Trabalho Prático 2
Soluções para Problemas Difíceis

Rafaela de Fátima Silva Alexandre[1]

[1]Universidade Federal de Minas Gerais (UFMG)
Caixa Postal 31.270-901 – Minas Gerais – MG – Brazil

[2]Departamento de Ciência da Computação - Universidade Federal de Minas Gerais
Belo Horizonte, Brazil

rafaelaalexandre@ufmg.br

Resumo. *Este meta-artigo descreve os aspectos práticos dos algoritmos para solucionar problemas difíceis. Avaliaremos as implementações dos algoritmos para computação de rotas no problema do caixeiro viajante. Especificamente, uma solução exata, baseada em branch-and-bound, e as duas soluções aproximadas vistas em sala de aula para o TSP euclidiano, twice-around-the-tree e algoritmo de Christofides.*

1. Introdução

Este artigo explora os aspectos práticos dos algoritmos para resolver problemas complexos, com foco no desafio do problema do caixeiro viajante (TSP), um tema clássico na computação e otimização. A motivação deste estudo reside na importância persistente do TSP em diversas aplicações práticas, como logística, planejamento de rotas e otimização de redes. A principal contribuição deste trabalho é a avaliação comparativa de algoritmos para o TSP, incluindo uma solução exata baseada em branch-and-bound e duas soluções aproximadas - twice-around-the-tree e o algoritmo de Christofides.

2. Implementação

2.1. Twice Around the Tree

No algoritmo ”Twice Around the Tree”, a estimativa de custo é baseada na construção de uma Árvore Geradora Mínima (AGM) a partir de um vértice raiz. Esta abordagem é fundamentada na teoria de que a AGM de um grafo fornece uma subestimativa do custo do ciclo hamiltoniano, uma vez que qualquer ciclo hamiltoniano pode ser transformado em uma AGM ao remover uma de suas arestas.

Utilizou-se a biblioteca NetworkX para manipulação de grafos em Python, dada sua flexibilidade e vasta gama de algoritmos incorporados. A AGM é construída utilizando o algoritmo de Prim, por sua eficiência e simplicidade. Para a caminhada pré-ordem na AGM, foi implementada uma função de busca em profundidade (depth-first search), que é adequada para explorar todas as arestas e vértices de uma árvore.

A escolha entre best-first e depth-first no branch-and-bound depende do trade-off entre uso de memória e tempo de execução. Optou-se pela busca em profundidade (depth-first), considerando sua menor exigência de memória e a natureza do problema TSP, onde a exploração completa de um caminho antes de passar para outro é mais intuitiva.

Passo 1 Construa uma árvore geradora mínima do grafo correspondente a uma instância dada do problema do caixeiro viajante.

Passo 2 Começando em um vértice arbitrário, faça uma caminhada ao redor da árvore geradora mínima registrando todos os vértices por onde passa.)

Passo 3 Examine a lista de vértices obtida no Passo 2 e elimine dela todas as ocorrências repetidas do mesmo vértice, exceto a inicial no final da lista. (Este passo é equivalente a fazer atalhos na caminhada.) Os vértices restantes na lista formarão um circuito hamiltoniano, que é a saída do algoritmo.

Exemplo 2

Pode-se aplicar este algoritmo ao grafo na Figura 1. A árvore geradora mínima deste grafo é composta pelas arestas (a, b), (b, c), (b, d) e (d, e). Uma caminhada ao redor da árvore duas vezes, que começa e termina no vértice a, é:

$$a, b, c, b, d, e, d, b, a.$$

Eliminando o segundo b (um atalho de c para d), o segundo d e o terceiro b (um atalho de e para a) resulta no circuito hamiltoniano

$$a, b, c, d, e, a$$

com comprimento 39.

O passeio obtido no Exemplo 2 não é ótimo. Embora essa instância seja pequena o suficiente para encontrar uma solução ótima por busca exaustiva ou branch-and-bound, foi escolhio não fazê-lo para reiterar um ponto geral. Como regra, não é possível saber qual é o comprimento de um passeio ótimo de fato, e, portanto, não é possivel calcular a fração exata $\frac{f(sa)}{f(s^*)}$. Para o algoritmo, pode-se pelo menos estimá-lo acima, desde que o grafo seja euclidiano.

Teorema 2

O algoritmo TAT é um algoritmo de 2-aproximação para o problema do Caixeiro viajante com distâncias euclidianas.

