Análise de Algoritmos para o Problema do Caixeiro Viajante

Daniel Oliveira Barbosa

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

Resumo. *Nesse artigo tratamos de três algoritmos para resolver o problema do caixeiro viajante: branch and bound, twice around the tree e christofides. Para cada um dos algoritmos, entramos em detalhes sobre as decisões de implementação, como heurísticas, ferramentas e bibliotecas usadas. Dado isso, fizemos uma análise teórica de complexidade em termos de tempo e espaço que foi comprovada pela análise experimental. Com os experimentos, pudemos observar que a linguagem de programação em que os algoritmos são implementados são decisivos para atingir resultados satisfatórios em tempo hábil. Assim, pudemos concluir que há um inevitável trade-off entre qualidade da solução e tempo de execução.*

1. Introdução

Um dos problemas mais canônicos da ciência da computação é o problema do caixeiro viajante. Dado a vasta aplicabilidade do problema (que abreviaremos por TSP - traveling salesman problem) no mundo real, vale uma análise aprofundada sobre algumas de suas soluções. Vamos tratar de uma solução exata do problema que possui um custo computacional exponencial, e outras duas soluções de custo polinomial, porém com um compromissos quanto à exatidão:

- Branch and Bound (solução exata)
- Twice Around the Tree (solução aproximada)
- Christofides (solução aproximada)

Para tanto, iniciaremos constatando as decisões de implementação, tais como as heurísticas, ferramentas e bibliotecas usadas. Após isso, faremos uma análise teórica de cada algoritmo para tentarmos prever o seu comportamento em casos de teste, na análise experimental. Por fim faremos uma comparação teórica e prática do desempenho das soluções, atestando o melhor caso de cada um. Dessa maneira, encerraremos o trabalho tendo uma visão geral do que pudemos conluir das implementações e suas análises.

2. Definindo o Problema do Caixeiro Viajante - TSP

Para iniciarmos, vamos passar por uma breve explicação do problema tratado no artigo: o TSP. Esse problema consiste em um conjunto de cidades que são todas interligadas entre si. Um caixeiro viajante precisa visitar todas as cidades, porém para ir de uma cidade a outra há um custo associado. Logo, queremos descobrir qual é o caminho (seqência de cidades) pelo qual o caixeiro viajante deve seguir para passar por todas as cidades com o menor custo possível, terminando na cidade inicial.

Com isso, esse problema é presente em outros casos de aplicações práticas como logística, design de circuitos e até sequenciamento de DNA. Contudo, esse problema é

NP-Completo, fato que pode ser verificado pela redução do problema do ciclo hamiltoniano a ele. Não provaremos isso aqui, porém essa colocação já nos mostra que é um problema difícil..

Dado uma instância do TSP de **n** cidades, qualquer conjunto de n cidades que não se repetem é uma solução válida para o problema. Dessa maneira, a complexidade assintótica de um algoritmo de força bruta (combinatorial) para o problema é da ordem de **O(n!)**. Ou seja, até para instâncias pequenas como n = 50 o TSP é muito custoso de se resolver, fato que veremos mais a frente.

3. Implementações e Decisões de Projeto

3.1. Modelagem do Problema

Dado a similaridade do problema com grafos, decidimos modelá-lo com esse objeto matemático. O mapeamento será feito da seguinte maneira:

- Cada cidade corresponderá a um nó no grafo
- O caminho entre duas cidades corresponderá à aresta entre dois nós
- O custo associado a ir de uma cidade a outra equivalerá ao peso da aresta que liga um nó a outro

Dado que todas as cidades são interligadas entre si, o grafo é completo. Para a implementação do grafo, decidimos por utilizar uma matriz de adjascências pela facilidade do acesso direto ao peso de uma aresta.

