O(|E|^2)$, como a MST e o perfect matching deixam apenas as arestas necessária para o circuito, podemos dizer que, para este caso, a complexidade pode ser dada como $O(|V|^2)$.

Por fim, dado que o Circuito Euleriano foi computado, faz-se um pós processamento sobre o resultado para remover duplicatas, aplicando o conceito de desigualdade triangular sempre que uma

repetição de vértice for encontrada. Essa computação pode ser feita em $O(|V|^2)$, fechando assim o ciclo necessário para o *tour* do TSP. Tendo o *tour* e o Grafo da instância TSP, é possível iterar pelos vértices na ordem que aparecem e calcular o peso das arestas que os ligam, computando assim o peso total do caminho. Isso pode ser feito em tempo $O(|V|)$.

Por fim, o algoritmo apresenta uma solução com fator de aproximação constante igual a 1,5, conforme a prova demonstrada durante as aulas expositivas. Além disso, vimos que sua complexidade assintótica de tempo é dominada pelo *Edmonds' Blossom Algorithm*, que apresenta complexidade assintótica de tempo igual a $O(|V|^3)$. Do ponto de vista de complexidade assintótica de espaço, vemos que o grafo apresenta complexidade de espaço da ordem de $O(|V|^2)$, além disso são usadas várias estruturas auxiliares para armazenar o *tour* e pilhas, porém todos com complexidade de espaço igual a $O(|V|)$, portanto a complexidade de espaço é dominada pelo grafo e é igual a $O(|V|^2)$.

2.3. Algoritmos Exatos

Branch-and-Bound

A estratégia *Branch and Bound* é uma abordagem muito popular para resolver problemas de característica combinatória de forma exata, por exemplo, o *Traveling Salesperson Problem* (TSP). A ideia gira em torno de realizar uma exploração do espaço possível de soluções, usando técnicas para priorizar ramos mais promissores e descartar soluções subótimas ou impossíveis. A ideia por trás da abordagem gira em torno do cálculo de uma estimativa da solução ótima, que têm de ser calculada sem grandes esforços, pois esse cálculo se repetirá com frequência. Desta forma, começa-se em um nó raiz (solução vazia) e a cada passo o algoritmo expande a solução parcial, priorizando aquelas que apresentam uma melhor estimativa de solução ótima.

Nesse contexto, é necessário definir uma estratégia de caminhamento na árvore do espaço de soluções, para o problema em questão foi escolhido um caminhamento por lista de prioridades, este caminhamento é conhecido como *Best First Search*. Ele funciona de forma a sempre priorizar os ramos mais promissores, para o problema TSP, o ramo com menor custo total estimado para o *tour*. Esta abordagem pode fazer com que seja mais difícil chegar a soluções parciais subótimas, diferentemente de uma abordagem DFS que rapidamente alcança uma folha da árvore do espaço de soluções, porém, a idéia é de que ao chegar a um nó folha, há uma boa probabilidade de que aquela seja a melhor solução possível, descartando assim vários outros ramos sem a necessidade de expandi-los em um processo exponencial.

Dessa forma, o algoritmo consiste na construção de nós que representam uma solução parcial do problema, nestes nós armazena-se o nível (profundidade), o custo do caminho acumulado, o caminho acumulado e o *bound* que é a estimativa para o custo ótimo. De início computa-se um nó raiz, partindo do vértice 0 do grafo, e adiciona-o à fila de prioridades, após isso inicia-se um processo iterativo que executa enquanto existirem nós na fila de prioridade.