Prova

Obviamente, o algoritmo TAT é feito em tempo polinomial se for possivel usar um algoritmo razoável como o de Prim ou Kruskal no Passo 1. É preciso mostrar que, para qualquer instância euclidiana do problema do caixeiro viajante, o comprimento de um passeio sa obtido pelo TAT é no máximo o dobro do comprimento do passeio ótimo s^*, ou seja,

$$f(sa) \leq 2f(s^*).$$

Uma vez que remover qualquer aresta de s^* produz uma árvore de abrangência T de peso $w(T)$, que deve ser maior ou igual ao peso da árvore geradora mínima do grafo $w(T^*)$, obtem-se a desigualdade

$$f(s^*) > w(T) \geq w(T^*).$$

Esta desigualdade implica que

$2f(s^*) > 2w(T^*) =$ o comprimento da caminhada obtida no Passo 2 do algoritmo.

Os possíveis atalhos delineados no Passo 3 do algoritmo para obter sa não podem aumentar o comprimento total da caminhada em um grafo euclidiano, ou seja,

o comprimento da caminhada obtida no Passo 2 $\geq$ o comprimento do passeio sa.

Combinando as duas últimas desigualdades

$$2f(s^*) > f(sa),$$

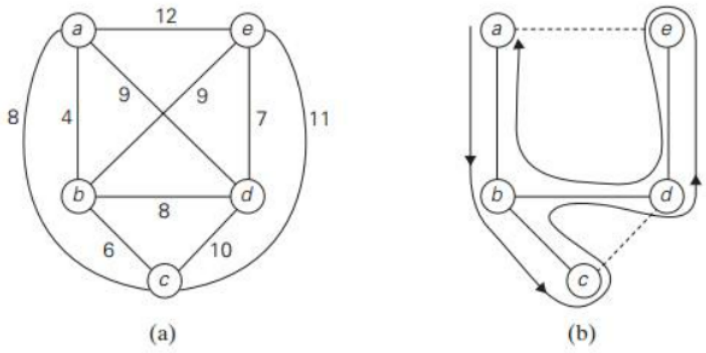

Figure 1. Ilustração do algoritmo de duas voltas ao redor da árvore. (a) Grafo. (b) Caminhada ao redor da árvore geradora mínima com os atalhos.6

2.2. Branch and Bound

A estimativa de custo é calculada inicialmente como a soma das arestas mínimas de entrada e saída para cada nó, dividida por dois, de acordo com a heurística do branch-and-bound. Isso estabelece o limite inferior (bound) do custo do caminho sendo explorado. Durante a exploração, se um caminho promissor é encontrado, o limite é ajustado para refletir o novo custo mínimo encontrado.

Um array é usado para manter o registro do caminho atual sendo explorado e, outro, utilizado para marcar os nós já visitados no caminho atual para evitar ciclos. Uma pilha armazena os estados do caminho durante a busca para permitir backtracking. Os custos dos caminhos são armazenados e atualizados conforme novos caminhos são explorados.

O algoritmo utiliza uma abordagem de profundidade para explorar os caminhos. Isso é evidenciado pelo uso de uma pilha para armazenar os estados do caminho e pelo loop que explora os nós de forma recorrente a partir de um ponto inicial. A abordagem de profundidade é adequada para este problema, pois permite uma exploração sistemática do espaço de busca, utilizando a heurística do branch-and-bound para cortar os ramos não promissores.

Um limite de tempo é imposto para a execução do algoritmo, gerando uma exceção *ExecutionTimeout* se o tempo exceder o limite pré-definido. Existem funções que são usadas para calcular as arestas de menor custo conectando os nós, o que é crucial para a heurística de branch-and-bound. O algoritmo busca continuamente melhorar o custo ótimo e atualizar o caminho ótimo à medida que novos caminhos são explorados.

2.3. Algoritmo de Christofides

O algoritmo de Christofides também utiliza uma árvore geradora mínima, mas faz isso de uma maneira mais sofisticada que o algoritmo Twice Around the Tree[2]. Note que uma caminhada ao redor da árvore gerada pelo último algoritmo é um circuito euleriano no multigrafo obtido ao duplicar todas as arestas no grafo dado. Dada a definição de que um circuito euleriano existe em um multigrafo conectado se, e somente se, todos os seus vértices tiverem graus pares.