3.2. Branch and Bound

3.2.1. Ideia Geral do Algoritmo

DELETE Uma solução do problema do TSP é uma sequência de vértices. Dado isso, podemos pensar na computação do problema do TSP como uma árvore de busca, onde a cada nível da árvore, decidimos qual nó incluiremos dentro da solução (dentre os que ainda não foram inclusos), e as folhas da árvore representam soluções completas. Porém, veja que podemos usar o backtracking para eliminarmos ramos de computação tão logo sejam deduzidos serem inválidos. Veja que no TSP, basta que uma solução não tenham nós que se repetem para ser uma solução válida. Logo, todos os ramos de computação onde um certamente levará a soluções que possuem nós repetidos já podem ser podados por invalidez.

Além disso, podemos calcular estimativas (limites inferiores) para o custo dos ramos de computação e usá-las para eliminar os ramos que certamente terão maior custo que a melhor solução encontrada até o momento. Assim, aliando as podas de ramos inválidos com as podas dos ramos menos ótimos, temos o algoritmo do branch and bound. DELETE END

Uma solução válida do problema do TSP é uma sequência de vértices. À vista disso, podemos pensar na computação do problema do TSP como uma árvore de busca, onde a cada nível da árvore, decidimos qual nó incluiremos dentro da solução (dentre os que ainda não foram inclusos). Nesse caso, as folhas da árvore representariam soluções completas, ou seja, circuitos hamiltonianos do grafo. Porém, ao invés de passar por todos

os ramos da árvore de busca, podemos calcular estimativas (limites inferiores) para o custo dos ramos de computação e usá-las para eliminar os ramos que certamente terão maior custo que a melhor solução encontrada até o momento. Assim, temos o branch and bound: um algoritmo que explora toda a árvore de busca, eliminando ramos de computação tão logo eles forem determinados como menos ótimos que a melhor solução encontrada.

3.2.2. Implementação

Inicialmente, a intenção era implementar o algoritmo apenas em python, visto sua simplicidade de uso. Contudo, a falta de eficiência do python foi tanta que até para as menores instâncias dos testes, o programa nem sequer chegou a um nó folha (solução completa) durante o limite de 30 minutos estipulado. Tentamos usar estratégias como iniciar com uma solução trivial e usar o decorador **@njit** da biblioteca **numba** para otimizar a computação, porém ainda assim não foi possível chegar a um nó folha nas menores instâncias de teste. Ainda não satasfeitos, queríamos explorar os limites de quais instâncias poderíamos pelo o menos encontrar alguma solução, mesmo não sendo a ótima. Por isso, optamos por fazer, além das implementações com e sem solução trivial em python, mais duas implemenações do algoritmo, totalizando nos seguintes:

- Implementação em python inicializando o algoritmo sem uma solução trivial
  - Inicialmente temos uma solução vazia e um melhor custo atual infinito
  - Usamos o decorador @njit e operações vetorizadas em numpy nessa implementação para tentarmos extrair o máximo do python
- Implementação em python inicializando o algoritmo com uma solução trivial
  - Ao invés de iniciarmos a computação com uma solução vazia e melhor custo atual infinito, podemos iniciar com uma solução trivial da sequência numérica dos nós: 0, 1, 2, 3, ..., n (dado um grafo de n nós) e o melhor custo atual como o custo dessa sequência.
  - Isso é possível pois, como mencionado anteriormente, qualquer sequência de vértices que não se repetem é uma solução válida
  - Com essa otimização, podemos mais rapidamente começar a podar ramos da árvore de busca e reduzir o tempo de execução do programa, ao invés de esperar chegar a um nó folha para apenas então atualizar o valor do melhor custo até o momento e possibilitar podar ramos.
  - Ainda usamos o decorador @njit e operações vetorizadas em numpy nessa implementação para tentarmos extrair o máximo do python
- Implementação em C++ sem solução inicial trivial
  - Essa implementação é equivalente à implementação em python inicializada sem uma solução trivial
- Implementação em C++ com solução inicial trivial
  - Essa implementação é equivalente à implementação em python inicializada com uma solução trivial

Assim, dada essas 2 implementações em python e mais 2 em C++, pudemos ter uma visão mais geral sobre como o algoritmo do branch and bound funciona em diferentes contextos de linguagens de programação e diferentes estratégias de otimização. Mais a frente, na seção de análise experimental, veremos o resultado dessas variações.

