Relatório Final de Iniciação Científica
Modalidade PDPD – Edital 02/2018

Algoritmos para os problemas do Caixeiro Viajante e da Árvore de Steiner

Resumo

No problema do Caixeiro Viajante temos um conjunto de cidades e queremos encontrar uma rota de comprimento mínimo que passa por todas elas exatamente uma vez. No problema da Árvore de Steiner temos temos um conjunto de pontos e queremos conectá-los por uma rede que pode utilizar outros pontos intermediários de forma que a soma dos comprimentos das linhas que ligam os pontos seja mínima. Ambos problemas são clássicos e centrais na área de otimização combinatória, com diversas aplicações práticas.

Este relatório descreve as atividades desenvolvidas pelo aluno Marcelo Tranche de Souza Junior sua iniciação científica. Elas envolveram o estudo de diferentes algoritmos para os problemas mencionados, bem como outros problemas relacionados.

Palavras-chave: Algoritmos; Caixeiro Viajante; Árvore de Steiner; Otimização Combinatória; Algoritmos de Aproximação.

Área do conhecimento: Teoria da Computação.

Autor:
Marcelo Tranche de Souza Junior
(voluntário)

Orientadora:
Profa. Dra. Carla Negri Lintzmayer

26 de agosto de 2019

Sumário

1 Resumo das atividades desenvolvidas

Nos primeiros cinco meses do projeto, o aluno estudou técnicas e resultados básicos da ciência da computação, indicados a seguir:

- Revisão de conceitos básicos de algoritmos e programação
- Algoritmos recursivos e provas por indução
- Estruturas de dados básicas (listas, pilhas, filas, árvores)
- Conceitos básicos de análise de algoritmos (notação assintótica)
- Divisão e conquista (MergeSort)
- Algoritmos gulosos (problema do escalonamento, mochila fracionária)
- Programação dinâmica (problema da mochila inteira)
- Grafos e buscas em grafos
- Árvore geradora mínima (Kruskal)
- Caminhos mínimos em grafos (Dijkstra)
- Programação linear e programação linear inteira
- Ideia sobre NP-dificuldade e abordagens para lidar com os problemas

Não cobrimos todos esses itens com detalhes, mas esse estudo serviu para dar uma base inicial ao aluno, que estava em seu primeiro ano. Note como algumas abordagens envolveram o estudo de um único algoritmo daquele tipo.

Nos meses seguintes o aluno estudou resultados clássicos referentes aos dois problemas escolhidos. O foco foi em algoritmos de aproximação, de forma que inicialmente foi visto um algoritmo simples para o problema da cobertura por vértices, para introdução e compreensão do conceito, para que em seguida o aluno estudasse de fato detalhes sobre o que são ambos os problemas e fazer um levantamento dos resultados clássicos existentes para que posteriormente tais resultados fossem estudados.

2 Metodologia

A área de otimização combinatória é muito rica em técnicas e abordagens. Algumas dessas abordagens envolvem experimentação prática com algoritmos, como por exemplo quando o estudo trata de heurísticas ou de algoritmos exatos como *branch and bound* ou programas lineares. Nesses casos, a implementação dos algoritmos para comparações faz parte da metologia de pesquisa. No entanto, nosso foco foi a pesquisa teórica dos problemas mencionados dando ênfase ao estudo de algoritmos de aproximação. Nesse caso, a metodologia de pesquisa mais encontrada na literatura considera apenas análise assintótica dos algoritmos e demonstrações matemáticas de seus fatores de aproximação.

Sendo assim, no início do projeto o candidato estudou técnicas e resultados básicos da área da ciência da computação, em particular com relação a projeto e análise de algoritmos. Também foi necessário que o candidato adquirisse conhecimentos sobre técnicas específicas para elaboração de soluções para problemas de otimização, sendo elas a descrição de problemas por meio de programação linear e programação linear inteira, e também a elaboração de algoritmos de aproximação. Isso feito por meio de estudo de livros da área de ciência da computação, sendo o livro "Algoritmos" [2] usado como base principal.

Em seguida, o candidato estudou resultados importantes referentes a primeiramente o problema de cobertura de vértices, para em seguida partir para os problemas do Caixeiro Viajante e da Árvore de Steiner, problemas foco do projeto, sendo isso feito também por meio do estudo de livros e, eventualmente, de artigos científicos, cujos acessos são fornecidos pela UFABC.

Esses métodos, inclusive, são usuais na área de teoria da computação e otimização combinatória: aprender técnicas já existentes podem no futuro auxiliar o aluno a resolver problemas similares, na busca de novos resultados.

Partindo disso, o estudante documentou os enunciados dos problemas abordados, suas descrições por programação linear, e também soluções importantes para eles, apresentando seus respectivos algoritmos e suas provas por meio de lemas e teoremas.

3 Resultados

Otimização combinatória é uma área da ciência da computação que estuda problemas nos quais o objetivo é encontrar a melhor solução, ou a que mais se aproxima da melhor, dentro de um conjunto de domínio finito de soluções.

Problemas de otimização combinatória em geral são NP-difíceis, ou seja, não podemos resolvê-los em tempo polinomial, a menos que P = NP, fazendo com que seja necessário utilizar outras técnicas para encontrar soluções razoáveis para eles. A seguir contemplaremos duas dessas técnicas: programação linear, na Seção 3.1, e algoritmos de aproximação, na Seção 3.2. Como exemplo de aplicação das técnicas, serão mostradas aplicações sobre o clássico problema *Vertex cover*, descrito na Seção 3.3. Os problemas principais que foram estudados durante o período da iniciação científica são descritos em seguida. O *Traveling salesman problem* e seus resultados principais são descritos na Seção 3.4 enquanto o *Steiner tree problem* e seus resultados são descritos na Seção 3.5.

3.1 Programação linear

Programação linear é uma maneira formal de descrever um problema de otimização, seja ele de maximização (*Programa linear de maximização*) ou de minimização (*Programa linear de minimização*). De forma geral, todo programa linear tem um conjunto de **variáveis de decisão**, que indicam as escolhas feitas para chegar a uma solução, de **restrições**, que descrevem as características do problema, e uma **função de objetivo**, que indica o custo das soluções e se o programa linear é de maximização ou de minimização. A função de objetivo e as restrições de um programa linear são representadas por equações ou inequações lineares.

Em geral um programa linear com n variáveis e m restrições é descrito da seguinte forma:

$$\begin{aligned} \text{minimizar} \quad & \sum_{j=1}^{n} c_j x_j & \\ \text{sujeito a} \quad & \sum_{j=1}^{n} a_{ij} x_j \geq b_i & \forall i \in \{1, \ldots, m\} \\ & x_j \geq 0 & \forall j \in \{1, \ldots, n\} \end{aligned}$$

onde a_{ij}, b_i e c_j são constantes, x_j são as variáveis de decisão, $\sum_{j=1}^{n} a_{ij} x_j \geq b_i$ são as restrições e $\sum_{j=1}^{n} c_j x_j$ é a função objetivo.

