Aproximações para TSP fazendo uso das heurísticas: Bellmore & Nemhauser, Twice-Around, Christofides, Insere Vértice 1 e Insere Vértice 2

Everton Santos de Andrade Junior[1]

[1]Departamento de Computação (DCOMP) – Universidade Federal de Sergipe (UFS)
Av. Marechal Rondon, s/n – Jardim Rosa Elze – CEP 49100-000
São Cristóvão – SE – Brazil

everton.junior@dcomp.ufs.br

Resumo. *O TSP (Travelling salesman problem) consiste em encontrar um ciclo hamiltoniano de custo mínimo em um grafo ponderado e não direcionado, esse problema é NP-Hard, ou seja, é desconhecido um algoritmo de complexidade de tempo polinomial para encontrar tal ciclo. Porém, existem heurísticas que geram uma aproximação, ou seja, um ciclo de baixo custo. Com isso, este artigo tem como objetivo trazer aplicações de algumas dessas heurísticas para encontrar esses ciclos de baixo custo em grafos completos, disponibilizado em banco de dados usado para o TSP. Além disso, esse artigo explica a ideia conceitual e compara essas heurísticas, levando em consideração as aproximações e as soluções exatas para cada um desses grafos. Ao final, através de uma implementação na linguagem de programação Python, conseguimos trazer dados para cada heurística e grafo com finalidade de esclarecer como esses algoritmos comparam entre si, considerando complexidade computacional, tempo de execução e custo do ciclo encontrado.*

Esta página foi intencionalmente deixada em branco.

1. Introdução

Neste artigo, O TSP (*Travelling salesman problem*) consiste em encontrar um ciclo hamiltoniano de custo mínimo H, que é representada nessa implementação como uma sequência de vértices de um grafo *completo* $G = (V, E, w)$ onde $w : E \to \mathbb{R}$ é a função peso. Tendo em vista que esse problema é *NP-Hard*, foi utilizado cinco heurísticas para encontrar aproximações: Bellmore & Nemhauser, Twice-Around, Christofides, Insere Vértice 1 e Insere Vértice 2 .

Definimos H^* como o ciclo exato de custo minimo e H como a aproximação encontrada pelos algoritmos e ainda usando H_i ou $H[i]$ tal que $i = 1, 2, ..., |H|$ indica o i-ésimo item da sequencia. Também, é usado a notação $w(H) = \sum_{i=1}^{|H|-1} w(H_i, H_{i+1})$

Dessas heurísticas usados, cada uma se encaixa em três categorias, duas usam uma MST (arvore geradora de custo mínimo), uma consiste em fazer escolhas gananciosas/gulosas, e as outras duas restantes utilizam a estratégia de expansão de um ciclo inicial qualquer, inserindo vértices até gerar uma ciclo hamiltoniano de baixo custo aproximado.

A linguagem de programação *Python* foi usada na implementação e, ainda, o embasamento teórico de cada uma dessas heurísticas foi feito através do livro [Goldbarg and Goldbarg 2012].

Os grafos utilizados foram retirados da base de dados encontrada em (https://people.sc.fsu.edu/ jburkardt/datasets/tsp/tsp.html), todos eles são grafos completos, fazendo o TSP para esse artigo ser uma versão simplificado do problema. Os nomes de cada grafo são ATT48, DANTZIG42, FIVE, FRI26, GR17, P01 e mais detalhes sobre a quantidade de vértices, o custo do H^* de cada um desses grafos foram disponibilizados pela base e serão deliberados nas seções a seguir.

Portanto, este artigo objetiva familiarizar o leitor com algum desses algoritmos, em especial, as 5 (cinco) heurísticas citadas. Além de expor a ideia de cada um desses procedimentos, seu tempo de execução e complexidade computacional.

Nos próximos tópicos são listados as heurísticas, os quais incluem a fundamentação geral sobre o algoritmo, seguido de um pseudo código e tabelas dos resultados encontrados.

2. Bellmore & Nemhauser

A heurística de Bellmore & Nemhauser faz uso da estratégia gulosa, e sua implementação é considerada uma das mais simples. A ideia é começar com uma lista H que, inicialmente, possui um vértice $u \in V$ qualquer, encontramos outro vértice $k \in V \wedge k \notin H$ que seja vizinho do último vértice adicionado em H (definimos $L[-1]$ como ultimo item de qualquer lista/sequencia L) e ainda que a aresta $(H[-1], k)$ possua o menor custo possível, essa é a escolha gulosa.