Para cada nó, verifica-se se o seu nível (profundidade) é maior que o número de vértices, caso positivo significa que a solução parcial acumulada já inclui todos os vértices e forma um ciclo, dessa forma se a solução encontrada nesta folha for melhor que a melhor solução até o momento, atualiza-se a melhor solução até o momento para a solução da folha. Agora, se a profundidade do nó for menor que n, verifica-se se a estimativa de solução ótima do nó é melhor que a melhor solução até o momento, caso contrário pode considerar o nó como infrutífero. Caso positivo, então a solução parcial do nó ainda pode levar a uma solução melhor do que a computada até o momento, nesse caso, existem ainda dois casos possíveis, um onde o nível (profundidade) é menor que o valor n, ou seja, ainda há vértices para serem

visitados, nesse caso, para cada um dos n vértices, verifica-se se ele já não está no caminho parcial do nó, se realmente existe uma aresta entre ele e o último nó adicionado ao caminho, caso positivo então calcula-se o *bound*, ou seja, a estimativa da solução dado que este vértice k seja adotado como o próximo, então, se esta estimativa é menor que a melhor solução até o momento, então adiciona-se este novo nó à fila de prioridades. Existe ainda um outro caso, que é quando a profundidade é igual ao número de vértices, neste cenário todos os vértices já fazem parte da solução, e é apenas necessário que o ciclo seja fechado, neste momento é feita uma verificação para assegurar-se que todos os vértices do grafo foram visitados, e adiciona-se o vértice inicial ao fim do caminho acumulado, atualizando sua estimativa (*bound*).

O algoritmo descrito acima segue de forma bem aproximada o pseudo-código fornecido nos slides da disciplina. Porém nos slides não é descrito a estratégia para o cálculo do *bound*, no trabalho foi escolhida uma abordagem onde já considera-se o bound dada a adição de um próximo vértice, por isso, usa-se uma abordagem onde a função que retorna o novo bound, recebe o antigo nó e o novo a ser adicionado (com o *bound* inicialmente zerado). A partir daí, verifica-se o caso se que todos os vértices já estão na solução, nesse caso sabemos que não há aresta a ser escolhida a não ser a que liga o último vértice ao primeiro. Agora, caso o caminho do nó ainda não tenha todos os vértices, então é feita uma busca por uma aresta para um vértice que ainda não está na solução parcial e que tem o menor custo, então o custo desta aresta mínima é adicionada ao *bound* do nó anterior, determinando assim o *bound* do novo nó.

Na perspectiva abstrata do problema, o algoritmo opera de forma iterativa sobre os nós da árvore do espaço de soluções, utilizando uma abordagem heurística que remove nós que não se mostram frutíferos e priorizando aqueles que se mostram como mais promissores. Porém, considerando um cenário de pior caso, onde a poda de ramos não seja eficiente, teremos então um processo de exploração na árvore como um todo. Nesse caso, estaremos de frente com um problema permutativo como foi descrito na **Introdução**, e este processo cai em uma complexidade assintótica de tempo da ordem de $O(|V|!)$, onde $|V|$ é o número de vértices do grafo.

Sob a ótica da análise de complexidade de espaço, precisamos levar em consideração o custo de $O(|V|^2)$ para o armazenamento do grafo, porém, além disso é necessário armazenar os nós da árvore. A suposição mais intuitiva é a de que seria necessário armazenar $O(|V|!)$ nós, sendo um para cada nó da árvore. Porém esta intuição se mostra errônea, uma vez que os mesmos conceitos de complexidade de tempo não se aplicam para a complexidade de espaço, dada a possibilidade do reaproveitamento de espaço. A ideia é de que para cada iteração do *loop*, não só os nós atuais estão armazenados, mas também os nós de níveis (profundidades) anteriores, bem como as possíveis ramificações a partir de cada um dos nós, ou seja, em um pior caso, em um nó no nível n teriam outros $n - 1$ nós anteriormente armazenados, além de que, para cada nó, poderiam ter até n possíveis ramificações, levando a um custo assintótico de espaço da ordem de $O(|V|^2)$. Agora, considerando que este nó de nível n seja atingido, ele será liberado da memória após a iteração envolvendo-o, e o algoritmo passará para computar outro nó que poderá aproveitar este espaço.