O algoritmo de Christofides obtém tal multigrafo adicionando ao grafo as arestas de um emparelhamento de peso mínimo de todos os vértices de grau ímpar em sua árvore geradora mínima. (O número de tais vértices é sempre par e, portanto, isso sempre pode ser feito.) Em seguida, o algoritmo encontra um circuito euleriano no multigrafo e o transforma em um circuito hamiltoniano por atalhos, exatamente da mesma forma que é feito no último passo do algoritmo *Twice Around the Tree*.

O algoritmo de Christofides, é uma heurística que busca solucionar o problema do caixeiro-viajante dentro de uma margem de 1,5 vezes o valor da solução ótima para grafos que atendem à desigualdade triangular. As etapas implementadas para a aplicação do algoritmo são delineadas a seguir.

A árvore geradora mínima (AGM) é obtida por meio da biblioteca NetworkX, tratando-se do esqueleto inicial para a construção da solução. A AGM garante a interconexão de todas as localidades com o mínimo custo agregado, evitando a formação de ciclos.

Os vértices que possuem grau ímpar na AGM são identificados subsequentemente. Esta identificação é crucial para as etapas seguintes, pois a formação de um circuito euleriano é condicionada pela paridade no grau de todos os vértices.

Para os vértices de grau ímpar, procede-se à criação de um emparelhamento que minimize o peso das arestas interconectando-os. A finalidade desta etapa é ajustar a AGM para que se alcance a paridade de graus necessária para a existência de um circuito euleriano.

Emprega-se um procedimento algorítmico para determinar o circuito euleriano no multigrafo. Tal circuito percorre todas as arestas uma única vez, constituindo o trajeto a ser percorrido pelo caixeiro-viajante.

Finalmente, o circuito euleriano é transformado em um trajeto viável para o caixeiro-viajante. Isso é feito pela eliminação de visitações repetidas a cidades, culminando no fechamento do circuito no ponto de origem. O percurso resultante atende ao problema proposto, assegurando-se de que seu custo não exceda 1,5 vezes o custo do percurso ótimo.

3. Experimentos

Neste trabalho, foram usados diversos datasets na área de problemas de otimização combinatoria, especificamente para o problema do caixeiro viajante (TSP). A tabela a seguir resume os datasets utilizados, incluindo o número de nós (cidades) e o valor do limiar (distância ou custo) para cada um:

Table 1. Alguns Datasets Utilizados no Experimento

Dataset	Nós	Limiar
eil51	51	426
berlin52	52	7542
st70	70	675
eil76	76	538
pr76	76	108159
rat99	99	1211
kroA100	100	21282
...	...	...

3.1. Metodologia

Neste estudo, empregamos uma metodologia para avaliar o desempenho de vários algoritmos no contexto do problema do caixeiro viajante (TSP). A abordagem foi estruturada em duas fases principais: a execução dos algoritmos e a análise comparativa de seu desempenho.

Os algoritmos selecionados, incluindo o Algoritmo de Christofides, Branch and Bound, e Twice Around the Tree, foram implementados e testados em uma variedade de datasets. Estes datasets variam em tamanho e complexidade, proporcionando uma avaliação abrangente da eficácia e eficiência de cada algoritmo.

3.2. Medidas de Desempenho

Tempo de Execução: O tempo de execução foi cronometrado para cada algoritmo em cada dataset. Esta medida fornece uma avaliação direta da eficiência computacional dos algoritmos.

Qualidade da Solução: Para o problema do TSP, a qualidade da solução foi avaliada com base na distância total percorrida. Isso permite uma comparação direta entre as

soluções obtidas pelos diferentes algoritmos e a solução ótima conhecida ou as melhores soluções conhecidas para cada dataset.

Além de medir o tempo de execução e a qualidade das soluções, conduziu-se uma análise comparativa para avaliar a eficácia dos algoritmos. Esta análise envolveu comparar as soluções obtidas com as soluções ótimas ou as melhores soluções conhecidas, permitindo avaliar a capacidade de cada algoritmo em aproximar a solução ótima.

A eficiência computacional dos algoritmos foi avaliada não apenas em termos de tempo de execução, mas também considerando a complexidade computacional teórica. Discutindo como diferentes características dos datasets, como o número de nós e a distribuição das distâncias, afetam o desempenho dos algoritmos.

4. Experimentos: Algoritmo de Christofides

O objetivo é comparar a distância (limiar) da solução ótima com a distância obtida pelo algoritmo de Christofides. Além disso, calcular a diferença percentual entre as distâncias para avaliar o quão longe está da solução ótima.