Independente da implementação, optamos pelo best first search como a forma de travessia da árvore de busca dado que ela prioriza a exploração dos ramos mais promissores. Assim aumentamos a probabilidade de se encontrar uma primeira melhor solução com menor custo possível. Consequentemente, aumentamos o número total de nós podados e a eficiência do programa. Para viabilizar o best first search utilizamos uma fila de prioridade, para garantir que os ramos de menor custo sejam explorados primeiro. Nos algoritmos em python, implementamos essa estrutura de dados através da biblioteca **heapq**. Em C++ utilizamos o **priority_queue** da biblioteca **queue**.

Juntamente a isso, para os calcular a estimativa (bound) de um ramo de computação, seguimos a lógica descrita a seguir. Para cada nó não incluso na solução, somamos o valor das suas duas arestas de menor peso. Para os nós que já foram inclusos na solução, se for o último ou o primeiro nó da solução, uma das arestas já terá sido escolhida pela solução, então devemos somar a aresta já determinada pela solução com uma outra aresta que seja de menor peso para esse nó. Para os nós que já foram inclusos na solução, porém não são o último ou o primeiro nó da sequência, a própria solução já nos dá as aresta, então devemos somar as duas arestas já determinadas pela solução para esse nó. Assim, somando esse par de arestas para cada nó do grafo, dividimos por 2 para contabilizar a repetição de arestas e pegamos o teto desse resultado como a estimativa (limite inferior) para o custo de um ramo de computação.

3.3. Twice Around The Tree

3.3.1. Ideia Geral do algoritmo

Um dos requisitos do problema do TSP é que todos os nós do grafo sejam visitados e que usamos o menor custo possível para visitar cada um. Dessa maneira, é natural que uma árvore geradora mínima (que abreviaremos por MST - minimum spanning tree) seja útil para nos auxiliar a encontrar uma solução para o TSP. Contudo, por ser uma árvore, a MST não pode ser solução, pois pelo o menos um vértice terá de ser repetido ao caminhar por essa árvore. Para resolvermos isso, se duplicarmos as arestas da MST e caminharmos por esse multigrafo em pré-ordem, teremos um circuto euleriano. Se pularmos os nós já visitados nesse circuito, teremos um circuito hamiltoniano, que é uma solução válida do TSP.

Dessa forma, o algoritmo twice around the tree consiste em encontrar a MST do grafo dado como entrada, e depois gerar uma solução a partir do circuito euleriano pulando os nós repetidos. Veja que esse algoritmo não garante uma solução exata como o branch and bound, porém podemos afirmar que possui constante de aproximação de 2. Não provaremos isso aqui, mas vale ressaltar que isso só é possível dado que o peso das arestas é determinado por uma métrica, onde vale a desigualdade triangular. Logo, podemos garantir que ”pular” um nó já visitado tenha custo menor ou igual a não pulá-lo.

3.3.2. Implementação

Implementado apenas em python, nos valemos da biblioteca **networkx** como ferramenta usada para armazenar o grafo e fazer os cálculos necessários com o ele. Usamos a função **minimum_spanning_tree** para calcular a MST que usa como padrão o algoritmo

de Kruskal. Por outro lado, foi observado que era desnecessário duplicarmos as arestas do grafo, gerarmos o circuito euleriano e depois iterar pelos nós incluindo os não visitados na solução. Ao invés disso, apenas caminhamos pela MST em pré ordem com depth first search (DFS) usando a função **dfs_preorder_nodes**, incluindo os nós sempre que não visitados na solução. Dessa forma, temos uma solução equivalente de forma mais eficiente.