Qualquer conjunto de valores que as variáveis possam assumir de maneira que continuem respeitando as restrições é chamado de **solução viável**, e o valor resultante da função de objetivo de acordo com os valores das soluções viáveis é chamado de **valor de objetivo**. Então, pode-se chamar qualquer conjunto de valores das variáveis que melhor otimiza o valor de objetivo de **solução ótima**, e o seu resultado de **valor de objetivo ótimo**.

3.1.1 Teorema fundamental de programação linear

Todo programa linear pode ter uma solução ótima com valor de objetivo finito, ser impossível ou ser ilimitado [2, p. 647].

Para que um algoritmo que se proponha a resolver programas lineares funcione, este, quando receber qualquer programa linear, deve retornar algo coerente para cada uma das possibilidades citadas.

3.1.2 Procedimento gráfico

Uma maneira de representar e solucionar um problema de programação linear é através da observação da área formada pelas restrições lineares quando representadas em um gráfico que possui as variáveis de decisão $x_1, x_2, \ldots, x_n$ como eixos.

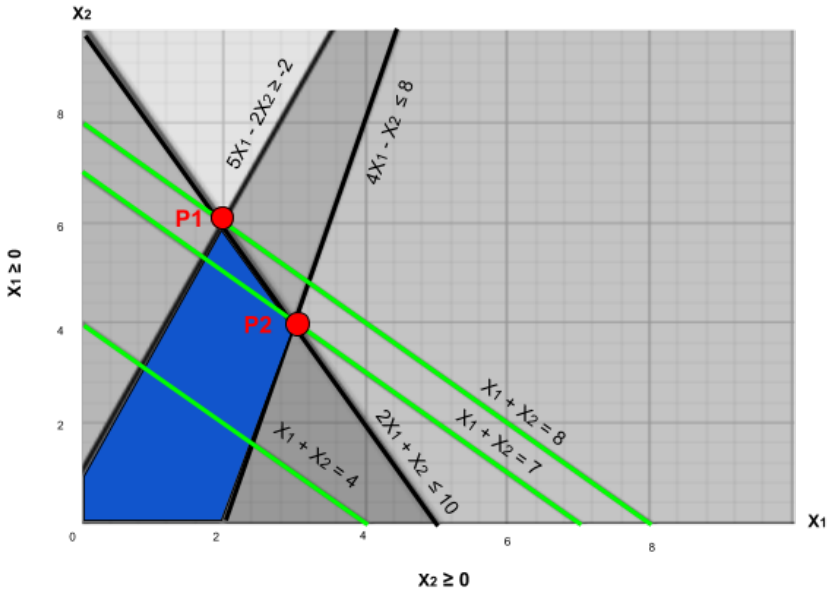

Figura 1: Gráfico do programa linear mostrado em (1).

A região delimitada pelas restrições conterá todas as soluções possíveis para o problema e é chamada de **simplex**. Para se encontrar a solução ótima para a função objetivo $f(x_1, x_2, \ldots, x_n)$ do problema, deve-se encontrar a interseção da região simplex com $f(x_1, x_2, \ldots, x_n) = b$, tal que b seja o valor de objetivo máximo (ou mínimo) do problema. Caso a interseção aconteça com algum vértice do simplex, então o problema possui apenas uma solução ótima, mas caso a função passe por mais pontos da região delimitada, então todos esses pontos serão soluções ótimas.

Como exemplo [2, p. 613], considere o seguinte programa linear:

$$\begin{aligned} \text{maximizar} \quad & x_1 + x_2 \\ \text{sujeito a} \quad & 4x_1 - x_2 \leq 8 \\ & 2x_1 + x_2 \leq 10 \\ & 5x_1 - 2x_2 \geq -2 \\ & x_1, x_2 \geq 0 \end{aligned} \tag{1}$$

A Figura 1 representa o gráfico do problema em questão. A região destacada em azul contém todas as soluções possíveis para o problema. O gráfico mostra que $x_1 + x_2 = 4$ possui pontos para algumas soluções possíveis, mas não apresenta o valor de objetivo máximo, uma vez que $x_1 + x_2 = 8$ (P_1) e $x_1 + x_2 = 7$ (P_2) também intersectam o simplex. Como esses pontos são interseções nos vértices da região, isso mostra que o problema possui apenas uma solução ótima, sendo $P_1 = (2, 6)$ ou $P_2 = (3, 4)$. Portanto, basta escolher o vértice que resultará no maior valor de objetivo que, no caso desse problema, será P_1.

3.1.3 Formas de representação

Existem duas formas de representar um programa linear que facilitam o entendimento e solução do problema que ele está tentando resolver. Essas são chamadas de *forma padrão* e *forma relaxada*, apresentadas a seguir.

Um programa linear está na **forma padrão** se sua função objetivo é de **maximização**, suas restrições são desigualdades com sinal de **menor ou igual** e as variáveis tenham **restrições de não negatividade**. Se alguma dessas características não estiver sendo seguida por um dado programa, é possível convertê-lo para que assuma a forma padrão, mantendo-o equivalente ao programa original.

Caso a função de objetivo $f(x_1, x_2, \ldots, x_n)$ com solução de objetivo z seja de minimização, para que ela vire uma função de maximização ainda equivalente basta fazer $-f(x_1, x_2, \ldots, x_n)$ e, consequentemente, sua solução de objetivo será $-z$.

Caso exista uma restrição de igualdade $g(x_1, x_2, \ldots, x_n) = b$, basta criar duas restrições $g(x_1, x_2...x_n) \leq b$ e $-g(x_1, x_2, \ldots, x_n) \leq -b$.

Por fim, caso alguma variável x não tenha restrição de não negatividade, será necessário substituir essa variável x por duas variáveis x_p e x_q, de maneira que $x = x_p - x_q$, e a restrição $x_p, x_q \geq 0$ deve ser adicionada.

A **forma relaxada** se diferencia da forma padrão pois todas suas restrições, exceto as restrições de não negatividade, devem ser **restrições de igualdade**.

Também é possível transformar um programa linear com restrições de desigualdade $g(x_1, x_2...x_n) \leq b$ para a forma relaxada. Basta utilizar $x_p = b - g(x_1, x_2...x_n)$, sendo x_p chamada de **variável relaxada**, e essa também deve possuir a restrição de não negatividade ($x_p \geq 0$). Todas as variáveis relaxadas criadas devem ficar ao lado esquerdo da nova restrição e são chamadas de **variáveis básicas**, e as que ficaram do lado direito são chamadas de **variáveis não básicas**.

3.1.4 Algoritmo Simplex

O algoritmo Simplex é um algoritmo para solução de programas lineares de tempo de execução não polinomial nos piores casos, porém bastante ágil na maioria das vezes.

