RELATÓRIO - TP 2
ALGORITMOS II, UFMG

Ester Sara Assis
2021031785

Júlia Paes de Viterbo
2021032137

ABSTRACT

O estudo de algoritmos na graduação começa desde os primeiros semestres no curso de Ciência da Computação. À medida em que são introduzidas soluções cada vez mais complexas, constrói-se a compreensão de que existem problemas para os quais não existe resposta exata em todos os casos. Nesse contexto, insere-se o estudo de algoritmos que se utilizam de estratégias de corte, como *Backtracking* e *Branch and Bound*, e de metodologias aproximativas, baseadas em heurísticas. Apesar de ser um dos problemas mais citados no meio acadêmico desde aqueles primeiros semestres, o Problema do Caixeiro Viajante (TSP) é apenas entendido por completo após o estudo das tais estratégias. Nesse sentido, o presente estudo aprofunda-se na investigação dele, centrando sua análise tanto no comportamento, quanto no desempenho de diferentes algoritmos para o problema.

1. INTRODUÇÃO

Este trabalho investiga o desempenho de algoritmos para o Problema do Caixeiro Viajante em sua versão euclidiana (**Problema do Caixeiro Viajante Euclidiano**). Foi solicitada análise de desempenho de três algoritmos:

1. Branch and Bound (3);
2. Twice Around The Tree (4);
3. Christofides (5).

e usadas instâncias compiladas da biblioteca `TSPLIB` [1] como teste. Para implementação desse problema, utilizamos um classe geral Traveling Salesman Problem (2), com definições padrões reaproveitadas para todos os algoritmos, e classes individuais para cada um deles. Além disso, para optimizar o processamento de arquivos teste, uma classe Traveling Salesman Problem Runner (6) também foi introduzida.

2. TRAVELING SALESMAN PROBLEM

2.1 Apresentação

A classe Traveling Salesman Problem é a responsável por quatro etapas importantes: Construir o grafo com base nos arquivos teste, executar os três algoritmos de forma assíncrona, calcular o limiar de qualidade final de cada um e, por fim, gerar o arquivo de saída contendo as métricas finais.

2.2 Implementação

2.2.1 Construção do Grafo

A construção do grafo é feita com base no arquivo teste e sua formatação definida na especificação da TSPLIB95 [1], de forma que instâncias cujo tipo não seja Euclidiano 2D são puladas, assim como aquelas de dimensão maior que 3000 (valor máximo de nodos suportados para computação na máquina usada). Tendo as coordenadas do arquivo, calculam-se os pesos das arestas do grafo completo utilizando a equação (1).

$$\text{weight} = \left\lceil \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \right\rceil \tag{1}$$

Por fim, utiliza-se a biblioteca *networkx* para criar uma instância *Graph*, que será a principal estrutura de dados do projeto.

2.2.2 Execução de Algoritmos

Para executar os três algoritmos foi usada uma abordagem assíncrona, de forma que todos os três executam de forma simultânea. Com isso, é realizado o controle de limite de tempo (*timeout*). Para tal, a biblioteca usada foi *asyncio*. Para que os algoritmos não sejam bloqueantes entre si por conta dessa abordagem e o tempo de espera seja grande, em todos os loopings internos dos mesmos foi necessário usar o comando *await asyncio.sleep(0),* que libera o controle para o gerenciador de tarefas assíncrono.

2.2.3 Cálculo do Limiar de Qualidade

Para calcular o limiar de qualidade, usou-se o erro relativo com base no valor ótimo, mostrado na equação (2). Esse valor é obtido por meio de um arquivo passado pelo usuário (caso esse arquivo não seja identificado, a implementação continua funcional, mas a métrica de qualidade é invalidada com o valor "-1").

$$\max \alpha(I) = \frac{r(I) - \text{OPT}(I)}{\text{OPT}(I)} \tag{2}$$

2.2.4 Geração de Arquivo de Saída

O arquivo de saída é gerado no layout é do tipo *instance.tsp.result* e seu formato é como o exemplo abaixo, para cada um dos algoritmos. Caso o tempo de *timeout* seja atingido, todos os valores ficam como NA.

Minimum Path: `[0, 48, ...]`
Minimum Cost: 9513
Execution Time: 0.0961 seconds
Memory Usage: 56 bytes
Relative Error: 0.2613

3. BRANCH AND BOUND

3.1 Apresentação

O algoritmo de *Branch and Bound* para TSP utiliza-se de podas durante a exploração de caminhos possíveis a partir de um vértice inicial com base no cálculo de um *lower bound*. Para tanto, somam-se as duas menores arestas ainda não utilizadas no caminho àquelas já utilizadas e divide-se o resultado por dois para cada possibilidade de escolha. Nesse sentido, ramos com *lower bounds* maiores do que o(s) ótimo(s) encontrado(s) não são explorados.