3.4. Christofides

3.4.1. Ideia Geral do algoritmo

O algoritmo de Christofides é uma otimização feita em cima do twice around the tree em relação à qualidade da solução. O seu idealizador observou que, no twice around the tree, a fase de duplicação das arestas da MST só era necessária para os nós de grau ímpar. Isso acontece pois, era possível chegar até eles sem repetir nós, mas voltar deles implicava necessariamente passar por nós já visitados. Assim, ele propôs adicionar ao grafo apenas as arestas necessárias para reduzir a quantidade de nós que sejam repetidos devido aos nós de grau ímpar. Isso é feito primeiro calculando o matching de peso mínimo do subgrafo induzido dos nós de grau ímpar, depois adicionando as arestas do matching à MST e finalmente calculando o circuito hamiltoniano solução a partir do circuito euleriano (pulando os vértices já visitados). Com essas modificações, a solução passa a ter constante de aproximação de 1.5 que não provaremos aqui, mas é viabilizada também pelo fato de que o peso das arestas é determinado por uma métrica.

3.4.2. Implementação

Como o twice around the tree, também fizemos uma implementação apenas em python e usamos a biblioteca **networkx** para lidar com o grafo. Assim usamos as seguintes funções:

- **Calcular a MST**: usamos novamente a função **minimum_spanning_tree**
- **Calcular o matching de peso mínimo**: usamos a função **min_weight_matching** (baseado no algoritmo de Blossom)
- **Calcular o circuito euleriano**: usamos a função **eulerian_circuit**
- **Calcular o circuito hamiltoniano solução**: iteramos pelos nós do circuito euleriano incluindo-os na solução caso ainda não tivessem sido visitados

4. Análise de Complexidade

Para o algoritmo de branch and bound, apesar de podarmos os ramos menos ótimos, no pior caso ainda teríamos de computar todos os ramos. Como qualquer sequência de nós sem repetição é uma solução válida, o custo assintótico de pior caso do algoritmo é **O(n!)**, ou seja, exponencial. Isso será percebido claramente na análise experimental. Assim, trocamos a execução em tempos razoáveis por exatidão da solução.

Para o twice around the tree, o custo assintótico é dominado pelo cálculo da MST. Dado que isso é feito através do algoritmo de Kruskal, temos que a comlexidade do twice around the tree é da ordem de $\mathbf{O}(|\mathbf{E}|\mathbf{log}|\mathbf{V}|)$. Já para o Christofides, o custo assintótico é dominado pelo cálculo do matching de peso mínimo, que possui uma complexidade de

$\mathbf{O}(|\mathbf{V}|^{\mathbf{3}})$. Assim, em ambos os algoritmos são polinomiais, em detrimento da qualidade da solução.

Como pode-se perceber o custo do branch and bound é maior que o custo do twice arount the tree que é maior que o custo do christofides, enquanto a relação de ordem da qualidade da solução é inversa a essa. Ou seja, de modo geral, podemos inferir que quanto mais exato a solução, maior o custo computacional necessária para obtê-la. Esse fato também será percebido na análise experimental.

5. Análise Experimental

Na nossa análise experimental usamos intâncias de teste do TSPLIB para o problema TSP simétrico, cujas soluções ótimas já são conhecidas. Optamos por analisar as seguintes variáveis:

- Se foi capaz de chegar a um nó folha (solução mesmo que não seja ótimo)
- Tempo de execução
- Qualidade da solução obtida em relação à solução ótima já conhecida para a instância
- Quantidade máxima de memória usada durante a execução
- Quantidade de podas feitas durante a execução

5.1. Chegada em nó folha

A intenção inicial do nosso trabalho era comparar os três algoritmos sendo implementados apenas em python. Porém, durante nossos testes, as duas implementações do branch and bound com python (com e sem solução trivial) sequer chegavam a um nó folha dentro do tempo limite de 30 minutos, mesmo nas menores instâncias como o "eli51" e o "berlin52". A implementação com solução trivial até retornava uma solução, porém era a própria solução trivial com a qual o algoritmo foi inicializado.