Esse algoritmo primeiramente assume que o programa linear está na forma relaxada. A partir disso, ele determina uma **solução básica**, que é uma configuração de valores básicos $a_1, a_2, \ldots, a_n$ para as variáveis $x_1, x_2, \ldots, x_n$ do problema que dê uma solução possível para o programa. Essa solução básica pode ser dada escolhendo valores possíveis para as variáveis não básicas, e a partir das restrições de igualdade determinar os valores das variáveis básicas. Vale lembrar que os valores das variáveis ainda devem seguir as restrições de não negatividade. Uma forma sim-

ples de construir a solução básica, se possível, é escolhendo o valor 0 para todas as variáveis não básicas.

A partir disso, o algoritmo simplex consiste na substituição algébrica de determinadas variáveis não básicas por variáveis básicas, de maneira que o programa linear resultante seja equivalente ao inicial e, portanto, suas soluções ótimas sejam as mesmas. As substituições devem seguir o processo de **criações de pivôs** descrito a seguir.

A partir da solução básica, deve-se escolher uma das variáveis não básicas, que não faça o valor de objetivo diminuir conforme ela aumente, e aumentá-la até um valor máximo que ainda faça com que todas as variáveis básicas respeitem a restrição de não negatividade. A variável escolhida é chamada de **variável de entrada**. Com isso, a restrição que permitir o menor valor para a variável de entrada será a **restrição mais rígida**, e a variável básica associada a essa restrição será chamada de **variável de saída**.

Após isso, deve-se fazer uma substituição algébrica entre a variável de entrada e a variável de saída, e com isso se obterá um programa linear diferente do inicial, porém equivalente. Então, os valores das variáveis não básicas que foram escolhidos durante a formação da solução básica continuam os mesmos e, consequentemente, os valores das variáveis básicas devem ser calculados novamente, resultando em uma nova solução básica.

Esse processo deve ser repetido até que a função de objetivo do programa linear resultante possua apenas variáveis que, se aumentadas, diminuam o valor de objetivo. Quando isso acontecer, a solução básica desse programa também será uma de suas soluções ótimas e, consequentemente, também será do programa inicial.

3.1.5 Dualidade

A dualidade de programação linear é uma propriedade que prova que todo problema de programação linear possui uma outra estrutura de programação linear associada chamada de **programa linear *dual***, que muitas vezes ajuda na compreensão do problema original. Em particular, a resolução de um deles constitui na resolução do outro.

A criação dessa nova estrutura acontece a partir do programa linear original, chamado de **programa linear primitivo**. Considere o seguinte programa linear

primitivo na forma padrão:

$$\begin{aligned}
\text{maximizar} \quad & a_1x_1 + a_2x_2 + \cdots + a_nx_n \\
\text{sujeito a} \quad & b_{11}x_1 + b_{12}x_2 + \cdots + b_{1n}x_n \geq c_1 \\
& b_{21}x_1 + b_{22}x_2 + \cdots + b_{2n}x_n \geq c_2 \\
& \cdots \\
& b_{m1}x_1 + b_{m2}x_2 + \cdots + b_{mn}x_n \geq c_m \\
& x_1, x_2, \ldots, x_n \geq 0
\end{aligned}$$

Seu programa dual terá as variáveis de decisão $y_1, y_2, \ldots, y_m$. A ideia para gerá-lo é multiplicar cada uma das m restrições do programa primitivo por uma dessas variáveis. Com isso, a função de objetivo do programa dual será $c_1y_1 + c_2y_2 + \cdots + y_mx_m$, enquanto que as restrições serão formadas pelas constantes b_{ij} e a_i. Como resultado, temos:

$$\begin{aligned}
\text{minimizar} \quad & c_1y_1 + c_2y_2 + \cdots + c_my_m \\
\text{sujeito a} \quad & b_{11}y_1 + b_{21}y_2 + \cdots + b_{m1}y_m \geq a_1 \\
& b_{12}y_1 + b_{22}y_2 + \cdots + b_{m2}y_m \geq a_2 \\
& \cdots \\
& b_{1n}y_1 + b_{2n}y_2 + \cdots + b_{mn}y_m \geq a_n \\
& y_1, y_2, \ldots, y_m \geq 0
\end{aligned}$$

Pode-se provar que a solução desenvolvida para o problema é ótima se a solução resultante do programa linear primitivo for igual à solução resultante do programa linear dual. Ou seja, o algoritmo encontra a solução ótima se, e somente se, $a_1x_1 + a_2x_2 + \cdots + a_nx_n = c_1y_1 + c_2y_2 + \cdots + c_my_m$.

3.1.6 Programação linear inteira (PLI)

Um programa linear inteiro é uma classe dos programas lineares cujas variáveis que formulam o problema determinado podem apenas assumir valores inteiros.

Problemas que são formulados em PLI são NP-difíceis [14], ou seja, são problemas cuja a solução ótima provavelmente não pode ser encontrada em tempo polinomial. Assim, eles não podem ser resolvidos através do algoritmo simplex (veja Seção 3.1.4), por exemplo, fazendo com que seja necessária a implementação de outros métodos, como algoritmos de aproximação (veja Seção 3.2), para desenvolver soluções aproximadas.

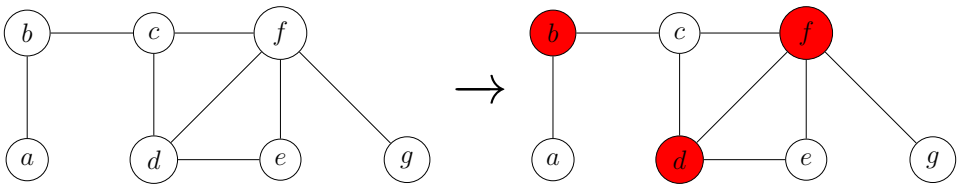

Figura 2: Grafo G à esquerda e uma cobertura por vértices de G em destaque à direita.

3.2 Algoritmos de aproximação

Algoritmos de aproximação são algoritmos de tempo polinomial que devolvem uma solução cujo custo tem garantia de ser próximo ao custo da solução ótima.

Seja C o custo da solução produzida por um algoritmo e C^* o custo da solução ótima. Em problemas de minimização claramente vale que $0 < C^* \leq C$, enquanto que em problemas de maximização vale que $0 < C \leq C^*$. O algoritmo é uma $p(n)$-aproximação, ou tem uma taxa de aproximação $p(n)$, se, considerando um problema de minimização, a relação $C \leq p(n)C^*$ for sempre verdade para qualquer entrada de tamanho n. Se o problema for de maximização, a relação deve ser $p(n)C^* \leq C$.