2.1. Pseudocode - Bellmore & Nemhause

```
// Heuristica de Bellmore & Nemhauser
algoritmo Bellmore-Nemhauser( Entrada: G = (V,E, weight) ) → lista:
    // Começamos por qualquer vertice u
    u ∈ V
    //Adicionamos o o ciclo hamiltoniano H
    H = [u]
    // enquanto não visitamos todos vertices
    enquanto |H| < |V| {
        min_weight = ∞
        min_vertice = -1
        // Escolhendo o vertice v cujo aresta (v,u) tem peso minimo
        para v ∈ G.neighbours(u) {
            // pulamos essa estapa caso já existe em H
            se v ∈ H {
                pule
            }
            w = weight((u,v))
            se w < min_weight {
                min_weight = w
                min_vertice  = v
            }
        }
        // adicionamos esse tal v
        H.append(min_vertice)
        // Trocamos u de lugar com esse tal v encontrado e repetimos o processo
        u = min_vertice
    }
    retorne H
```

Figura 1. **Pseudo código da heurística gulosa de Bellmore & Nemhause**

2.2. Análise e Resultados - Bellmore & Nemhause

Visto que a heurística de Bellmore & Nemhauser checa todos os vizinho do último vértice adicionado a H, e ao total H termina de ser preenchido quando $|H| = |V|$, então a complexidade de tempo deste algoritmo é $O(|V|^2)$.

Segue abaixo a tabela de análise indicando o tempo de execução (foi usado a mesma máquina para todos os testes deste artigo), o custo do ciclo hamiltoniano aproximado H encontrado, denotado por $w(H)$,o custo do ciclo exato H^* encontrado, denotado por $w(H^*)$, e a razão entre esses dois valores (se a razão é igual a 1.0 então o H é ótimo, isto é $w(H) = w(H^*)$), abaixo também indicamos o número de vértices de cada grafo.

Bellmore & Nemhause					
Graphs	**ATT48**	**FIVE**	**DANTZIG42**	**FRI26**	**P01**
Execution time (s)	0.0001255	0.0000270	0.0000994	0.0000679	0.0000396
Approximation $w(H)$	40551	21	956	1112	291
Exact $w(H^*)$	33523	19	699	937	291
Ratio $w(H)/w(H^*)$	1.2096	1.1053	1.3677	1.1868	1.0000
Nº of vertices $\|V\|$	48	5	42	26	15

3. Twice Around

A heurística Twice Around faz uso de uma árvore geradora mínima (MST) $T = (V_T, E_T)$ do grafo $G = (V, E)$. Depois, são duplicadas todas as arestas de T, isto é, $E_T \leftarrow E_T \cup E_T$, isso implica que o $deg(v)\ \forall v \in V_T$ foi duplicado, e portanto $deg(v) = 2k$ para algum $k \in \mathbb{N}$, ou seja, $deg(v)$ é par $\forall v \in V_T$. Pelo teorema de Euler T é garantido possuir um ciclo euleriano C, pois o grau de todos os seus vértices é par. É assumido, também, que G é completo, e portanto, podemos retirar vértices repetidos em C e, ainda sim, encontrar um ciclo hamiltoniano H, e é justamente isso que essa heurística faz.

Para entender o porquê esse algoritmo traz um boa aproximação, precisa ser imposto sobre G que $w(a, b) = w(b, a) \geq 0$, ou seja o custo de sair de a para b é o mesmo que b para a, além da *desigualdade triangular* que diz $w(a, c) + w(c, b) \geq w(a, b)$, isto é, 'passar' por um vértice intermediário c é igual ou menos eficiente que sair de a para b diretamente.

Com isso em mente, sabendo que o ciclo euleriano C pode ter no máximo $2|V_T|$ vértices, que pode acontecer ao usar os arcos duplicados que encontramos na MST para entrar e sair de um dado vértice. Pela desigualdade triangular, remover vértices repetidos de C para gerar H não pode aumentar o custo do caminho, apenas diminuir, pois estamos eliminado um vértice intermediário, portanto $w(C) \leq 2 \cdot w(T)$.