3. Experimentação e Resultados

De frente com a natureza prática do trabalho, faz-se necessário um processo de experimentação e avaliação de resultados. Por isso, foi desenvolvido um trabalho sistemático na submissão de *datasets* para

que fosse analisados pelos algoritmos descritos. Nesse processo puderam ser computadas métricas que guiaram esta análise.

Para a experimentação são necessários dois recursos, sendo eles os dados que possam ser mapeados para uma instância do problema TSP para que os caminhos sejam computados e também os recursos computacionais para a execução dos algoritmos. No que tange aos dados, foram usados os *datasets* apontados pelo professor, que são de domínio público e puderam ser obtidos pelo site da Universidade *Heidelberg*. No que se refere aos recursos computacionais, foi usado um computador pessoal que dispõe de um processador *Ryzen 7 5700x* (4.6GHz Turbo), 32GB de memória RAM 3200 MHz configurado com o Subsistema Linux para Windows (WSL 2). Além disso, foi utilizado o compilador C++ conhecido como G++, usado na versão 11.4.0.

Os *datasets* apontados pelo professor são provenientes da biblioteca TSPLIB, e caracterizam problemas do tipo euclidiano, desta forma é possível utilizar os algoritmos aproximativos descritos, uma vez que só há garantia de fator de aproximação constante no caso de que as entradas respeitem as restrições do TSP euclidiano.

Nesse contexto, foi desenvolvido um script em *bash* capaz de executar a aplicação e fornecer como parâmetro o dataset desejado para a análise, passando todos os *datasets* listados pelo professor individualmente e armazenando as saídas em um arquivo de texto. Especialmente para o algoritmo de *Branch-and-Bound* existe um limitador implementado no código, que interrompe a execução do algoritmo se a execução passar de trinta minutos. Infelizmente isto impossibilitou que o algoritmo executasse com as entradas fornecidas, já que começam em 52 cidades. Contudo, consegui verificar a assertividade do algoritmo com entradas com menos de 20 cidades, mesmo que levando um tempo considerável, a solução ótima era encontrada. Outras abordagens para o *Branch-and-Bound* foram testadas para que, ao interromper a execução, a melhor solução parcial fosse retornada, mas os resultados sempre eram bem piores que os resultados aproximativos.

Apesar da limitação de tempo do algoritmo *Branch-and-Bound*, os algoritmos aproximativos executaram em tempo razoável para todas as entradas, até mesmo as que eram próximas de 20 mil cidades. Já no que se refere ao espaço, o consumo de memória RAM não foi alarmante para os algoritmos, sendo que o WSL não chegou a consumir mais do que 8 GB de memória. Este comportamento era esperado dada a análise de complexidade assintótica de espaço que foi supracitada.

Neste momento, serão apresentados alguns gráficos a fim de ilustrar de forma geral os resultados obtidos, cada gráfico será discutido em detalhes para que possa ser feita uma avaliação com o propósito de traçar paralelos entre os resultados esperados e os obtidos.