Seguindo essa lógica, a exploração dos nós se dá pela lógica de **Breadth-First Search**, em que a prioridade da fila se dá com base no valor de *lower bound* calculado para os vértices. A **Figura 1** mostra um exemplo de execução do algoritmo em um grafo simples.

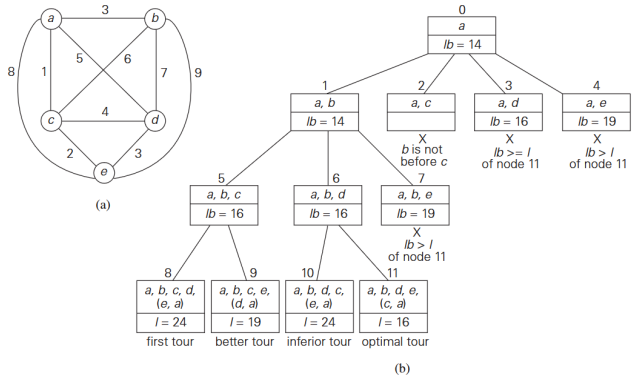

FIGURE 12.9 (a) Weighted graph. (b) State-space tree of the branch-and-bound algorithm to find a shortest Hamiltonian circuit in this graph. The list of vertices in a node specifies a beginning part of the Hamiltonian circuits represented by the node.

Figure 1. `LEVITIN` [4], cap. 12, sec. 2

3.2 Implementação

A implementação do algoritmo foi feita com base no pseudocódigo abaixo (Algoritmo 1), retirado do slide da aula 12 [2]:

Com isso, as estruturas de dados usadas foram implementadas da seguinte forma:

1. **Fila de Prioridade**: implementada usando a biblioteca `heapq`. A estrutura do heap permite que os nós sejam processados com base no menor limite inferior, garantindo que o nó mais promissor seja explorado primeiro;

2. **Lista**: o caminho parcial (sequência de vértices visitados até o momento) é armazenado como uma lista e é atualizado conforme novos vértices são visitados;

3. **Matriz de Adjacência**: o grafo, implementado usando a biblioteca `networkx`, tem os custos acessados via consulta de sua matriz de adjacência, para facilitar o acesso rápido.

Algorithm 1 Branch and Bound TSP

```
1: procedure BNB-TSP(A, n)
2:     root ← (bound([0]), 0, 0, [0])
3:     queue ← heap([root])
4:     best ← ∞
5:     sol ← ∅
6:     while queue ≠ ∅ do
7:         node ← queue.pop()
8:         if node.level > n then
9:             if best > node.cost then
10:                best ← node.cost
11:                sol ← node.s
12:            end if
13:        else if node.bound < best then
14:            if node.level < n then
15:                for k = 1 to n − 1 do
16:                    if k ∉ node.s and A[node.s[−1]][k] = 1 and then
17:                        bound(node.s ∪ {k}) < best
18:                        queue.push((bound(node.s ∪ {k}),
19:                        node.level + 1,
20:                        node.cost + A[node.s[−1]][k],
21:                        node.s ∪ {k}))
22:                    end if
23:                end for
24:            end if
25:            if A[node.s[−1]][0] = 1 and bound(node.s ∪ {0}) < best then
26:                if ∀i ∈ node.s then
27:                    queue.push((bound(node.s ∪ {0}),
28:                    node.level + 1,
29:                    node.cost + A[node.s[−1]][0],
30:                    node.s ∪ {0}))
31:                end if
32:            end if
33:        end if
34:    end while
35: end procedure
```

3.3 Métricas

Para calcular as métricas desse algoritmo, foi utilizado:

1. **Espaço**: O tamanho da pilha ao final do processamento e o tamanho do vetor de caminho mínimo com base na biblioteca sys.

2. **Tempo**: Dada pela biblioteca *time*.

3. **Qualidade**: Definição padrão dada em Traveling Salesman Problem [2].

4. TWICE AROUND THE TREE

4.1 Apresentação

O algoritmo de *Twice Around The Tree* para TSP é um algoritmo aproximativo com fator de aproximação 2 (até duas vezes o custo ótimo) que se baseia no caminhamento em sua árvore geradora mínima.