Com isso surgiu o questionamento: quanto que a falta de eficiência do python influência a incapacidade do programa de chegar a um nó folha? Assim, decidimos por fazer duas implementações equivalentes em C++ para verificar o quanto essa ineficiência pode afetar os resultados. Enquanto as implementações em python não foram capazes de chegar a nenhum nó folha, a implementação em C++ sem solução trivial foi capaz de chegar a nós folhas para instâncias de até 657 de nós. Com isso, pudemos chegar a nossa primeira observação das análises: a falta de eficiência do python possui um impacto enorme na computação das instâncias. Dessa maneira, a implementação de algoritmos com custo exponencial em linguagens pouco eficientes deve ser feita ciente dos resultados menos promissores.

Ademais, foi observado um fenômeno interessante para as implementações do branch and bound usando solução trivial, tanto para python quanto para C++, nunca chegavam a nós folhas. Isso se deve ao fato de que os algoritmos acabavam passando todo o tempo de execução podando ramos menos ótimos ao invés de descer a árvore de busca até chegar a um nó folha. Como os algoritmos twice around the tree e christofides não se valem dessa abstração de árvore de busca, eles não foram analisados quanto a essa variável.

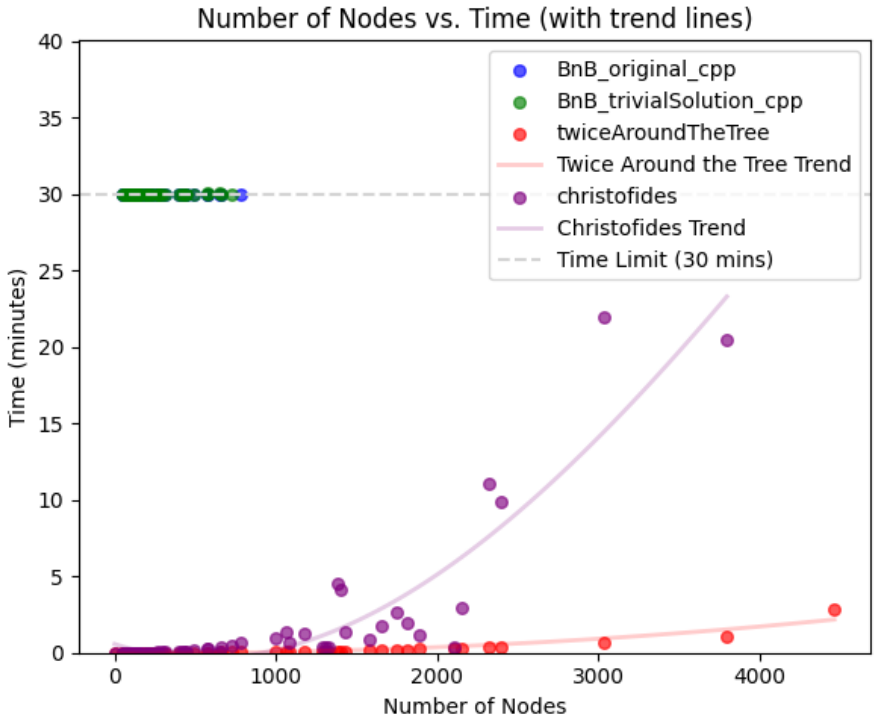

Figure 1. Número de Nós vs. Tempo de Execução (com linha de tendências)

5.2. Tempo de Execução

Como esperado, o custo exponencial do branch and bound impediu que o programa terminasse de executar por completo dentro do limite de tempo de 30 minutos para qualquer instância. Com isso, no gráfico da figura 1, podemos observer que todas as intâncias computadas pelas implementações do branch and bound em C++ estão algomeradas na linha tracejada em cinza que indica o limite de tempo de 30 minutos. Além disso, dado que as implementações em python do branch and bound não retornavam uma solução e um custo a não ser o que lhe foi atribuído durante sua inicialização, eles não foram analisados em relação às próximas métricas pois não agregariam nenhuma informação nova.