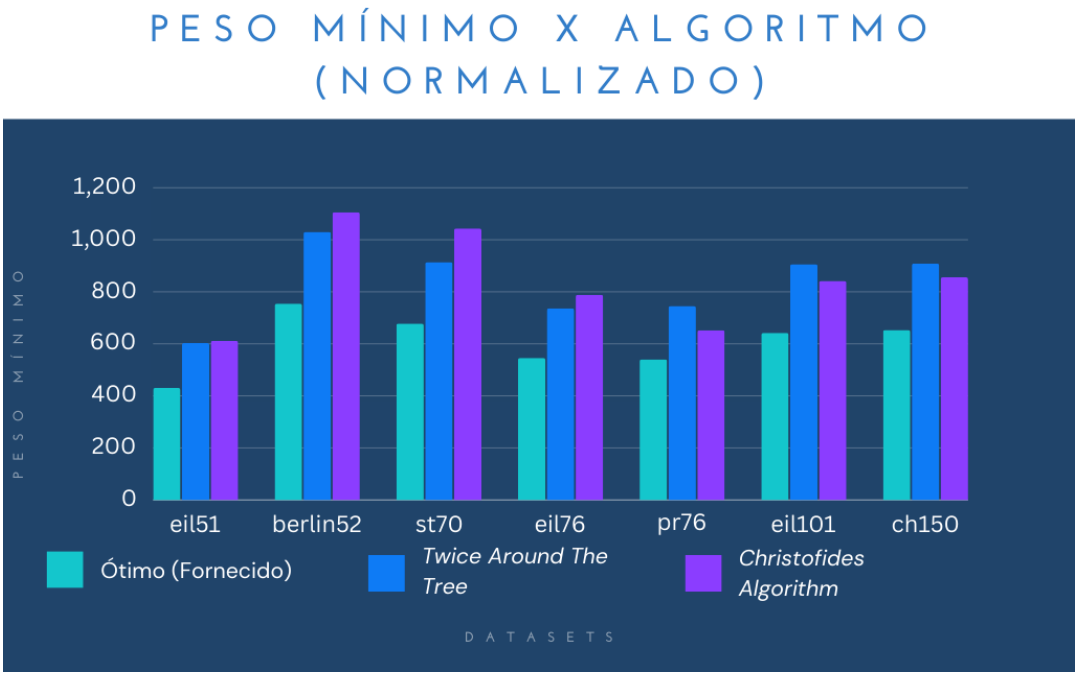

Figura 1 - Peso mínimo encontrado por dataset por algoritmo

A primeira figura representa graficamente o resultado da execução dos três algoritmos sob alguns dos *datasets* considerados. Basicamente é possível visualizar o caminho ótimo para cada uma das instância e também o resultado dos dois algoritmos aproximativos, infelizmente não há a possibilidade de fazer a visualização em grandes escalas no quesito número de cidades, isso porque a computação da solução ótima para estes cenários é complexa e não pode ser computada em tempo viável. De toda forma, a lição principal que é tirada da análise do gráfico se refere às proporções, essas podem ser visualizadas com clareza uma vez que os dados foram normalizados para que todos ficassem em um mesmo intervalo a fim de facilitar a interpretação dos resultados.

Como é esperado, o resultado ótimo sempre é menor que o resultado dos algoritmos aproximativos, e temos que, ao menos para as instâncias apresentadas, não há diferença significativa entre os dois algoritmos aproximativos para a grande maioria dos casos. Um comportamento que chamou-me a atenção é o fato de que nem sempre os algoritmos aproximativos apresentam um comportamento homogêneo, por vezes o algoritmo *Twice Around The Tree* desempenha melhor que o *Algoritmo de Christofides*, mesmo sendo que o fator de aproximação do segundo é menor que do primeiro. De toda forma, apesar de inesperado, este resultado é completamente justificável, até porque, o fator de aproximação de um algoritmo aproximativo diz respeito apenas ao pior caso de um algoritmo e não há casos isolados ou até mesmo ao caso médio. Então, no que tange a verificação do fator de aproximação dos algoritmos, o que realmente é importante é o fato de que, comparados à solução ótima, os algoritmos respeitam a restrição do algoritmo aproximativo, mesmo graficamente é possível observar que nenhuma das barrar chega a ser 50% maior que a solução ótima, tampouco 100% maior que a ótima. Desta forma verifica-se que os fatores aproximativos dos algoritmos estão mantidos, sendo que do algoritmo *Twice Around The Tree* temos a garantia de uma solução no máximo duas vezes o valor da ótima e no caso do algoritmo de *Christofides* temos a garantia de uma solução no máximo uma vez e meia o valor da ótima.