Visto que, por definição uma MST possui o menor custo possível ao conectar todos os vértices de G, então qualquer outro subgrafo que conecte todos os vértices de G terá custo maior ou igual a MST, isso implica que $w(T) \leq w(H^*)$ e que , então pela desigualdade triangular o caminho aproximado H encontrado por Twice Around têm custo $w(H) \leq w(C) \leq 2 \cdot w(H^*)$, e portanto o custo de do ciclo hamiltoniano aproximado não pode ser pior que 2 vezes o custo do ciclo hamiltoniano de menor custo (H^*).

3.1. Pseudocode - Twice Around

```
// Heuristica Twice around
algoritmo Twice-Around ( Entrada: G = (V,E, weight) ) → lista:
    // Ciclo hamiltoniano H a ser encontrados
    H = []
    // Encontramos M.S.T. T = (V_T, E_T) através de prims ou kruskal
    T = MST(G)                  // O(|V|^2)

    // Duplicamos todos os arcos em T
    para cada arco u, v ∈ E_T {   //O(|E|)
        w = weight((u,v))
        T.add_edge((v,u,w)) // duplica arcos
    }

    // Encontramos um cliclo euleriano
    L = hierholzer(T)           // O(|E|)

    // TW a parte de remoção de vertices duplos
    enquanto L ≠ ∅ {            // O(|E|)
        l = L.pop()
        se l ∉ H {
            H.append(l)
        }
    }
    // retornamos para o começo, formando um ciclo
    retorne H
```

Figura 2. Pseudocódigo da heurística Twice Around que faz uso de MST

3.2. Análise e Resultados - Twice Around

A heurística Twice Around têm seu custo de complexidade de tempo totalmente dominada pela criação da MST que custa $O(|V|^2)$ usando o algoritmo de Prim. A criação de um ciclo euleriano é $O(|E|)$ e temos um loop sobre todo o ciclo euleriano encontrando que é no máximo $O(2 \cdot |V|)$, portanto $T_{twicearound} \leq O(|V|^2)$.

Da mesma maneira, abaixo a tabela de análise indicando o tempo de execução, o custo do ciclo hamiltoniano aproximado H encontrado, o custo do ciclo exato H^*, e o restante das informações já ditas anteriormente.

Twice Around					
Graphs	**ATT48**	**FIVE**	**DANTZIG42**	**FRI26**	**P01**
Execution time (s)	0.0001263	0.0000179	0.0001160	0.0000713	0.0000393
Approximation $w(H)$	42548	21	928	1259	399
Exact $w(H^*)$	33523	19	699	937	291
Ratio $w(H)/w(H^*)$	1.2692	1.1053	1.3276	1.3436	1.3711
Nº of vertices $\|V\|$	48	5	42	26	15

4. Christofides

A heurística de Christofides, assim como Twice Around, faz uso de uma árvore geradora miníma (MST) $T = (V_T, E_T)$ do grafo G. Mas, no lugar de duplicar as arestas de T, criamos um grafo completo $K_{odd} = (V_{odd}, E_{odd})$ tal que $V_{odd}, = \{v | v \in V_T \wedge deg(v)$ é impar$\}$, note também que pelo teorema do handshaking $|V_{odd}|$ é par. Em seguinda, se encontra um conjunto de arestas $\in E_{odd}$ que forme *matching perfeito* de custo minimo M (pelo fato de $|V_{odd}|$ ser par, e K_{odd} ser completo, é garantido que existe um matching perfeito), no caso da nossa implementação foi utilizado bibliotecas de terceiros (http://jorisvr.nl/article/maximum-matching) para encontrar M, a qual adapta o algoritmo Blossom de Edmonds que custa $O(|E||V|^2)$.

A partir de M, é construído um multigrafo ponderado $G^* = (V, E_T \cup M)$, como M são arestas que conectam os vértices de grau ímpar em E_T e ainda M é perfeito, isto é, inclui todos os vértices $v \in V_T$ tal que v tem grau ímpar, o que implica que todo v recebem +1 aresta e portanto +1 grau, fazendo todos terem grau par nesse multigrafo, então o teorema de Euler permite encontrar um ciclo euleriano em G^*,daqui em diante tudo se mantém como no Twice Around, encontramos esse ciclo euleriano e removemos os vértices repetidos para encontrar a aproximação H para TSP em G.