Por outro lado, os algoritmos aproximativos conseguiram terminar de executar e retornar uma solução para as instâncias de até 4461 nós para o twice around the tree e até 3795 nós para o christofides. A diferença no número se deve a complexidade assintótica do christofides de $\mathbf{O}(|\mathbf{V}|^{\mathbf{3}})$ ser maior que do twice around the tree de $\mathbf{O}(|\mathbf{E}|\mathbf{log}|\mathbf{V}|)$. Esse fato pode ser observado claramente pelas diferentes taxas de crescimento das linhas de tendência em relação ao número de nós.

5.3. Qualidade da Solução

No quesito de qualidade da solução, as implementações de branch and bound não foram capazes de retornar uma solução exata por conta do limite de tempo. Na versão com do branch and bound C++ com solução trivial, as soluções obtidas foram no mínimo 1.03 vezes e no máximo 14.10 vezes pior que a solução ótima em uma distribuição relativamente uniforme. Já para o C++ sem solução trivial, as soluções obtidas foram no mínimo 4.66 vezes e no máximo 93.72 vezes pior que a solução ótima, onde a qualidade

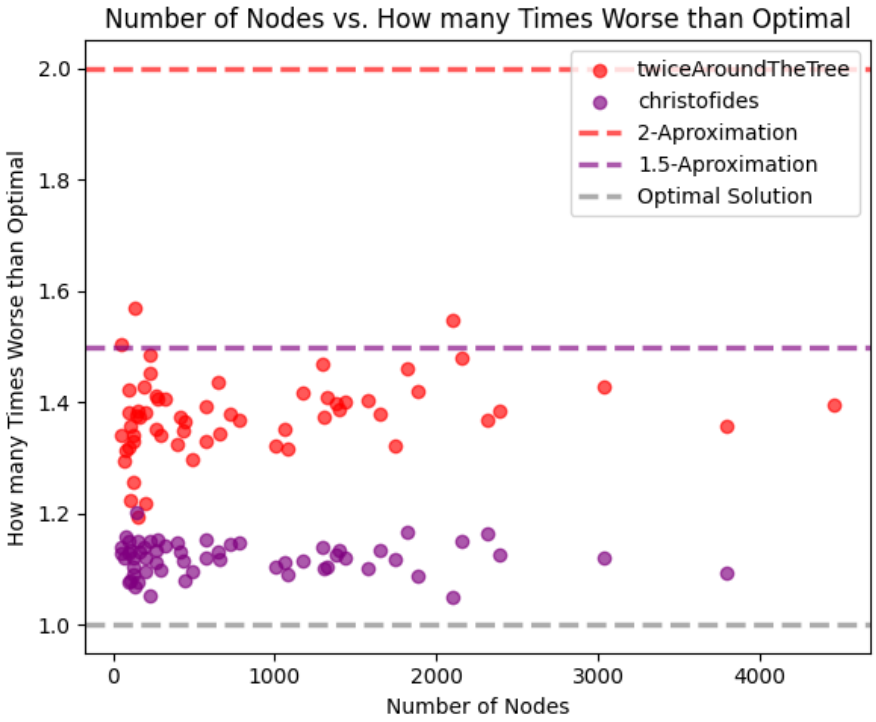

Figure 2. Número de Nós vs. Quantas Vezes a Solução Computada é Pior que a Solução Ótima

da solução obtida era inversamente proporcional ao número de nós. Isso é de se esperar já que, quanto menos nós, mais o algoritmo consegue chegar a nós folhas, convergindo mais para a solução ótima dentro do limite de tempo.

Já para os aproximativos, pudemos atestar na prática que o twice around the tree é no máximo 2 vezes pior e o christofides é no máximo 1.5 vezes pior que a solução ótima. Isso pode ser observado no gráfico da figura 2 onde nenhuma das instâncias ultrapassa seu respectivo limite de c-aproximação.