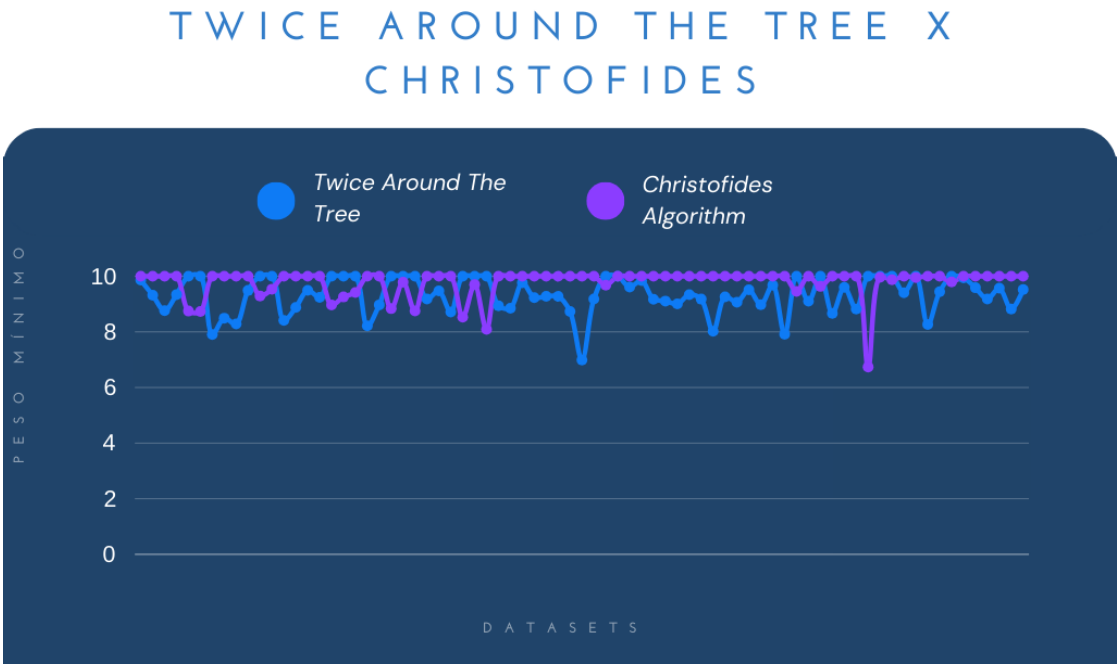

Figura 2 - Resultado normalizado do algoritmos *Twice Around the Tree* e *Christofides*

A segunda figura representa uma comparação entre os dois algoritmos aproximativos implementados segundo a óptica do resultado obtido. Os valores foram normalizados entre os datasets para que fosse possível fazer uma visualização não enviesada. No gráfico apresentado, o pior resultado entre os dois algoritmos sempre terá o valor 10, e então o segundo resultado é representado mantendo as proporções originais.

Como a visualização apresenta todos os *datasets* considerados, é possível ver o comportamento dos algoritmos ao longo do tempo. No geral, vemos que o comportamento apresentado na Figura 1 é mantido de certa forma, onde existe uma forte percepção de que o algoritmo *Twice Around the Tree* tende a obter resultados melhores. Mesmo havendo exceções é possível notar que os pontos azuis dominam a parte inferior do gráfico, o que representa as instâncias onde o algoritmo *Christofides* apresentou um resultado pior.

De forma geral, mesmo com o fato supracitado, é possível notar que a diferença entre os valores não é muito significativa para a maioria dos casos, se mantendo na casa dos 10-20% de desvio. Como mencionado anteriormente, apesar do fato de que o algoritmo de *Christofides* tem o fator de aproximação menor que o *Twice Around the Tree*, isso não diz respeito à execuções arbitrárias, apenas a um limite superior comparado ao caso ótimo. Portanto, mesmo que o resultado desafie a expectativa intuitiva, o resultado obtido é justificável e não quebra nenhuma restrição teórica dos algoritmos.