Note que para problemas de maximização vale que $p(n) \geq 1$, enquanto que para problemas de minimização vale $p(n) \leq 1$. Então, fica claro que quanto mais próximo $p(n)$ estiver de 1, melhor será a aproximação garantida pelo algoritmo.

3.2.1 Esquema de aproximação

Um esquema de aproximação é um algoritmo de aproximação que recebe, além da instância do problema a ser resolvido, um valor $\varepsilon > 0$ fixo, tal que o fator $p(n)$ de aproximação do algoritmo é $(1 + \varepsilon)$ se o problema é de minimização e é $(1 - \varepsilon)$ se o problema é de maximização. Em outras palavras, se um problema admite um esquema de aproximação, então ele permite que a aproximação seja feita em qualquer grau requerido. Em geral, no entanto, o tempo de execução do algoritmo tende a ser maior quanto menor o valor de ε.

3.3 Problema de cobertura por vértices

O problema *Vertex cover*, ou **problema de cobertura por vértices**, é um problema NP-difícil [2, p. 794-795] bem conhecido. Nele, um grafo não orientado $G = (V, E)$ é dado na entrada. O objetivo é achar um subconjunto $V' \subseteq V$ com o menor número de vértices tal que todos os vértices de V' cubram as arestas, ou seja, cada aresta $uv \in E$ é tal que $u \in V'$ ou $v \in V'$. A Figura 2 mostra um exemplo de uma cobertura de vértices para um determinado grafo.

3.3.1 Abordagem com programação linear inteira

Seja x_u uma variável cujo valor é 1 se o vértice u fizer parte da solução ou é 0, caso contrário. Considere o seguinte programa linear:

$$\text{minimizar} \quad \sum_{u \in V} x_u \tag{2}$$

$$\text{sujeito a} \quad x_u + x_v \geq 1 \qquad \forall uv \in E \tag{3}$$

$$x_u \in \{0, 1\} \qquad \forall u \in V \tag{4}$$

Note que ele formaliza o problema de cobertura por vértices: a função objetivo em (2) indica que queremos minimizar o número de vértices escolhidos e a restrição (3) garante que cada aresta tenha pelo menos um vértice cobrindo-a.

3.3.2 Abordagem com algoritmo de aproximação

Considere o algoritmo APPROX-VERTEX-COVER a seguir.

Algoritmo 1: APPROX-VERTEX-COVER

```
   Entrada: G = (V, E)
1  início
2      C ← ∅
3      E' ← E
4      repita
5          seja uv uma aresta arbitrária de E'
6          C ← C ∪ {u, v}
7          remover de E' toda aresta incidente a ambos u e v
8      até E' = ∅;
9      retorna C
10 fim
```

No algoritmo, o conjunto C será a cobertura por vértices construída e na linha 2 ele é inicializado com o conjunto vazio. Na linha 3, o conjunto E' é inicializado com as arestas do grafo. O *loop* das linhas 4 à 8 escolhe uma aresta uv qualquer de E' a cada passo, faz com que C seja incrementado com ambos os vértices u e v e elimina todas as arestas cobertas por u e v de E'. É fácil perceber que o algoritmo executa em tempo polinomial. O Teorema 1 mostra que esse algoritmo é de aproximação para o vertex cover.

Teorema 1. *O algoritmo* APPROX-VERTEX-COVER *é uma 2-aproximação para o vertex cover.*

Demonstração. Note que os vértices escolhidos para fazerem parte da solução C

cobrem todas as arestas do grafo, já que só são removidas arestas de E' que são cobertas por vértices em C e que o laço executa até que E' fique vazio.

Note ainda que a solução é feita a partir de escolhas de arestas arbitrárias que não possuem vértices em comum e, portanto, essas arestas formam um emparelhamento maximal no grafo G. Seja M o emparelhamento que contém as arestas que foram escolhidas pelo algoritmo.

Note que uma solução ótima C^* para o vertex cover possui pelo menos um vértice para cobrir cada aresta de M, de modo que $|M| \leq |C^*|$. Como a solução C oferecida pelo algoritmo possui os dois vértices de cada aresta de M, temos que $|C| = 2|M|$. Portanto, $|C| \leq 2|C^*|$, de onde vemos que o algoritmo é uma 2-aproximação. □

3.4 Problema do caixeiro viajante

Traveling salesman problem, ou o **problema do caixeiro viajante**, é um famoso problema NP-difícil [2, p. 799], que tem como objetivo achar o caminho de menor custo que um vendedor deve viajar, de maneira que ele visite todas as cidades uma única vez e depois volte para sua cidade de partida.

Formalmente, a entrada do problema pode ser modelada por um grafo não orientado completo $G = (V, E)$, onde cada vértice representa uma cidade e cada aresta ij possui um peso $c(ij)$, que representa o custo de viajar da cidade i à cidade j. Deseja-se encontrar um **ciclo hamiltoniano de custo mínimo** em G, ou seja, um percurso que começa em um vértice r e passa por cada um dos vértices do grafo uma única vez retornando ao vértice r e cuja soma das arestas percorridas seja a menor possível.

3.4.1 Abordagem com programação linear inteira

Seja x_{ij} uma variável cujo valor é 1 se a aresta $ij \in E$ for escolhida para fazer parte do ciclo ou é 0, caso contrário. Considere o seguinte programa linear inteiro [5]:

$$\text{minimizar} \quad \sum_{j=1}^{n} \sum_{i=1}^{n} c(ij) x_{ij} \tag{5}$$

$$\text{sujeito a} \quad \sum_{i=1}^{n} x_{ij} = 2 \qquad \forall j \in V \tag{6}$$

$$\sum_{i,j \in S} x_{ij} \leq |S| - 1 \qquad \forall S \subset V, S \neq \emptyset \tag{7}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \in V, i < j \tag{8}$$

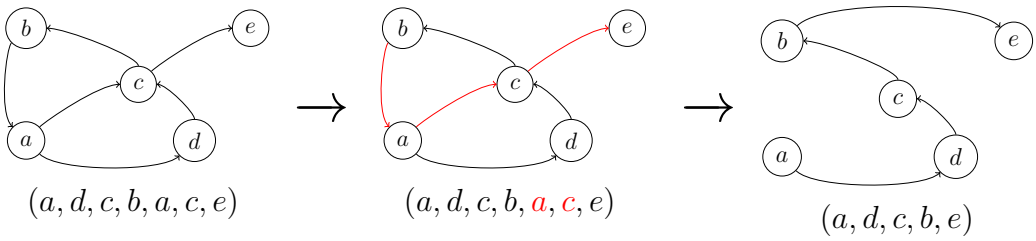

Figura 3: Suponha que temos um grafo completo com os vértices $\{a, b, c, d, e\}$. Na figura da esquerda, mostramos um passeio nesse grafo, (a, d, c, b, a, c, e). Na figura central, destacamos o último trecho desse passeio, b, a, c, e. A figura da esquerda mostra a substituição desse trecho pela aresta direta entre b e e.