Primeiro, calcular a diferença percentual para cada dataset. A diferença percentual é dada por:

$$\text{Diferença Percentual} = \left(\frac{\text{Distância do Algoritmo} - \text{Distância Ótima}}{\text{Distância Ótima}} \right) \times 100\%$$

Para datasets considerados pequenos, o algoritmo de Christofides teve o seguinte desempenho:

Table 2. Comparação entre Solução Exata e Algoritmo de Christofides

Dataset	Nós	Limiar Exato	Solução	Tempo (s)	(MB)	Desvio (%)
eil51	51	426	978	0.7195	1.5195	129.58%
berlin52	52	7542	16306	0.7072	0.1445	116.20%
st70	70	675	1943	0.7268	0.2070	187.85%

Ao análisar instâncias maiores, pode-se notar a tendência crescente do tempo de execução e a quantidade de memória usada. Há, também, instâncias que passaram do tempo limite (30 minutos), e não obtiveram a solução.

Table 3. Comparação entre Solução Ótima e Solução do Algoritmo de Christofides

Dataset	Nós	Limiar Exato	Solução	Tempo (s)	(MB)	Desvio (%)
d2103	2103	[79952,80450]	119865	18.138	92.109375	149.00(%)
u2152	2152	64253	298.893	1029.46	1183.91	171.00(%)
u2319	2319	234256	NaN	NaN	1357.6	NA
pr2392	2392	378032	NaN	NaN	1224.4	NA

A comparação completa dos 78 datasets encontra-se no apêndiceA.2

O fator de aproximação de 3/2 (1.5 vezes), gera um maior custo. Computando a MST do grafo, e, a partir disso, computar o *perfect matching*

Os resultados para d2103 estão dentro do limite teórico de 1,5 vezes a solução ótima, indicando um bom desempenho do algoritmo. No entanto, para u2152, a solução excedeu este limite, sugerindo que o algoritmo pode não ter sido tão eficaz neste caso ou que os dados do problema podem não cumprir totalmente as premissas do algoritmo, como a desigualdade triangular.

Os tempos de execução são relativamente rápidos, Porém, ao considerar o exemplo com o dataset *u2152*, podemos observar um grande salto em relação ao dataset anterior, chegando, quase, ao limite especificado (30 minutos). Isso deve-se, especialmente, ao número de nós e a complexidade do algoritmo. Isso está alinhado com a complexidade computacional do algoritmo, que é dominada pela etapa de emparelhamento perfeito, com uma complexidade de $O(n^3)$.

O uso da memória também teve um salto nos diferentes datsets. Por exemplo, para d2103, o uso de memória é 92.1 MB. Para o dataset *u2152* o algoritmo requer mais memória para armazenar a árvore geradora mínima, o emparelhamento e o multigrafo.

O algoritmo, apesar de funcionar rápidamente para instâncias menores, tem sua limitação encontrada a partir dos datasets que possuem nós de tamanhos mais significativos. Como a diferença entre rodar um dataset que possui 2103 nós, para 2319, o algoritmo já não consegue rodar em um tempo menor que o estabelecido pela especificação.

Para instâncias menores, o algoritmo de Christofides pode ser mais adequado, dada a sua garantia de aproximação e complexidade computacional. No entanto, para instâncias maiores, alternativas como algoritmos exatos ou outras heurísticas podem ser mais eficazes.

4.1. Twice Around the Tree

Table 4. Comparação entre Solução Ótima e Solução do Algoritmo "Twice Around the Tree"

Dataset	Nós	Limiar Exato	Solução	Tempo (s)	Memória (MB)	Desvio (%)
eil51	51	426	597	0.208	0.141	40.14
berlin52	52	7542	9908	0.205	0.004	31.37
st70	70	675	839	0.207	0.012	24.30
eil76	76	538	685	0.206	0.016	27.32

A comparação completa dos 78 datasets encontra-se no apêndiceA.4

O algoritmo demonstrou eficiência temporal nas instâncias menores, com tempos de execução variando entre 0.205 e 0.208 segundos.

Considerando os baixos tempos de execução para todas as instâncias, o algoritmo parece escalar bem com o aumento do número de nós. No entanto, seria interessante testar

o algoritmo em instâncias ainda maiores para avaliar sua escalabilidade em cenários mais desafiadores.