4.2 Implementação

A implementação do algoritmo foi feita com base no pseudocódigo abaixo (Algoritmo 2), retirado do slide da aula 13 [3]:

Algorithm 2 APPROX-TSP-TOUR(G, c)

1: select a vertex $r \in G.V$ to be a "root" vertex
2: compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r)
3: let H be a list of vertices, ordered according to when they are first visited in a preorder tree walk of T
4: **return** the hamiltonian cycle H

Para realizar esse passo a passo, uma vez que já há o grafo como uma instância Graph do networkx, foram utilizadas funções da mesma biblioteca: nx.minimum_spanning_tree e nx.dfs_preorder_nodes. A primeira gera a estrutura de dados principal do algoritmo, a árvore geradora mínima, como uma instância do tipo Graph. Com isso, para finalizar o algoritmo, foi inserido o primeiro vértice ao caminhamento pré ordenado gerado, de forma a transformá-lo no ciclo hamiltoniano.

4.3 Métricas

Para calcular as métricas desse algoritmo, foi utilizado:

1. **Espaço**: O tamanho da Minimum Spanning Tree, do vetor de caminhamento pré ordem e do ciclo hamiltoniano ao final do processamento com base na biblioteca sys.

2. **Tempo**: Dada pela biblioteca *time*.

3. **Qualidade**: Definição padrão dada em Traveling Salesman Problem [2].

5. CHRISTOFIDES

5.1 Apresentação

O algoritmo de *Christofides* para TSP é um algoritmo aproximativo com fator de aproximação 1.5 (até 1.5 vezes o custo ótimo) que, assim como o Twice Around the Tree, se baseia em sua árvore geradora mínima, porém, nesse usa-se a técnica de emparelhamento perfeito.

5.2 Implementação

A implementação do algoritmo foi feita com base no pseudocódigo abaixo (Algoritmo 3), feito com base no slide da aula 13 [3]:

Algorithm 3 CHRISTOFIDES-TSP(G)

1: Compute a minimum spanning tree T of G
2: Let I be the set of vertices with odd degree in T. Compute M, a minimum weight perfect matching on the subgraph induced by I
3: Let G' be the multigraph formed by combining the edges of M and T. Compute an Eulerian circuit in G' (using a Depth-First Search)
4: Simplify the Eulerian circuit by removing duplicate vertices, replacing subpaths $u \rightarrow w \rightarrow v$ with the edge $u \rightarrow v$
5: **return** the resulting Hamiltonian cycle

Para isso, da mesma forma que foi feito para o algoritmo Twice Around The Tree, foram utilizadas funções próprias da biblioteca *networkx*. Porém, optou-se por implementar a busca por emparelhamento perfeito utilizando o subgrafo induzido gerado pelos nós de grau ímpar da árvore mínima, uma vez que a função padrão da biblioteca não gerava circuitos eulerianos quando unidos a mesma. Com isso, aplicou-se a função nx.eulerian_circuit e, com seu resultado, obteve-se o ciclo hamiltoniano ao iterar por suas arestas.

5.3 Métricas

Para calcular as métricas desse algoritmo, foi utilizado:

1. **Espaço**: O tamanho da Minimum Spanning Tree, do vetor de vértices ímpares, do subigrafo induzido, do emparelhamento perfeito mínimo, do grafo euleriano e de seu circuito e do ciclo hamiltoniano, ao final do processamento com base na biblioteca sys.

2. **Tempo**: Dada pela biblioteca *time*.

3. **Qualidade**: Definição padrão dada em Traveling Salesman Problem [2].

6. TRAVELING SALESMAN PROBLEM RUNNER

6.1 Apresentação

A classe Traveling Salesman Problem Runner é responsável por processar os arquivos com base na pasta de instâncias de forma assíncrona, facilitando com que o

usuário não tenha que desprender de 30 minutos a cada instância testada caso está não seja computável.

6.2 Implementação

O runner foi implementado de forma que, com base nos parâmetros de: caminho da pasta a ser processada, caminho da pasta que irá conter os arquivos processados, caminho da pasta que irá conter os resultados dos arquivos processados e o caminho do arquivo de valores ótimos, irá realizar o processamento da seguinte forma:

Ao processar um arquivo presente na pasta de origem, enviando-o para a classe anteriormente citada Traveling Salesman Problem, este será realocado para a pasta de processados e seu resultado gerado na mesma classe será colocado na pasta de resultados. Essa abordagem é seguida para que, no caso de muitas instâncias, o usuário possa processa-las por partes, sem necessidade de re-executar aquelas que já foram.

Como citado, esse processo ocorre de forma assíncrona: Foi usada a biblioteca *asyncio* com a funcionalidade de semáforo para processar x arquivos ao mesmo tempo. Esse valor é modificável, mas, caso seja alto, a memória interna não irá conseguir guardar todos os dados e um evento "Killed" ocorrerá, por isso, foi escolhido um valor máximo igual a 5.