Note que ele formaliza o problema do caixeiro viajante: a função objetivo em (5) indica que queremos minimizar o custo das arestas escolhidas, a restrição (6) garante que cada vértice será visitado uma vez pelo ciclo (pois duas arestas devem sair desse vértice), e a restrição (7) garante que não serão formados ciclos que não sejam hamiltonianos (todo subconjunto de vértices de G que não seja o próprio $V(G)$ deve ter no máximo $|S| - 1$ arestas ligando seus vértices – dessa forma, não é possível S conter um ciclo).

Note que existem $O(2^{|V(G)|})$ restrições do tipo (7), uma para cada possível subconjunto de vértices, fazendo que essa formulação seja inviável conforme o tamanho do grafo aumenta.

3.4.2 Abordagem com algoritmo de aproximação

O problema geral do caixeiro viajante não admite um algoritmo que aproxime a solução com taxa de aproximação constante [13]. Uma versão particular do problema que tem muita relevância prática e contorna essa dificuldade é o **caixeiro viajante métrico**. Um grafo G com custos nas arestas é dito **métrico** se o custo de ir de um vértice u a um vértice w não for maior do que o custo de um caminho alternativo que, antes de chegar a w, passa por um vértice intermediário v. Formalmente, G é métrico se para todo trio de vértices u, v, w temos que $c(uw) \leq c(uv) + c(vw)$. Apresentaremos a seguir alguns algoritmos para o caixeiro viajante métrico.

O algoritmo que iremos apresentar, bem como outros nas seções seguintes, utiliza ***shortcuts***. Eles são úteis para remover trechos já visitados em *passeios*. Dado um passeio $(a_1, a_2, a_3, a_4, \ldots, a_k)$ em um grafo, onde cada a_i é um vértice e todo $a_i a_{i+1}$ é aresta do grafo, definimos um *shortcut* como a substituição de um trecho $a_i, a_{i+1}, \ldots, a_j$ pela aresta direta $a_i a_j$. A Figura 3 mostra um exemplo de um *shortcut* em um determinado passeio.

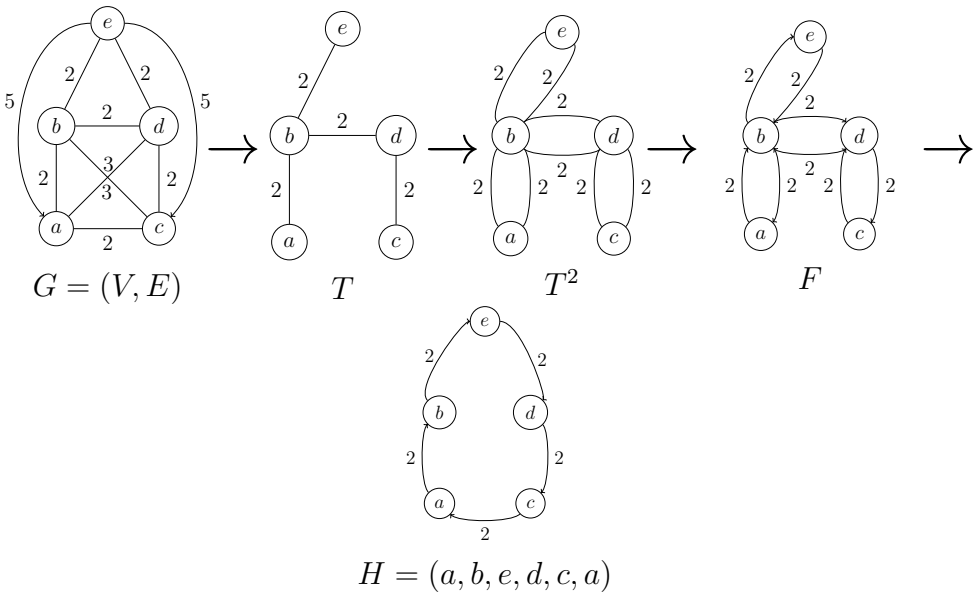

Figura 4: Exemplo de execução do algoritmo METRIC-TSP-Double-Tree.

Algoritmo *double tree*. O algoritmo a seguir recebe o grafo G e a função c de custo nas arestas.

Algoritmo 2: METRIC-TSP-DOUBLE-TREE

Entrada: $G = (V, E), c$

1 **início**
2 $\quad T \leftarrow$ ÁRVOREGERADORAMÍNIMA(G, c)
3 $\quad T^2 \leftarrow T$ com arestas duplicadas
4 $\quad F \leftarrow$ Trilha Euleriana sobre T^2
5 $\quad H \leftarrow$ Ciclo Hamiltoniano fazendo *shortcuts* em F
6 $\quad$ **retorna** H
7 **fim**

Na linha 2 o algoritmo chama uma função que retorna uma árvore geradora mínima do grafo G e armazena essa árvore em T (isso pode ser feito com o algoritmo de Kruskal ou Prim, por exemplo). Na linha 3, as arestas de T são duplicadas, de maneira que agora todos os vértices possuem grau par. Com isso, na linha 4 cria-se uma trilha Euleriana F sobre T^2, ou seja, uma trilha onde todas as arestas de T^2 são visitadas e nenhuma delas é repetida, porém vértices muito provavelmente são. A partir da trilha Euleriana F formada, podemos criar um ciclo Hamiltoniano eliminando vértices que se repetem em F por meio de *shortcuts*. A Figura 4 dá um exemplo de como o algoritmo funciona.

Teorema 2. *O algoritmo* METRIC-TSP-DOUBLE-TREE *é uma 2-aproximação.*

Demonstração. Seja H^* uma solução ótima. Note que se uma aresta dessa solução for removida, teremos uma árvore geradora para G, porém de custo maior ou igual ao custo da árvore T, que é mínima. Logo, $c(T) \leq c(H^*)$.