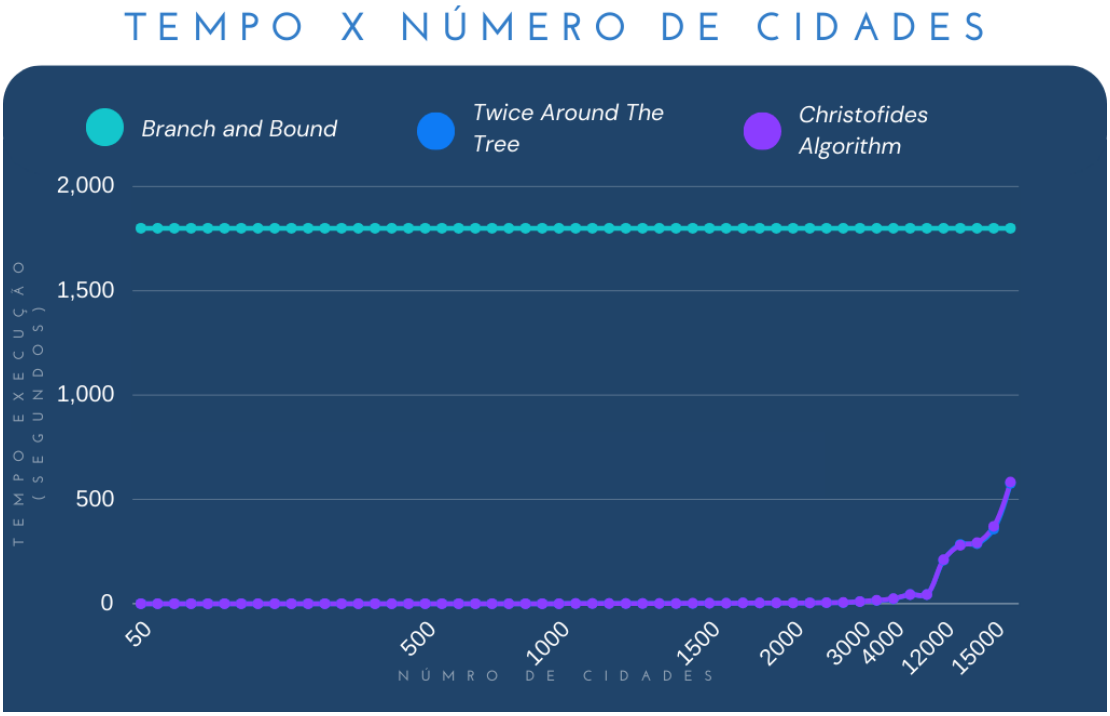

Figura 3 - Tempo de execução por número de cidades por algoritmo

Na figura 3, passamos agora para uma perspectiva que busca comparar os algoritmos no que se refere ao tempo de execução, que está diretamente atrelada à complexidade assintótica de tempo. É importante notar que a escala do tempo de execução do gráfico é dada em segundos. O primeiro comportamento que chama a atenção é a linha reta constante da execução do algoritmo *Branch and Bound,* este comportamento não é desejável uma vez que mostra que o algoritmo não conseguiu terminar antes do limite pré-determinado de 30 minutos, esta expectativa existia se considerar instâncias com mais de 100 cidades por exemplo, mas havia-se uma expectativa de que para instâncias menores que essa o algoritmo iria executar com sucesso, como o menor dos datasets apresenta 50 cidades, não houve resultado menor que o tempo limite.

Agora considerando os algoritmos aproximativos vemos que eles possuem um comportamento bem similar, com um começo consideravelmente *flat*, onde a maioria das instâncias consideradas executam em algo muito próximo de 0 segundos, esse comportamento perdura por bastante tempo no gráfico dado que a maioria das instâncias está localizada no intervalo de 50 a 2000 cidades. A partir deste ponto já é possível ver um singelo comportamento de crescimento do tempo de execução, que é acentuado a partir de 12 mil cidades. É possível ver que o tempo de execução de ambos os algoritmos aproximativos é similar, não havendo grande discrepâncias até dificultando a visualização das retas de forma separada. Apenas nota-se um comportamento singelo onde a curva do algoritmo *Twice around the Tree* parece ficar um pouco abaixo da curva do algoritmo de *Christofides*.