O algoritmo mostrou um uso de memória muito baixo para os primeiros testes, variando de 0.004 MB a 0.141 MB, o que indica uma boa eficiência de memória. Mas, em comparação com os datsets maiores:

Table 5. Comparação entre Solução Ótima e Solução do Algoritmo Twice Around the Tree - Datasets Maiores

Dataset	Nós	Limiar Exato	Solução	Tempo (s)	(MB)	Desvio (%)
pr2392	2392	378032	524996	6.01	381.64	38.87
pcb3038	3038	137694	196565	13.07	384.45	42.75
fl3795	3795	28747.5	40385	31.63	1200.52	40.43
fnl4461	4461	182566	254634	58.04	1959.25	39.50

A gestão eficiente da memória é um aspecto positivo, especialmente considerando a natureza complexa dos problemas do TSP, que muitas vezes requerem o armazenamento e a manipulação de uma grande quantidade de dados.

A análise do desvio percentual em relação às soluções ótimas mostra que o algoritmo não atinge a solução ótima, com desvios variando de 24.30% a 40.14%. Embora o algoritmo não ofereça soluções ótimas, fornece resultados razoavelmente próximos em um tempo muito curto.

Para instâncias menores (como as testadas), o algoritmo parece ser uma boa escolha quando a velocidade é mais crítica do que a obtenção da solução ótima. Como pode-se observar na tabela acima, mesmo com o crescimento consideravel do tamanho dos nós, o algoritmo se mostrou consistente, e manteve a diferença entre a solução ótima, dentro do mesmo valor notado pelos testes feitos em datasets menores. O tempo não obteve aumento significativo, apesar do uso da memoria ter crescido.

4.2. Branch and Bound

Para conjuntos de dados pequenos, um algoritmo de branch-and-bound pode funcionar de forma eficiente, mas à medida que o número de cidades aumenta, o número de caminhos potenciais (ramificações) a explorar também cresce exponencialmente, tornando o processo computacionalmente intensivo.

Mesmo com uma boa função de limitação, os algoritmos de branch-and-bound podem sofrer de limitações práticas, como restrições de memória e limites de tempo de execução. O algoritmo utiliza uma pilha para armazenar os estados (nós) a serem explorados, indicando uma abordagem de busca em profundidade. Em cada etapa, o algoritmo verifica se o limite inferior atual, somado ao custo do caminho, é menor que o custo ótimo encontrado até então. Se não for, o ramo é podado.

É difícil prever quais instâncias do TSP serão solucionáveis em um tempo realista. Essa imprevisibilidade é uma característica dos problemas NP-difíceis e é uma das razões pelas quais algoritmos heurísticos e de aproximação são frequentemente usados para instâncias maiores do TSP.

Apesar disso, o Branch and Bound é o único algoritmo que encontraria uma solução exata, por isso, acaba sendo o mais caro de todos. Mesmo para a instância de 51 nós, não foi possível obter sucesso ao tentar encontrar uma solução. Isso acontece devido ao fato de que o algoritmo Branch and Bound constroi árvores binárias de subproblemas, com um crescimento exponencial, explicando, assim, a limitação computacional.

Table 6. Comparação entre Solução Ótima e Solução do Algoritmo "Twice Around the Tree"

Dataset	Nós	Limiar Exato	Solução	Tempo (s)	Memória (MB)	Desvio (%)
eil51	51	426	NaN	NaN	NaN	NaN
berlin52	52	7542	NaN	NaN	NaN	NaN
st70	70	675	NaN	NaN	NaN	NaN
eil76	76	538	NaN	NaN	NaN	NaN

A comparação completa dos 78 datasets encontra-se no apêndiceA.3

4.3. Comparação entre os algoritmos

Ao comparar os algoritmos de Christofides, Twice Around the Tree (TAT), e Branch and Bound para resolver o problema do caixeiro viajante (TSP), foi possível observar diferenças significativas em termos de eficiência, precisão e viabilidade para diferentes tamanhos de instâncias.

- Algoritmo de Christofides

Eficiência: O algoritmo de Christofides é mais eficiente em instâncias menores, mas seu tempo de execução e uso de memória aumentam significativamente para instâncias maiores.

Precisão: Apresenta um desvio considerável da solução ótima (por exemplo, 129.58(% para eil51). Esse valor está dentro do valor aproximativo do algoritmo (1.5x)

Uso: É mais adequado para instâncias menores ou quando uma solução aproximada é aceitável, especialmente devido à sua garantia teórica de não exceder 1.5 vezes a solução ótima.