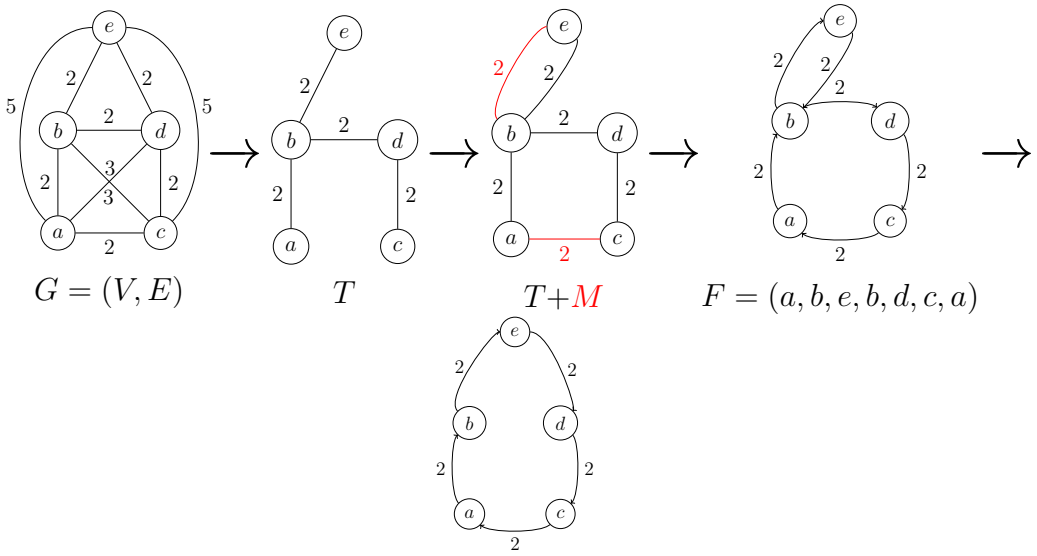

Figura 5: Exemplo de execução do algoritmo METRIC-TSP-Christofides.

É fácil observar que $c(F) = c(T^2) = 2c(T)$, e como arestas de F são retiradas para formar H, temos que $c(H) \leq c(F)$. Portanto, $c(H) \leq 2c(T) \leq 2c(H^*)$, de onde vemos que o algoritmo é uma 2-aproximação. □

Algoritmo de Christofides. O algoritmo a seguir é uma melhoria do anterior [1].

Algoritmo 3: METRIC-TSP-Christofides

Entrada: $G = (V, E), c$

1 **início**
2 $T \leftarrow$ ÁrvoreGeradoraMínima(G, c)
3 $M \leftarrow$ Emparelhamento perfeito de custo mínimo sobre os vértices de grau ímpar de T
4 $F \leftarrow$ Trilha Euleriana sobre $T + M$
5 $H \leftarrow$ Ciclo Hamiltoniano fazendo *shortcuts* em F
6 **retorna** H
7 **fim**

Na linha 2 o algoritmo chama uma função que retorna uma árvore geradora mínima do grafo G e armazena essa árvore em T. Na linha 3 é criado um emparelhamento apenas sobre os vértices de grau ímpar da árvore T. Isso é feito pois no grafo $T + M$ todos os vértices possuem grau par. Assim, na linha 4 podemos criar uma trilha Euleriana F sobre $T + M$. A partir da trilha Euleriana formada, criamos um ciclo Hamiltoniano H eliminando vértices que se repetem em F, da mesma forma do algoritmo anterior. A Figura 5 dá um exemplo de como o algoritmo funciona.

Lema 3. *O emparelhamento M gerado na linha 3 do algoritmo* METRIC-TSP-Christofides *tem custo $c(M) \leq \frac{1}{2}OPT$, onde OPT é o custo de uma solução*

ótima para o caixeiro viajante.

Demonstração. Note que podemos fazer *shortcuts* na solução ótima do TSP para obter um ciclo R sobre os vértices de grau ímpar da árvore T que é gerada na linha 2, de forma que $c(R) \leq OPT$. Esse ciclo R dá origem a dois emparelhamentos M_1 e M_2, e como M é um emparelhamento perfeito de custo mínimo, $c(M) \leq \min\{c(M_1), c(M_2)\}$.

Supondo, sem perda de generalidade, que $c(M_1) \leq c(M_2)$, logo $c(M_1) \leq \frac{1}{2}(c(M_1)+c(M_2))$, e como a união dos dois emparelhamentos resulta em R, temos que $\frac{1}{2}(c(M_1)+c(M_2)) = \frac{1}{2}c(R)$. Portanto $c(M) \leq \frac{1}{2}OPT$. □

Teorema 4. *O algoritmo* METRIC-TSP-CHRISTOFIDES *é uma* $\frac{3}{2}$*-aproximação.*

Demonstração. Note que $c(H) \leq c(F)$ uma vez que arestas de F são retiradas para formar H. Por sua vez, $c(F) = c(M) + c(T)$.

Como mostrado no Lema 3, $c(M) \leq \frac{1}{2}OPT$ e, se considerarmos uma solução ótima H^* para o TSP e tirarmos uma aresta dela, teremos uma árvore geradora porém de custo maior ou igual ao custo da árvore T, que é uma árvore geradora mínima. Logo, $c(T) \leq OPT$.

Portanto, $c(H) \leq \frac{3}{2}OPT$, de onde vemos que o algoritmo é uma $\frac{3}{2}$-aproximação. □

Algoritmo *cheapest insertion*. O algoritmo a seguir se baseia em um método apresentado por Nicholson [11] e analisado por Rosenkrantz *et al.* [12]. Ele recebe o grafo G, uma função de custo c sobre as arestas e um vértice inicial q (pode ser escolhido aleatoriamente).

Algoritmo 4: METRIC-TSP-CHEAPEST-INSERTION

Entrada: $G = (V, E), c, q$

1 **início**
2 seja r um vértice de G que minimize $c(qr)$
3 $T \leftarrow Subtour(q, r, q)$
4 **para cada** $x \in V - V(T)$ **faça**
5 seja $(i, j) \in T$
6 insira x em T que minimize $c(ix) + c(xj) - c(ij)$
7 **fim**
8 **retorna** T
9 **fim**

A ideia do algoritmo é manter um ciclo T sobre um conjunto de vértices e aumentar T, para que eventualmente ele vire um ciclo hamiltoniano, escolhendo um vértice que não está em T da seguinte forma. Seja ij uma aresta do ciclo T

atual e x um vértice que não está em T. O ciclo T pode ser aumentado trocando a aresta ij pelas arestas ix e xj. Esse ciclo T é chamado *subtour*.

Na linha 3 inicia-se um *subtour* que começa e termina no vértice inicial q e passa pelo vértice r, sendo esse o vértice que proporcionará o menor valor $c(qr)$. O *loop* das linhas 4 a 7 seleciona vértices x que estão no grafo de entrada G mas que ainda não estão em T, escolhe arestas ij de T de maneira que inserindo x entre i e j minimize $c(ix) + c(xj) - c(ij)$. Assim, os vértices inseridos em T tentam aumentar seu custo da menor maneira possível. Cada repetição do *loop* cria um novo *subtour* de acordo com o método de inserção descrito por Nicholson [11], até que todos os vértices de V estejam dentro do *tour* T.

Definimos $Tour(T, k)$ como um *subtour* obtido após a inserção do vértice k no *subtour* T através do método demonstrado no *loop* das linhas 4 a 7. Com isso, também definimos $c(Tour(T, k))$ como o custo de cada *substour* feito, o que implicitamente define que o custo do *tour* final $c(T)$ será $\sum_{i=1}^{n-1} c(T, k)$.