4.1. Pseudocode - Christofides

```
// Heuristica Christofides
algoritmo Christofides( Entrada: G = (V,E, weight) ) → lista:
    // Ciclo hamiltoniano H a ser encontrados
    H = []

    // Encontramos M.S.T. T = (V_T, E_T) através de prims ou kruskal
    T = MST(G)                  // O(|V|^2)

    //  Grafo que terá todos os vertices impáres de T
    Seja odd = (V_odd, E_odd) um grafo não direcionado e ponderado

    para cada t ∈ V_T {
        // checagem de vertices de grau impar
        se deg(t) for ímpar {
            odd.add_vertice(t)
        }
    }
    // Transformamos 'odd' em um grafo completo
    para cada v ∈ V_odd {
        para cada u ∈ V_odd{
            se u ≠ v {
                w = weight((u,v))
                odd.add_edge((u,v,w))
            }
        }
    }


    //  Através do algoritimo de Edmonds utilizando o  conceito de blossom
    // encontramos um Matching M de custo minimo | M é um conjunto de
    // arestas
    M = blossom(odd)

    // A quantidade de edges nesse matching desse ser igual a exatamente
    // metade da quantidade de vertices de odd
    // pois odd é completo e peso teorema de hand shaking em T sabemos
    // que odd deve ter nº de vertices even

    // Fazemos E_T ∪ M
    para cada e ∈ M {
        T.add_edge(e)
    }

    // Encontramos um cliclo euleriano
    L = hierholzer(T)          // O(|E|)

    // TW a parte de remoção de vertices duplos
    enquanto L ≠ ∅ {                // O(|E|)
        l = L.pop()
        se l ∉ H {
            H.append(l)
        }
    }
    // retornamos para o começo, formando um ciclo
    retorne H
```

Figura 3. **Pseudo código da heurística de Christofides que faz uso de MST e *perfect matching* de custo mínimo**

4.2. Análise e Resultados - Christofides

A heurística Christofides tem seu custo de complexidade idêntico ao Twice Around, exceto pela parte que se encontra o matching perfeito de custo mínimo, como foi comentado custa $O(|E||V|^2)$ que é bem mais custoso que encontrar MST ou qualquer outra parte dessa heurística e portando $T_{christofides}(E, V) \leq O(|E||V|^2)$.

Da mesma maneira, segue abaixo a tabela de informações sobre essa heurística abaixo.

Christofides					
Graphs	**ATT48**	**FIVE**	**DANTZIG42**	**FRI26**	**P01**
Execution time (s)	0.0001192	0.0000201	0.0001370	0.0000854	0.0000608
Approximation $w(H)$	49862	19	1039	1445	406
Exact $w(H^*)$	33523	19	699	937	291
Ratio $w(H)/w(H^*)$	1.2692	1.0000	1.4864	1.5422	1.3952
Nº of vertices $\|V\|$	48	5	42	26	15

5. Insere Vértice 1

A heurística de Insere Vértice 1 entra no campo de heurísticas de expansão, onde se começa por um ciclo qualquer H de $G = (V, E)$ e vamos adicionando um vértice entre esse ciclo até que ele se transforme em um ciclo hamiltoniano. O importante para esses tipos de heurísticas é o critério de inserção, no caso do Insere Vértice 1 primeiro é feito uma busca por algum vértice $k \in V$ que possa estender H, ou seja a aresta (k, H_j) deve ter o custo mínimo para um j qualquer, com a restrição que $k \notin H$, em outras palavras, k é o vértice "mais próximo" de algum vértice em H.

A partir de k, é checado todos os os vértices H_i e H_{i+1}, de maneira encontre qual $i = 1, 2, ..., |H| - 1$ faz custo de atravessar de H_i até H_{i+1} usando k como vértice intermediário ser o melhor entre todos os possíveis i. Resumindo, buscamos a aresta $(i, i + 1)$ que minimize a conta $w(H_i, k) + w(k, H_{i+1}) - w(H_i, H_{i+1})$. Por fim, k é inserido em H exatamente entre H_i e H_{i+1}.

5.1. Pseudocode - Insere Vértice 1

```
// Heuristica Insere-Vertice-1
algoritmo Insere-Vertice-1( Entrada: G = (V,E, weight) ) → lista:
    Seja H um ciclo inicial qualquer
    enquanto H não é um ciclo hamiltoniano {
        k = -1
        min_weight = ∞
        // Entre os vertice v de H, procuramos um vertice u tal que
        // aresta (u, v) possue menor peso
        para cada v ∈ H {
            para cada u ∈ neighbours(v) {
                w = weight((v,u))
                se w < min_weight && u ∉ H {
                    min_weight = w
                    k = u
                }
            }
        }

        /*
        Percorremos H em busca de um melhora que possamos inserir
        um vertice k entre H_i e H_{i+1} tal que vale mais apena passa por k
        primeiro do que passar de H_i direto para H_{i+1}
        Ou seja, custo (H_i − H_{i+1}) > custo(H_i − k − H_{i+1})
    ○   */
        insert_best = ∞
        insert_at = -1
        para i = 1,2, ..., |H| {
            // def: w_uv = weight((u, v))
            insert_cost = w_{H_i k} + w_{k H_{i+1}} − w_{H_i H_{i+1}}
            se insert_cost < insert_best {
                insert_best = insert_cost
                insert_at = i
            }
        }
        // inserimos no meio entre H_i e H_{i+1}
        Insira k em insert_at no ciclo H
    }
    retorne H
```