7. EXPERIMENTOS

Observação

As instâncias fornecidas no TSPLIB95 [1] são de dimensão muito grande para que funcionem com o algoritmo de branch and bound no tempo útil definido (30 minutos), por isso, foram adicionadas instâncias menores para que a análise seja completa.

7.1 Análise de Tempo para Instâncias Pequenas (<50)

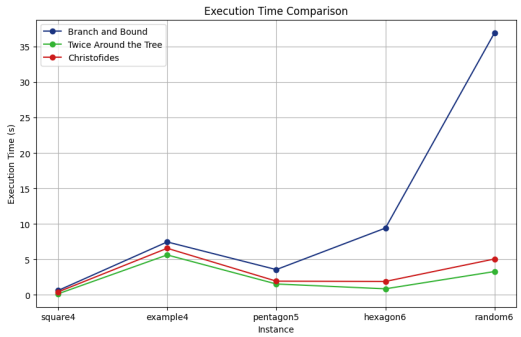

Figure 2. Comparação de Tempo de Execução para Instâncias Pequenas

Pela figura 2, percebe-se inicialmente um claro aumento no tempo de execução do algoritmo Branch and Bound em relação aos outros algoritmos conforme as instâncias aumentam de tamanho. Isso é esperado, uma vez que possui natureza exaustiva. Porém, algo notável nessa figura é o pico de tempo de execução em todos os 3 algoritmos para as instâncias *example4* e *random6*, o que se deve as características das mesmas: Diferentes das demais, que são geométricas e, portanto, possuem estrutura regular, os vértices e arestas são posicionados de forma aleatória, de forma que, para o Branch-and-Bound, mais combinações são possíveis e para o Twice Around the Tree, arestas longas podem ser inseridas devido as grandes lacunas entre grupos.

7.2 Análise de Memória para Instâncias Pequenas (<50)

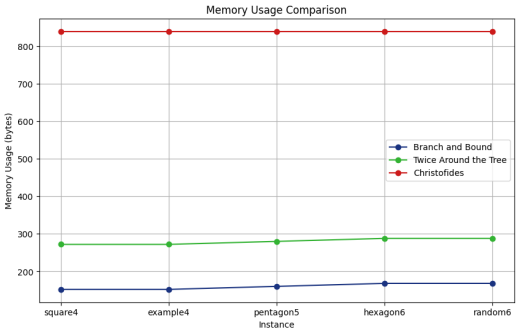

Figure 3. Comparação de Uso de Memória para Instâncias Pequenas

Pela figura 3, percebe-se uma clara vantagem do algoritmo Branch and Bound em relação aos demais, mantendo seu uso de memória equilibrado mesmo com aumento de instâncias. Isso é esperado, uma vez que o mecanismo de poda do mesmo economiza bastante memória. No algoritmo Twice Around, a necessidade de computar uma árvore geradora mínima aumenta seu uso de memória em relação ao anterior, mas ainda se mantém com consumo moderado. O algoritmo de Christofides, por sua vez, possui uma necessidade de memória consideravelmente maior, uma vez que sua implementação exige muitas estruturas e etapas intermediárias, como foi mostrado na seção x.

7.3 Análise de Qualidade para Instâncias Pequenas (<50)

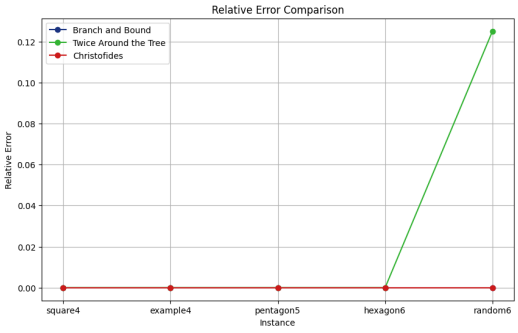

Figure 4. Comparação de Erro Relativo para Instâncias Pequenas

Pela figura 4, percebe-se que para instâncias pequenas, todos os algoritmos chegam bem próximos do valor ótimo. Para o Branch and Bound, por ser um algoritmo exaustivo, o ótimo sempre será encontrado, de forma que seu erro relativo sempre será 0. No algoritmo Twice Around, percebe que para a instância irregular *random6*, houve um erro relativo maior que zero, uma vez que por conta da irregularidade, a MST pode ter incluído arestas longas ou mal posicionadas. Esse impacto não ocorre no Christofides da mesma forma, pois sua implementação compensa o defeito do Twice Around ao corrigir arestas com o emparelhamento perfeito mínimo.