Lema 5 (. *1977-rose-cheap-insert] Dado um grafo de entrada $G = (V, E)$, um subtour T e um vértice k tal que k não está em T, então $c(Tour(T, k)) \leq 2c(kj)$, onde ij é a aresta de T que foi removida para a inserção de k.*

Demonstração. Quando T possui apenas um vértice, o resultado é óbvio.

Quando o T possuir mais que um vértice, sendo ij uma aresta de dentro de T e k o próximo vértice a ser inserido, temos que $c(Tour(T, k)) \leq c(ik) + c(kj) - c(ij)$, devido ao fato que a aresta ij é escolhida para minimizar a expressão à esquerda.

Como o grafo de entrada G é métrico, temos que $c(ik) - c(ij) \leq c(jk)$. As duas inequações, e sendo $c(jk) = c(kj)$, nos mostram que $c(Tour(T, k)) \leq 2c(kj)$. □

Lema 6 (. *1977-rose-cheap-insert] Dado um grafo de entrada $G = (V, E)$, e a solução T dada pelo algoritmo, então $c(T) \leq 2MST(G)$, onde $MST(G)$ é o custo da árvore geradora mínima de G, e $c(T)$ é o custo da solução T.*

Demonstração. Seja $(a_1, a_2, \ldots, a_n)$ a sequência dos vértices de G que indica a ordem que eles foram inseridos em T. Seja T' o *subtour* que contém apenas os vértices $(a_1, \ldots, a_\ell)$ para algum $\ell < n$.

Note que a $MST(G)$ terá sempre pelo menos uma aresta $e_i = a_i x_i$, sendo a_i pertencente a T' e x_i não. Com isso, a partir do Lema 5 temos então que $c(Tour(T, x_i)) \leq 2c(e_i)$, e somando para todo i, temos $\sum_{i=1}^{n-1} c(Tour(T, x_i)) \leq 2\sum_{i=1}^{n-1} c(e_i)$. Com isso, como todo vértice do grafo está em T, por definição temos que $c(T) \leq 2MST(G)$. □

Teorema 7 (. *1977-rose-cheap-insert] O algoritmo* TSP-CHEAPEST-INSERTION *é uma 2-aproximação.*

Demonstração. Seja T^* uma solução ótima do TSP, de custo OPT. Note que alguma aresta de T^* tem custo pelo menos $\frac{OPT}{n}$ (se todas tivessem custo menor do que $\frac{OPT}{n}$, teríamos uma contradição).

Assim, se tirarmos uma aresta e da solução ótima T^*, temos que $MST(G) \leq (1 - \frac{1}{n})OPT$, o que mostra implicitamente que $MST(G) \leq OPT$.

Como mostrado no Lema 6, $c(T) \leq 2MST(G)$, então, juntando as duas expressões temos $c(T) \leq (2 - \frac{2}{n})OPT < 2OPT$, de onde vemos que o algoritmo é uma 2-aproximação. □

O algoritmo de Christofides [1] é, até o momento, o algoritmo de aproximação com a melhor taxa de aproximação para o TSP métrico. Porém, existem outras variações do TSP, que possuem soluções por algoritmos de aproximação que valem ser citadas.

Uma dessas variações é conhecida como *graphic TSP*, onde nos é dado um grafo G' qualquer (não necessariamente completo), e um grafo completo G é construído sobre os mesmos vértices $V(G')$ tendo custo $c(uv)$ de uma aresta dado pelo número de arestas no menor caminho entre u e v em G'. Como sempre, deve-se encontrar um ciclo de custo mínimo que passe por todos os vértices de G mas que, em G', acabará podendo passar pelos mesmos vértices mais de uma vez. Para essa variação, Gharan, Saberi e Singh [4] mostram uma $(1.5 - \varepsilon)$-aproximação, Mömke e Svensson [9] mostram uma 1.461-aproximação, e Mucha [10] mostra uma 1.444-aproximação.

Existe também uma variação do TSP chamada de *st-path TSP*, onde nos é dado um grafo completo $G = (V, E)$ com uma função de custo c nas arestas e dois vértices, $s, t \in V(G)$, e deve-se encontrar o caminho de custo mínimo que ligue s a t e passe por todos os outros vértices. Para essa variação, Hoogeveen [8] mostra uma 1.667-aproximação, An, Kleinberg e Shmoys [7] mostram uma 1.618-aproximação, Sebő [15] mostra um 1.6-aproximação, Vygen [19] mostra uma 1.599-aproximação, Gottschalk e Vygen [6] mostram uma 1.566-aproximação, Sebő e Zuylen [16] mostram uma 1.529-aproximação, Traub e Vygen [17] mostram uma $(1.5 + \varepsilon)$-aproximação sendo $\varepsilon > 0$, e, bem recentemente, Zenklusen [20] mostra uma 1.5-aproximação.

3.5 Problema da árvore de Steiner

Steiner tree, ou **árvore de Steiner**, é um problema NP-difícil [3], similar ao TSP, que tem como objetivo encontrar o caminho de custo mínimo que conecte um determinado conjunto de terminais, podendo passar por outros objetos fora desse conjunto, com o intuito de tornar o custo do caminho menor.

Formalmente, o problema recebe como entrada um grafo não orientado completo

$G = (V, E)$, cujos vértices são divididos em duas categorias: um conjunto R de vértices que queremos conectar e um conjunto $V \setminus R$, dos vértices chamados **vértices de Steiner**. Com isso, deseja-se encontrar uma árvore de custo mínimo em G que contenha todos os vértices de R, e eventuais vértices de Steiner que sejam necessários.

3.5.1 Abordagem com programação linear inteira

Seja x_{ij} uma variável cujo valor é 1 se a aresta $ij \in E(G)$ for escolhida para fazer parte da árvore ou é 0, caso contrário. Considere o seguinte programa linear inteiro:

$$\text{minimizar} \quad \sum_{j=1}^{n} \sum_{i=1}^{n} c_{ij} x_{ij} \tag{9}$$

$$\text{sujeito a} \quad \sum_{i=1}^{n} x_{ij} \geq 1 \qquad \forall j \in R \tag{10}$$

$$\sum_{i \in S} \sum_{j \notin S} x_{ij} \geq 1 \qquad \forall S \subset V, S \neq \emptyset \tag{11}$$

$$\sum_{i,j \in S} x_{ij} \leq |S| - 1 \qquad \forall S \subset V, S \neq \emptyset \tag{12}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \in R, i < j \tag{13}$$

Note que ele formaliza o problema da árvore de Steiner: a função de objetivo em (9) indica que queremos minimizar o custo das arestas escolhidas, a restrição (10) faz com que haja pelo menos uma aresta selecionada de cada um dos vértices exigidos, a restrição (11) garante que uma estrutura conexa seja formada, forçando com que haja sempre uma aresta conectando todos os subconjuntos de vértices, e a restrição (12) garantirá que não sejam formados ciclos, garantindo a formação de uma árvore. Essa última restrição é equivalente à presente no programa linear do TSP.