5.4. Quantidade de Podas (Cortes) de Ramos

Em relação à quantidade de podas feitas, os algoritmos aproximativos não são considerados pois não usam o conceito de árvore de busca. Durante a análise dessa variável, foi observado que o número de cortes é inversamente proporcional ao número de nós da instância. Esse resultado era de esperar dado que, quando mais profunda a árvore de busca, mais tempo é gasto descendo ela do que fazendo podas. Veja o gráfico da figura 3 que corrobora essa conclusão.

5.5. Memória Máxima Usada

Por fim, a análise de memória foi condizente com as observações feitas anteriormente. Primeiramente, observamos que a memória utilizada, em média, nos algoritmos aproximativos é significativamente menor do que do branch and bound, como visto na tabela 1. Isso se deve ao fato de que a fila de prioridade utilizada no branch and bound tende a crescer exponencialmente como visto no gráfico da figura 4.

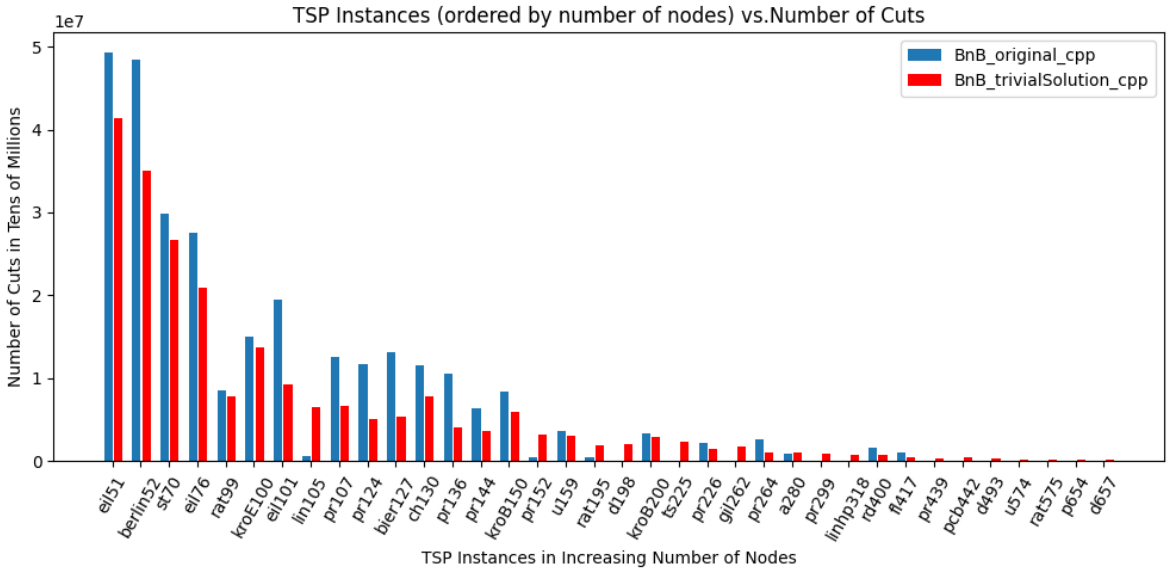

Figure 3. Número de Nós vs. Cuts

Por outro lado, podemos ver também na figura 4 que a implementação com solução trivial usa pouca memória em relação a implementação sem solução trivial. Isso se deve ao fato de que, com uma solução trivial, já se pode podar ramos logo de início. Assim, o algoritmo chega menos profundo na árvore e consequentemente, menos nós são colocados na fila de prioridade.