7.4 Análise de Tempo para Instâncias Médias (>50, <1000)

Figure 5. Comparação de Tempo de Execução para Instâncias Médias

Pela figura 5, podemos analisar alguns pontos interessantes: O tempo de execução do Twice Around the Tree é sempre menor que o do Christofides, o que é esperado, uma vez que o segundo é o primeiro mas com uma quantidade bem maior de passos e estruturas intermediários. Porém, um ponto curioso é a variação de tempo não dependente da dimensão da instância, mas sim das características particulares das mesmas. Um exemplo de análise específica para o trecho de eil51 até kroB200 onde pr144 > eil51 > kroB100 > kroB200, é que percebe-se que os problemas kro100 e kro200 tem distribuições mais uniformes, de forma que o cálculo é facilitado em relação a instâncias menos uniformes, porém menores. Também percebe-se que a instância eil51 é muito densa, o que pode aumentar seu tempo de execução pelo elevado número de arestas próximas. Um outro caso particular é o d657, que mesmo sendo uma instância com grande dimensão possui tempo de execução baixo: O problema dessa instância é um problema de perfuração, e analisando seus dados percebe-se que sua distribuição é muito uniforme e seu espaçamento entre nós é razoavelmente constante, reduzindo a complexidade das heurísticas.

7.5 Análise de Memória para Instâncias Médias (>50, <1000)

Pela figura 6, percebe-se que, em relação a memória, o tamanho da dimensão das instâncias tem maior efeito que as particularidades de cada uma, ainda que estas possam

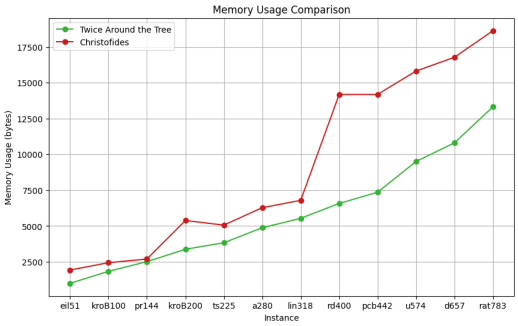

Figure 6. Comparação de Uso de Memória para Instâncias Médias

ser ligeiramente notadas. O maior ponto de interesse é o uso muito maior de memória do algoritmo Christofides, que possui elevações mais bruscas conforme o tamanho das instâncias cresce.

7.6 Análise de Qualidade para Instâncias Médias (>50, <1000)

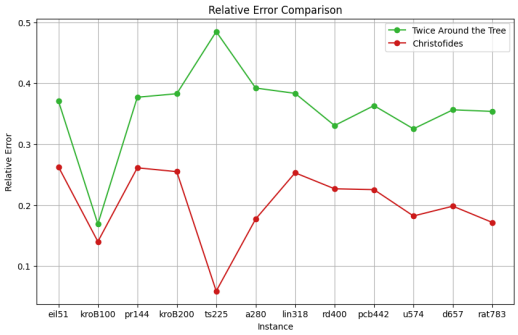

Figure 7. Comparação de Erro Relativo para Instâncias Médias

Pela figura 7, percebe-se que, exceto por algumas instâncias cujas particularidades surtem muito efeito, as taxas de erro relativo dos algoritmos se mantém relativamente constante, estando ao redor de 0.2 a 0.3 para Christofides e 0.3 e 0.4 para Twice Around. Isso é esperado, uma vez que o primeiro garante maior precisão ao utilizar estruturas de matching perfeito e circuito euclidiano para reduzir seu fator aproximativo de 2 para 1.5 em relação ao segundo. Uma particularidade interessante é da instância ts225, onde seu erro relativo é muito alto para Twice Around e muito baixo para Christofides. Apesar de ser necessária uma análise mais profunda para entender a razão, uma possibilidade é que essa instância seja um caso de borda onde o percurso é dominado por arestas muito longas, o que afeta o primeiro algoritmo, mas é corrigido pelo segundo, que retira redundâncias.

7.7 Análise de Tempo para Instâncias Grandes (>1000)

Pela figura 8 assim como para instâncias médias, o fator particular das instâncias tem grande efeito sobre essa

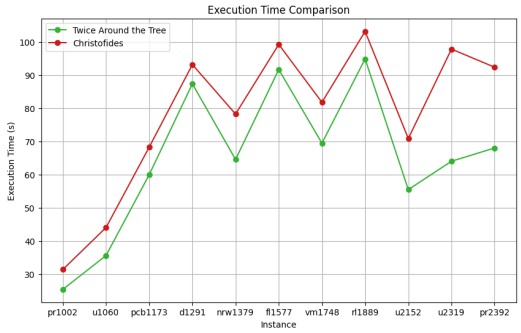

Figure 8. Comparação de Tempo de Execução para Instâncias Grandes

métrica, porém, percebe-se uma importância maior no tamanho da dimensão, uma vez que o tempo aumenta com as mesmas, ainda que com exceções. Christofides continua gastando mais tempo que o Twice Around, ainda que a diferença seja menor.