Este comportamento é esperado em uma certa maneira, onde havia a expectativa de que o algoritmo *Branch and Bound* teria um tempo de execução bem superior aos algoritmos aproximativos dada a sua complexidade assintótica exponencial. Agora, do ponto de vista dos algoritmos aproximativos, havia-se uma expectativa de que a diferença no tempo de execução dos algoritmos fosse um pouco mais acentuada, mas mesmo para uma entrada de 18 mil cidades (último ponto), não há grande acentuação da diferença dos valores de tempo de execução. Outra observação interessante é de que, mesmo com a maior

das instâncias, os algoritmos aproximativos não levaram mais do que 10 minutos para executarem, o que mostra que os algoritmos polinomiais não são afetados de forma dramática pelo aumento no tamanho da entrada.

Com as informações apresentadas, podemos já visualizar que a abordagem aproximativa para a construção de algoritmos para lidar com problemas difíceis se mostra promissora tanto no quesito de proximidade com o resultado ótimo quanto na grande melhoria no tempo de execução. Na próxima seção teremos uma discussão mais aprofundada sobre as conclusões finais tiradas do projeto como um todo.

4. Conclusão

Após a elicitação da seção de experimentos e resultados, é possível, agora, partir para uma conclusão geral do projeto, onde os pontos mais relevantes podem ser elucidados. De forma geral, foi possível observar como os algoritmos aproximativos se comportam em relação às instâncias indicadas no enunciado do trabalho.

Os resultados comprovam a eficácia das respostas obtidas pelos algoritmos aproximativos, onde vimos que para casos onde a solução ótima foi disponibilizada, as respostas aproximativas se mantinham no intervalo de 1.5 vezes o valor ótimo, além disso, apresentaram um tempo de execução abruptamente menor se comparado com o algoritmo *Branch and Bound*, reforçando a ideia de que os algoritmos aproximativos sacrificam parte da precisão em troca de um ganho considerável de eficiência.

De forma geral, pudemos observar que o algoritmo *Branch and Bound* ficou muito para trás dos aproximativos no que se refere ao tempo de execução. Porém, uma comparação que se mostrou muito proveitosa foi a do algoritmo de *Christofides* com o *Twice Around the Tree*, mesmo ambos se caracterizando como algoritmos aproximativos, eles apresentam notável diferença em termos teóricos, como seus respectivos fatores de aproximação e complexidade assintótica. Contudo, se mostraram consideravelmente similares na prática observada.

Vale a pena mencionar que o algoritmo *Twice Around the Tree* mostrou-se surpreendentemente eficaz comparado ao *Christofides*, de forma que para grande parte das instâncias apresentou um resultado melhor, mesmo possuindo um fator de aproximação maior, o que induz pensar que os resultados observados também seriam maiores. Porém, como foi mencionado, o fator de aproximação é um limite superior, e não determina como o algoritmo se comporta para casos arbitrários. Nesse contexto, considero que tive uma surpresa positiva em termos da eficácia do algoritmo *Twice Around the Tree*, que teve um desempenho em média 10-20% superior ao *Christofides*.

Agora, outro ponto notável é o fato de que, em termos de tempo, o algoritmo de *Christofides* se mostrou extremamente parelho ao *Twice Around the Tree* sendo difícil até diferenciar ambos em uma visualização gráfica. Foi apresentado que a complexidade assintótica de tempo do algoritmo de *Christofides* é maior, por isso, observar por meio de experimentações um resultado tão parelho é algo a se considerar como surpreendente. Mais uma vez, estamos de frente com um caso onde um limite (complexidade assintótica de tempo), não diz respeito diretamente ao padrão observado em instâncias arbitrárias, pois não necessariamente teremos um cenário de pior caso frequente que será capaz de gerar uma discrepância nos tempos de execução. Por isso, considero que tive uma surpresa positiva no que se refere à eficiência do algoritmo *Christofides*, que teve um tempo de execução observado muito parelho ao *Twice Around the Tree*.