- Twice Around the Tree (TAT)

Eficiência: Mostra-se extremamente eficiente em termos de tempo e uso de memória, mesmo para instâncias maiores.

Precisão: Fornece soluções com desvios menores em comparação com o algoritmo de Christofides (por exemplo, 40.14% para eil51), indicando uma aproximação mais próxima da solução ótima.

Uso: Ideal para situações onde a eficiência é crucial e uma aproximação razoavelmente precisa é suficiente.

- Branch and Bound

Eficiência: Tem uma eficiência decrescente com o aumento do número de nós, tornando-se impraticável para instâncias maiores devido ao crescimento exponencial do espaço de busca.

Precisão: É o único que pode garantir a solução exata, mas em muitos casos, não consegue alcançá-la dentro de um limite de tempo razoável.

Uso: Recomendado apenas para instâncias muito pequenas, onde é crucial obter a solução exata.

5. Conclusão

As seguintes conclusões podem ser destacadas:

Trade-off entre Precisão e Eficiência:

A escolha do algoritmo apropriado para o TSP depende fortemente de um equilíbrio entre a precisão da solução e a eficiência computacional (tempo e memória). Enquanto o Branch and Bound oferece a solução exata, sua aplicabilidade é limitada a instâncias muito pequenas devido à sua natureza exponencial. Algoritmos heurísticos como Christofides e TAT, apesar de não garantirem a solução ótima, são muito mais viáveis para instâncias de tamanho médio a grande.

Desempenho em Diferentes Tamanhos de Instâncias:

Para instâncias pequenas, o Branch and Bound pode ser considerado quando a obtenção da solução exata é crucial. Para instâncias médias e grandes, os algoritmos heurísticos, especialmente o TAT, demonstram um desempenho superior, oferecendo soluções aproximadas com eficiência de tempo e memória notavelmente melhores.

Impacto do Tamanho da Instância: Observou-se que, à medida que o número de nós aumenta, os algoritmos heurísticos se tornam significativamente mais eficientes em comparação com o Branch and Bound. A capacidade de escalar com o tamanho da instância é um fator crítico na escolha do algoritmo para instâncias maiores.

Precisão das Soluções: Os experimentos revelaram que, embora o TAT e o algoritmo de Christofides não alcancem a solução ótima, eles fornecem aproximações razoáveis em um tempo muito mais curto. O desvio percentual da solução ótima foi consideravelmente menor para o TAT em comparação com o algoritmo de Christofides.

Uso de Memória e Tempo de Execução:

O TAT mostrou ser excepcionalmente eficiente em termos de uso de memória e tempo de execução, mesmo para instâncias maiores, tornando-o uma escolha robusta para uma ampla gama de tamanhos de instância. Apesar de encontrar limitações, também, para datasets com nós maiores, como o r1515 (nós: 5912), que não obteve solução dentro do tempo especificado

O Branch and Bound, embora preciso, mostrou limitações significativas em termos de uso de memória e tempo de execução, até mesmo para os primeiros datasets, e, especialmente, para instâncias maiores.

Aplicabilidade Prática:

Em cenários práticos, onde o tempo e os recursos computacionais são limitados, os

algoritmos heurísticos como TAT e Christofides são preferíveis. A escolha do algoritmo deve considerar a natureza específica do problema, como a importância da precisão em relação à eficiência.

6. References

[tsp b] [bra] [tsp a] [tsp c]

References

[tsp a] Approximate solution for travelling salesman problem using mst. https://www.geeksforgeeks.org/approximate-solution-for-travelling-salesman-problem-using-mst/. Accessed: [your access date].

[tsp b] Approximation algorithms for the traveling salesman problem. https://www.brainkart.com/article/Approximation-Algorithms-for-the-Traveling-Salesman-Problem_8065/. Accessed: 24-Jan-2020.

[bra] Branch and bound algorithm. https://www.geeksforgeeks.org/branch-and-bound-algorithm/. Accessed: [your access date].

[tsp c] Tsplib95. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. Accessed: [your access date].

A. Apêndice

A.1. Tabela de Datasets do Problema do Caixeiro Viajante Geométrico

Soluções Ótimas

A.2. Soluções obtidas - Algoritmo de Christofides

Soluções

A.3. Soluções obtidas - Branch and Bound

Soluções

A.4. Soluções obtidas - Twice Around the Tree

Soluções