7.8 Análise de Memória para Instâncias Grandes (>1000)

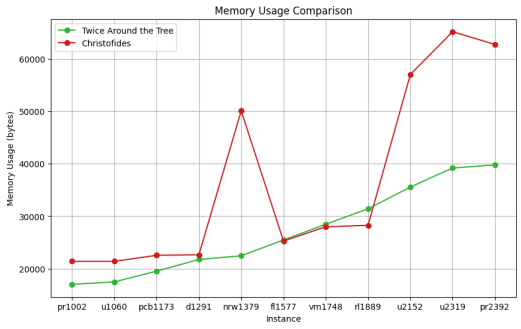

Figure 9. Comparação de Tempo de Execução para Instâncias Grandes

Pela figura 9, percebe-se que para instâncias grandes a quantidade de memória necessária quando se tratando do algoritmo de Christofides é muito dependente das características particulares da instância, podendo ser elevada drásticamente como em *nrw1379*. Uma possibilidade para esse acontecimento é haver uma alta quantidade de emparelhamentos ou nodos de grau ímpar na árvore. Já para Twice Around, o uso de memória é bem mais constante, aumentando conforme a dimensão aumenta. Isso se deve ao fato do mesmo só ter que guardar sua árvore mínima e seu caminho final, não dependendo de características particulares, como quantidade de nodos ímpares.

7.9 Análise de Qualidade para Instâncias Grandes (>1000)

Pela figura 10, pode-se perceber uma diferença ainda maior entre a precisão dos dois algoritmos, com Christofides estando ao redor de 0.15 a 0.20 e Twice Around entre 0.35 e 0.45. Apesar de haverem ligeiros picos que podem depender de casos de borda particulares das instâncias, seu valor

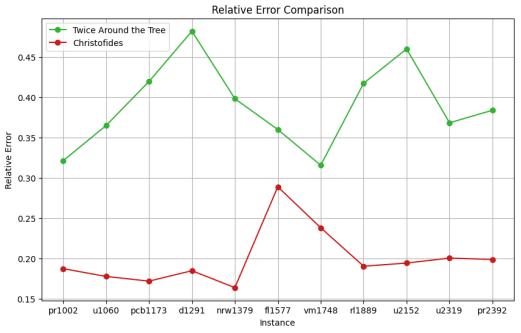

Figure 10. Comparação de Erro Relativo para Instâncias Grandes

é relativamente constante com o aumento da dimensão.

7.10 Comparação geral

Com base em todos os gráficos expostos e a tabela de resultados 1, algumas considerações puderam ser retiradas em relação as métricas de cada algoritmo: Em relação ao tempo de execução, o algoritmo mais eficiente é o Twice Around, seguido pelo Christofides e sendo o de menor eficiência, o Branch and Bound que, por ser exaustivo, só pode ser considerado para instâncias muito pequenas. Para essa métrica, observou-se que as particularidades da instância tem efeito muito grande, muitas vezes superando a importância de sua dimensão, principalmente para instâncias médias. Já se tratando do uso de memória, percebemos um uso muito inferior do Branch and Bound, seguido pelo Twice Around e por fim, com uso extensivo de memória, o Christofides. A dimensão da instância tem efeito principal nessa métrica, mas as particularidades do problema também podem ter um efeito considerável, principalmente em instâncias maiores. Já se tratando de qualidade, percebe-se que o Branch and Bound garante um erro relativo de 0, uma vez que não é aproximativo. Em relação aos outros dois, Christofides garante uma precisão consideravelmente maior que Twice Around a medida que a instância aumenta. Essa métrica é relativamente constante com o aumento da dimensão e sofre feitos médios com base em particularidades.

8. CONCLUSÃO

Com base em todas as análises feitas dos algoritmos Branch and Bound, Twice Around the Tree e Christofides, validou-se tanto com o estudo das implementações, quanto com os resultados e análises descobertos na etapa de experimentos, pontos relevantes de cada um e comparações entre eles. Evidenciou-se a preferência de uso do Branch and Bound para instâncias muito pequenas e problemas nos quais é necessário o caminho ótimo, do uso de Twice Around the Tree para instâncias de todos os tamanhos onde a eficiência e o uso de memória são os pontos principais, em detrimento da importância de precisão e, por fim, do uso de Christofides para instâncias de todos os tamanhos onde a precisão é mais importante que a rapidez e o uso eficiente de memória. Também percebeu-se a influência

de características particulares dos problemas nas métricas, como casos de borda e ordenação dos nodos. Portanto, o estudo foi de importância para evidenciar fatos teoricamente sabidos das implementações e compreender novos, não tão claramente percebidos de início.