De maneira geral, o trabalho feito demonstra claramente as diferenças de eficácia e eficiência entre algoritmos exatos e aproximativos para problemas difíceis, em especial, o problema TSP. A Figura 1 é capaz de demonstrar as perdas observadas no que se refere a precisão da resposta e a Figura 3 consegue

demonstrar os ganhos no que se refere ao tempo de execução. Então, conclui-se que um algoritmo *Branch and Bound*, mesmo para instâncias consideradas “pequenas” não se mostrou viável, tendo um tempo de execução abruptamente maior. Por outro lado, os algoritmos aproximativos mostraram um desempenho acima do esperado, mostrando-se muito promissores para lidar com problemas do tipo TSP euclidiano, tendo o algoritmo de *Christofides* destaque positivo no que se refere ao tempo de execução menor que o esperado (garantindo um fator de aproximação de 1.5) e um destaque para o *Twice Around the Tree* que, para os experimentos realizados, se mostrou melhor que o algoritmo de *Christofides* no que se refere à eficácia.

Em suma, o estudo desenvolvido mostrou-se muito eficiente para que uma decisão racional baseada em dados e argumentações teóricas possa ser tomada ao lidar com um problema mapeável ao Problema do Caixeiro Viajante Euclidiano. O trabalho mostrou os desafios ao lidar com soluções ótimas para problemas difíceis, além de enaltecer a potencial eficiência ao se usar algoritmos aproximativos para lidar com problemas reais. O trabalho desenvolveu uma exploração densa sobre o tema proposto, trazendo à tona os desafios de lidar com problemas difíceis em cenários reais, além de trazer um confronto entre as possibilidades, enaltecendo as diferenças entre expectativas e valores observados com experimentação.

REFERÊNCIAS

BRAIN KART. **Approximation Algorithms for the Traveling Salesman Problem**. Disponível em: https://www.brainkart.com/article/Approximation-Algorithms-for-the-Traveling-Salesman-Problem_8065. Acesso em: 8 dez. 2023.

CPLUSPLUS. **Standard C++ Library reference**. Disponível em: https://cplusplus.com/reference/. Acesso em: 8 dez. 2023.

GEEKS FOR GEEKS. **Best First Search (Informed Search)**. Disponível em: https://www.geeksforgeeks.org/best-first-search-informed-search/. Acesso em: 8 dez. 2023.

GEEKS FOR GEEKS. **Branch and Bound Algorithm**. Disponível em: https://www.geeksforgeeks.org/branch-and-bound-algorithm/. Acesso em: 8 dez. 2023.

GEEKS FOR GEEKS. **Prim's Algorithm for Minimum Spanning Tree (MST)**. Disponível em: https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/. Acesso em: 8 dez. 2023.

GEEKS FOR GEEKS. **The C++ Standard Template Library (STL)**. Disponível em: https://www.geeksforgeeks.org/the-c-standard-template-library-stl/. Acesso em: 8 dez. 2023.

PROGRAMIZ. **Graph Data Stucture**. Disponível em: https://www.programiz.com/dsa/graph. Acesso em: 8 dez. 2023.

SHOEMAKER, Amy; VARE, Sugar. Edmonds' Blossom Algorithm. **Stanford University**, Stanford, v. 323, n. 1, p. 1-27, jun./2016. Disponível em: https://stanford.edu/~rezab/classes/cme323/S16/projects_reports/shoemaker_vare.pdf. Acesso em: 8 dez. 2023.

WIKIPÉDIA. **Desigualdade triangular**. Disponível em: https://pt.wikipedia.org/wiki/Desigualdade_triangular. Acesso em: 8 dez. 2023.

WIKIPÉDIA. **Eulerian path**. Disponível em: https://en.wikipedia.org/wiki/Eulerian_path. Acesso em: 8 dez. 2023.

WIKIPÉDIA. **Matching (graph theory)**. Disponível em: https://en.wikipedia.org/wiki/Matching_(graph_theory). Acesso em: 8 dez. 2023.