Implementação	Memória Utilizada em Média (KB)
BnB_original_cpp	6956.87
BnB_trivialSolution_cpp	3404.44
christofides	33.69
twiceAroundTheTree	35.72

Table 1. Memória Utilizada em Média (KB) por cada Implementação

6. Conclusão

Em suma, nesse trabalho implementamos e analisamos diferentes soluções para o problema do caixeiro viajante. Pudemos notar primeiramente que o uso de linguagens de programação ineficiêntes em termos computacionais para resolver problemas exponenciais deve ser feita com muita cautela, dado os prejuízos em termos de resultados que podem haver. Além disso, pudemos verificar o impacto que o custo n! de um algoritmo pode ter em relação a memória e tempo necessárias para a execução através dos gráficos e valores exponenciais obtidos. Também pudemos observar o respeito dos algoritmos aproximados das suas respecitvas contantes de aproximação e também as suas distintas complexidades assintóticas pelos gráficos de qualidade da solução e tempo, respectivamente.

Em termos comparativos, pudemos comprovar nas nossas análises experimentais as previsões feitas pela análise teórica: o custo exponencial do branch and bound limita severamente a tratabilidade, quando comparados em relação aos algoritmos aproximados polinomiais. Com isso, podemos concluir que, para casos onde a computação de

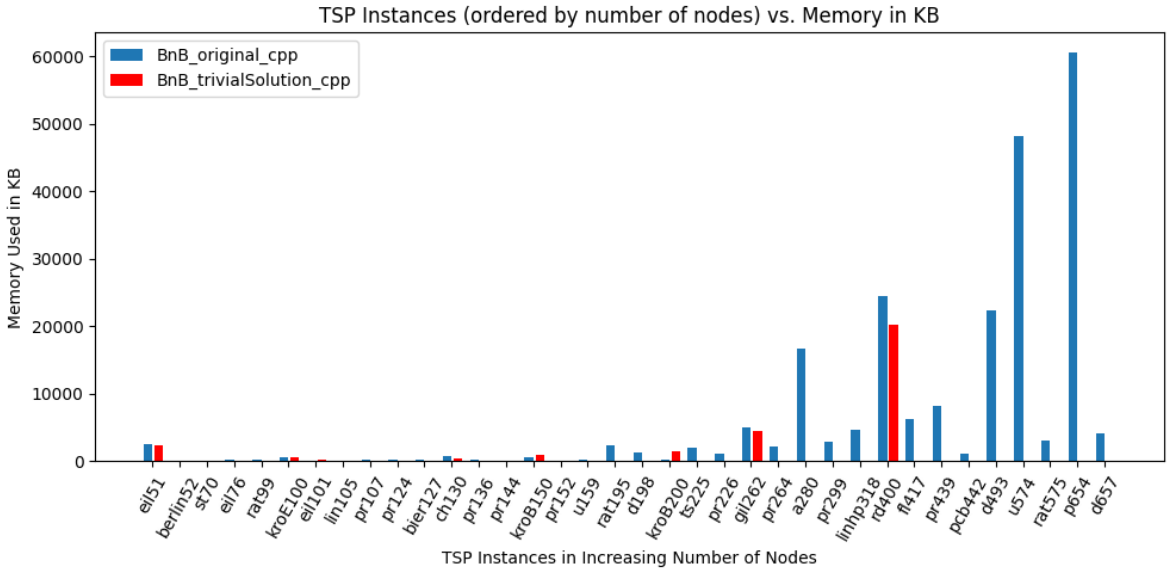

Figure 4. Número de Nós vs. Memoria

uma instância do TSP precisa ser feito rapidamente e/ou usando poucos recursos computacionais com uma tolerância por erros, os aproximativos são ideais. Contudo, se a solução precisa ser exata, usar uma solução trivial pode ajudar em reduzir a necessidade de recursos computacionais.

Portanto, concluímos o trabalho com um conhecimento mais abrangente sobre linguagens de programação e seus efeitos sobre implementações, o TSP e suas diferentes soluções, e os resultados teóricos e práticos deles.

7. Referências

Levitin, A. (2011). Introduction to the design and analysis of algorithms (3rd ed.). Upper Saddle River, NJ: Pearson.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). London, England: MIT Press.