9. APÊNDICE

As implementações, tabelas e gráficos deste trabalho estão disponíveis no repositório GitHub: https://github.com/estersassis/traveling-salesman-problem/tree/main.

9.1 Instruções de Execução

Execução que irá utilizar os valores padrão para as pastas e arquivos:

```
$ python3 main.py
```

Exemplo de execução que controla o caminho de pastas e arquivos:

```
$ python3 main.py -p tsp -o
   processed_folder -r results_folder
   -opt optimal_file.txt
```

Pode-se usar a tag abaixo paga garantir que os arquivos voltem para pasta de origem após processamento.

```
$ python3 main.py --move-back
```

Para mais informações sobre as tags, use:

```
$ python3 main.py -h
```

10. REFERENCES

[1] REINELT, Gerhard. Universität Heidelberg. TSPLIB. Disponível em: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. Acesso em: 11 jan. 2025.

[2] VIMIEIRO, Renato. DCC/ICEx/UFMG. Aula 12 – Soluções exatas para problemas difíceis (branch-and-bound). DCC207 – Algoritmos 2. Disponível em: https://virtual.ufmg.br/20242/pluginfile.php/331282/mod_folder/content/0/aula12-branch-and-bound.pdf?forcedownload=1. Acesso em: 11 jan. 2025

[3] VIMIEIRO, Renato. DCC/ICEx/UFMG. Aula 13 – Soluções aproximadas para problemas difíceis. DCC207 – Algoritmos 2. Disponível em: https://virtual.ufmg.br/20242/pluginfile.php/331282/mod_folder/content/0/aula13-alg-aproximativos.pdf?forcedownload=1. Acesso em: 11 jan. 2025

[4] LEVITIN, Anany. 2012. Editora Pearson. The Design and Analysis of Algorithms. 3rd edition.

Table 1. Comparação de Algoritmos por instância com base em métricas de tempo, espaço e qualidade