Figura 4. Pseudo código da heurística Insere-Vértice-1 que utiliza um ciclo inicial H

5.2. Análise e Resultados - Insere Vértice 1

Como vimos no pseudo a heurística Insere Vértice 1 faz um procura para cada vértice que já $v \in H$ que pode ser todos os vértices V, e ainda buscamos todos os vizinhos de cada v, portanto só nessa parte há um custo de $O(|E||V|)$ e logo em seguida gastando

$O(|H|) \leq O(|V|)$ para encontrar i que é o índice de melhor custo de inserção de k em H, portanto $T_{InsereVertice1} \leq O(|E||V|)$.

Segue a tabela:

Insert Vertice 1					
Graphs	**ATT48**	**FIVE**	**DANTZIG42**	**FRI26**	**P01**
Execution time (s)	0.0001284	0.0000193	0.0001570	0.0000959	0.0000477
Approximation $w(H)$	37639	21	806	1035	344
Exact $w(H^*)$	33523	19	699	937	291
Ratio $w(H)/w(H^*)$	1.1228	1.1053	1.1531	1.1046	1.1821
Nº of vertices $\|V\|$	48	5	42	26	15

6. Insere Vértice 2

A heurística de Insere Vértice 2, também é da categoria de expansão, a única diferença entre a Insere Vértice 1 é a busca do vértice k que no lugar de buscar o k que possua menor custo em relação a os vértices do ciclo H até uma dada iteração, o Insere Vértice 2 testa todas os possíveis $k \in V$ tal que $k \notin H$ que minimize $w(H_i, k) + w(k, H_{i+1}) - w(H_i, H_{i+1})$, e por isso temos uma complexidade mais alta, mas fora isso é o mesmo conceito da versão anterior.

6.1. Pseudocode - Insere Vértice 2

```
// Heuristica Insere-Vertice-2
algoritmo Insere-Vertice-2( Entrada: G = (V,E, weight) ) → lista:
    Seja H um ciclo inicial qualquer
    enquanto H não é um ciclo hamiltoniano {
        /*
        Percorremos todo v ∈ V e ainda v ∉ H em busca de um melhora
        que possamos inserir um vertice k entre H_i e H_{i+1} tal que vale mais
        apena passa por k primeiro do que passar de H_i direto para H_{i+1}
        Ou seja, custo (H_i − H_{i+1}) > custo(H_i − k − H_{i+1})
    ○   */
        insert_best = ∞
        insert_at = -1
        para cada k ∈ E ∧ k ∉ H {
            para i = 1,2, ..., |H| -1{
                // def: w_uv = weight((u, v))
                insert_cost = w_{H_i k} + w_{k H_{i+1}} − w_{H_i H_{i+1}}
                se insert_cost < insert_best {
                    insert_best = insert_cost
                    insert_at = i
                }
            }
        }
        // inserimos no meio entre H_i e H_{i+1}
        Insira k em insert_at no ciclo H
    }
    retorne H
```

Figura 5. **Pseudo código da heurística Insere-Vértice-2 que utiliza um ciclo inicial H**

6.2. Análise e Resultados - Insere Vértice 2

Como vimos no pseudo a heurística Insere Vértice 1 é $O(|E||V|)$. Mas como no início buscando todos os possíveis $k \in V \wedge k \notin H$ exatamente $|H|$ vezes, implicando em um aumento na complexidade de tempo, subindo para $O(|E||V|^2)$.

Insert Vertice 2					
Graphs	**ATT48**	**FIVE**	**DANTZIG42**	**FRI26**	**P01**
Execution time (s)	0.0001046	0.0000195	0.0000937	0.0000609	0.0000386
Approximation $w(H)$	120399	23	710	1135	552
Exact $w(H^*)$	33523	19	699	937	291
Ratio $w(H)/w(H^*)$	3.5915	1.2105	1.0157	1.2113	1.8969
Nº of vertices $\|V\|$	48	5	42	26	15

7. Resultados e Discussão

Apresento a tabela em ordem das médias entre razão entre a aproximação e o exato, ou seja, quanto menor mais parecido com a solução ótimo, a tabela está em ordem do melhor para o pior.

Comparing Heuristics		
Heuristic	$\sum$**Ratio**$/5$	**Result**
Insere Vertice 1	(1.1228+1.1053+1.1531+1.1046+1.1821)/5	1.13358
Bellmore & Nemhauser	(1.2096 + 1.1053 + 1.3677 + 1.1868 + 1.0000)/5	1.17388
Twice Around	(1.2692 + 1.1053 + 1.3276 + 1.3436 + 1.3711)/5	1.28336
Christofides	(1.4874 + 1.0000 + 1.4864 + 1.5422 + 1.3952)/5	1.38224
Insere Vertice 2	(3.5915+1.2105+1.0157+1.2113+1.8969)/5	1.78518

Podemos notar que apesar dos algoritmos que usam uma MST, como de Twice Around e Chistofides, possuírem garantia otimização, ou seja, não pode ser pior que uma constante $c \in \mathbb{R}$ vezes o custo de H^* (isso se considerarmos a *desigualdade triangular*), há casos que uma heurística mais simples pode gerar um custo total menor, e ainda ser mais rápido computacionalmente.

Ainda, entre Twice Around e Chistofides, apesar da heurística do segundo possuir uma otimização mais agressiva , por conta uso do matching perfeito de custo mínimo, para garantir ainda melhor "*upper bound*", ainda sim pode acontecer do Twice Around encontrar um ciclo de custo menor que o ciclo encontrado por Christofides, em 3 dos 5 grafos Twice Around foi melhor que Christofides.

Um outro exemplo disso é Insere Vértice 1 e 2, apesar da versão 2 ter complexidade de tempo mais alta que a versão 1, a primeira produziu o melhor resultado entre a média das razões de todas as heurísticas usadas, ficando entre um erro abaixo de um fator de 1.2 vezes pior que o ciclo ótimo, sendo assim Insere Vértice 1 foi o algoritmo que teve mais sucesso para esses grafos completos.

Também, dependendo do grafo, uma escolha gulosa pode ser a melhor, foi o caso do grafo P01 de 15 vértices e 105 arestas. Nesse grafo, escolhendo o vizinho mais próximo (Bellmore & Nemhause) foi encontrado um caminho ótimo, isto é $w(H) = w(H^*)$, não necessariamente o mesmo ciclo, mas com o mesmo custo. Ainda vale notar que Bellmore & Nemhause se saiu bem em geral, não foi "apenas um golpe de sorte" em um grafo específico, mas sim é a heurística de ficou em segundo lugar entre as todas as cinco, apesar de ser a mais simples de todas.

8. Considerações Finais

Neste trabalho foi realizado um mapeamento de 5 (cinco) heurísticas para o problema de encontrar um ciclo hamiltoniano de custo mínimo para grafos completos que é NP-Hard, um dos objetivos foi identificar a relação de tempo de execução, complexidade de tempo, e relação do ciclo aproximado e o ciclo exato dessas cinco heurísticas.

Foi feito, além dessa relação, a discussão da escolha de algumas dessas heurísticas em detrimento de outras, salientando que alguns algoritmos mais simples podem ser, na prática, melhor que alguns mais elaborados. Além disso, fizemos a explicação tanto

descrita, como por pseudo código, das ideias e conceitos por trás de cada uma dessas heurísticas.

Por fim, tendo elaborado esse artigo, é perceptível que ainda há muito a ser explorado, outras heurísticas existem que podem ser agregadas em outro artigo e discutidas, havendo a possibilidade de utilizar outros grafos que não sejam completos ou ainda que garantam a desigualdade triangular, além de outras restrições.

Referências

[dat] Data for the Traveling Salesperson Problem tsp. `https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html`. Accessed: 2022-05-29.

[max] Maximum Weighted Matching. `http://jorisvr.nl/article/maximum-matching`. Accessed: 2022-06-01.

[Goldbarg and Goldbarg 2012] Goldbarg, M. and Goldbarg, E. (2012). *Grafos*. Elsevier Brasil.