Instância	Branch and Bound			Twice Around The Tree			Christofides		
	Tempo de Execução (s)	Memória Usada (bytes)	Erro Relativo	Tempo de Execução	Memória Usada	Erro Relativo	Tempo de Execução	Memória Usada	Erro Relativo
square4	0.6188	152	0.0000	0.1504	272	0.0000	0.4182	840	0.0000
example4	7.4727	152	0.0000	5.6393	272	0.0000	6.5795	840	0.0000
pentagon5	3.5704	160	0.0000	1.5395	280	0.0000	1.9388	840	0.0000
hexagon6	9.4287	168	0.0000	0.8583	288	0.0000	1.8978	840	0.0000
random6	36.9849	168	0.0000	3.2964	288	0.1250	5.0689	840	0.0000
eil51	NA	NA	NA	14.8142	1000	0.3709	28.9705	1928	0.2629
berlin52	NA	NA	NA	12.5042	1008	0.3410	18.5811	1960	0.2613
st70	NA	NA	NA	3.1513	1344	0.3156	3.9799	2120	0.1881
pr76	NA	NA	NA	1.2716	1392	0.3437	1.7366	2248	0.1392
eil76	NA	NA	NA	7.2037	1392	0.2937	17.9978	3848	0.3327
rat99	NA	NA	NA	2.8886	1832	0.3980	6.2675	4072	0.1503
kroB100	NA	NA	NA	8.1539	1840	0.1691	18.5245	2440	0.1402
kroC100	NA	NA	NA	3.0811	1840	0.3479	6.1549	4072	0.2319
rd100	NA	NA	NA	0.7212	1840	0.3641	2.4853	4072	0.2412
kroD100	NA	NA	NA	19.8249	1840	0.2732	26.9781	4072	0.1693
kroA100	NA	NA	NA	15.2560	1840	0.2785	27.5859	4072	0.3121
kroE100	NA	NA	NA	8.2173	1840	0.3578	19.6434	4072	0.2243
eil101	NA	NA	NA	14.4749	1848	0.3196	28.0410	4168	0.2051
lin105	NA	NA	NA	18.8975	1880	0.3558	28.9114	4072	0.2335
pr107	NA	NA	NA	19.4653	1896	0.2242	29.8087	4072	0.1060
pr124	NA	NA	NA	0.3898	2192	0.2560	2.9943	2536	0.1508
bier127	NA	NA	NA	0.6914	2216	0.3411	2.7314	4424	0.2504
ch130	NA	NA	NA	16.8960	2400	0.3304	28.6498	4392	0.1872
pr136	NA	NA	NA	18.3909	2448	0.5697	28.0968	2760	0.1376
pr144	NA	NA	NA	22.8040	2512	0.3769	34.3576	2696	0.2612
kroA150	NA	NA	NA	4.2479	2752	0.3240	9.4223	4776	0.3520
kroB150	NA	NA	NA	29.9274	2752	0.3835	55.8425	4776	0.3869
ch150	NA	NA	NA	29.6313	2752	0.2786	49.6479	4584	0.2198
pr152	NA	NA	NA	29.9353	2768	0.1943	41.8978	2888	0.1057
u159	NA	NA	NA	24.2429	2824	0.3734	32.8977	4776	0.1975
rat195	NA	NA	NA	23.5801	3336	0.3922	39.7617	5128	0.1477
d198	NA	NA	NA	1.5452	3360	0.2000	3.1420	5128	0.1417
kroA200	NA	NA	NA	10.2729	3376	0.3630	19.2753	5384	0.1891
kroB200	NA	NA	NA	0.5828	3376	0.3827	8.2323	5384	0.2549
tsp225	NA	NA	NA	1.1397	3832	0.3072	3.7609	5384	0.1925
ts225	NA	NA	NA	1.2079	3832	0.4846	3.6314	5064	0.0590
pr226	NA	NA	NA	26.3486	3840	0.4520	36.1347	5384	0.2034
gil262	NA	NA	NA	1.0668	4416	0.3911	11.5888	5960	0.2140
pr264	NA	NA	NA	27.2000	4432	0.3528	36.3634	5672	0.2428
a280	NA	NA	NA	23.2211	4880	0.3928	34.3569	6280	0.1772
pr299	NA	NA	NA	19.6858	5032	0.3414	35.4874	6280	0.1795
linhp318	NA	NA	NA	1.8305	5536	0.3833	7.2648	6792	0.2530
lin318	NA	NA	NA	1.8813	5536	0.3833	5.9609	6792	0.2530
rd400	NA	NA	NA	3.5222	6576	0.3305	19.8208	14184	0.2268
fl417	NA	NA	NA	1.9323	7160	0.3658	34.0166	7816	0.4909
pr439	NA	NA	NA	4.0681	7336	0.3489	8.8072	7624	0.2738
pcb442	NA	NA	NA	3.1230	7360	0.3632	7.9380	14184	0.2254
d493	NA	NA	NA	5.4058	8280	0.2823	16.6583	14952	0.1692
u574	NA	NA	NA	37.9611	9504	0.3251	47.3874	15816	0.1822
rat575	NA	NA	NA	29.4236	9512	0.3868	52.9401	15816	0.2190
d657	NA	NA	NA	6.1246	10808	0.3563	19.4656	16776	0.1983
u724	NA	NA	NA	26.6545	12048	0.3773	33.7829	17832	0.2145
rat783	NA	NA	NA	38.4905	13320	0.3537	57.4490	18632	0.1716
pr1002	NA	NA	NA	25.3394	16992	0.3211	31.3299	21384	0.1872
u1060	NA	NA	NA	35.5346	17456	0.3648	43.9984	21384	0.1775
vm1084	NA	NA	NA	50.7281	17648	0.3175	58.0029	20936	0.2118
pcb1173	NA	NA	NA	59.9563	19512	0.4195	68.2689	22536	0.1717
d1291	NA	NA	NA	87.3728	21736	0.4819	93.1906	22632	0.1846
rl1323	NA	NA	NA	35.9634	21992	0.4080	40.2842	22312	0.1858
nrw1379	NA	NA	NA	64.5381	22440	0.3981	78.2509	50120	0.1636
u1432	NA	NA	NA	63.1302	24304	0.4017	75.2710	26344	0.1609
fl1577	NA	NA	NA	91.7035	25464	0.3599	99.2735	25256	0.2888
d1655	NA	NA	NA	23.1082	27720	0.3727	69.8194	27976	0.1672
vm1748	NA	NA	NA	69.4517	28464	0.3155	81.8029	27976	0.2379
rl1889	NA	NA	NA	94.8980	31416	0.4173	103.2051	28264	0.1902
u2152	NA	NA	NA	55.4464	35568	0.4599	70.8627	57064	0.1941
u2319	NA	NA	NA	64.0130	39208	0.3682	97.8653	65224	0.2003
pr2392	NA	NA	NA	68.0181	39792	0.3838	92.3765	62792	0.1986