Note que existem $O(2^{|V(G)|})$ restrições dos tipos (12) e (11), uma para cada possível subconjunto de vértices, assim como na formulação do TSP, fazendo que essa formulação seja inviável conforme o tamanho do grafo aumenta.

3.5.2 Abordagem com algoritmo de aproximação

Utilizaremos uma versão particular do problema, a **árvore de Steiner métrica**, onde assumimos que o grafo de entrada G é métrico, assim como no TSP métrico.movemos eventuais arestas repet

Algoritmo baseado em MST. O algoritmo a seguir recebe o conjunto de vértices exigidos R e a função c de custo nas arestas. A notação $G[R]$ indica o grafo induzido pelos vértices de R, isto é, o subgrafo de G cujo conjunto de vértices é R e contém apenas arestas cujos extremos são vértices de R.

Algoritmo 5: METRIC-ST

Entrada: $G = (V = R \cup X, E), c$

```
1 início
2 |  S ← ÁrvoreGeradoraMínima(G[R], c)
3 |  retorna S
4 fim
```

Lema 8 ([18])**.** *Dado um grafo $G = (V, E)$ e um conjunto de vértices exigidos R para a árvore de Steiner, então $MST(G[R]) \leq 2OPT(G)$, onde $MST(G[R])$ é o custo de uma árvore geradora mínima sobre os vértices de R, e $OPT(G)$ o custo de uma solução ótima da árvore de Steiner para G.*

Demonstração. Seja S^* uma solução ótima para a árvore de Steiner contendo todos os vértices exigidos, e possivelmente vértices de Steiner. Note que se duplicarmos as arestas de S^*, conseguiremos estabelecer uma trilha Euleriana F sobre elas de custo $2OPT(G)$. Agora, fazemos *shortcuts* sobre essa trilha para chegar a um ciclo Hamiltoniano H que contém apenas vértices de R. Por causa dos vértices eliminados nos *shortcuts*, vale que $c(H) \leq c(F)$. Logo, $c(H) \leq 2OPT(G)$.

Agora, se eliminarmos uma das arestas de H obteremos uma árvore geradora T que contém todos os vértices exigidos de R e, claramente, $c(T) \leq c(H)$. Como $MST(G[R]) \leq c(T)$, temos que $MST(G[R]) \leq 2OPT(G)$. □

Corolário 9. *O algoritmo METRIC-ST é uma 2-aproximação.*

Algoritmo para Steiner tree não métrico. O algoritmo a seguir recebe o conjunto de vértices exigidos R que estão presentes em um grafo G não métrico, e a função c de custo nas arestas.

Algoritmo 6: NOT-METRIC-ST

Entrada: $G = (V = R \cup X, E), c$

```
1 início
2 |  G' ← Complemento métrico de G com custos c'
3 |  T' ← ÁrvoreGeradoraMínima(G'[R], c)
4 |  T ← Árvore de menor custo com as arestas de T' em G
5 |  retorna T
6 fim
```

Na linha 2 cria-se um grafo G' que é completo e tem os mesmos vértices do grafo de entrada G. Sendo c' a função de custo das arestas de G', fazemos $c'(uv)$ igual

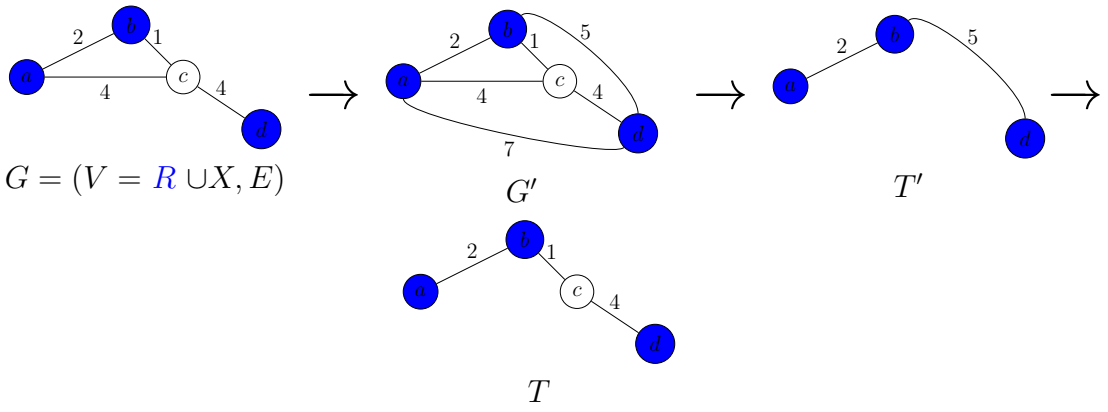

Figura 6: Exemplo de execução do algoritmo NOT-METRIC-ST.

ao custo do menor caminho entre u e v em G com a função de custo c. Na linha 3 o algoritmo chama uma função que retorna uma árvore geradora mínima de $G'[R]$ e armazena essa árvore em T'. Assim, na linha 4 cria-se uma árvore geradora T de G, de maneira que as conexões dessa árvores serão baseadas nas arestas de T', sendo que se ab é uma aresta de T', então deve haver um caminho em T que conecte a e b, podendo ter quantos vértices intermediários necessários. Devido à possível repetição, removemos eventuais arestas repetidas para gerar T. A Figura 6 dá um exemplo de como o algoritmo funciona.

Teorema 10. *O algoritmo* NOT-METRIC-ST *é uma* 2*-aproximação.*

Demonstração. Seja S uma solução ótima do ST no grafo G, não necessariamente completo, de custo $OPT(G)$, e seja S' uma solução ótima do ST no grafo G', que é o complemento métrico de G, de custo $OPT(G')$.

Note que o algoritmo retorna T como solução, e que $c(T) \leq c(T')$, uma vez que T' faz conexões entre todos os vértices de R e a transformação de T' em T pode conter vértices intermediários que, ao serem removidos, podem diminuir o custo. Como mostrado no Lema 8, $c(T') \leq 2OPT(G')$.

Dado que S também é uma solução para G', então temos que $OPT(G') \leq c'(S)$, e como G' é um complemento métrico de G, $c(S) = c'(S)$, fazendo que $OPT(G') \leq OPT(G)$. Juntando as expressões, temos $c(T) \leq c(T') \leq 2OPT(G') \leq 2OPT(G)$, de onde vemos que o algoritmo é uma 2-aproximação. □

Referências

[1] N. Christofides. Worst-Case Analysis of a New Heuristic for the Traveling Salesman Problem. Technical report, Carnegie Mellon University, 02 1976.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Algoritmos*. Campus Ltda., second edition, 2002.

[3] S. Eades, P.and Whitesides. The realization problem for euclidean minimum spanning trees is np-hard. *Algorithmica*, pages 60–82, 1996.

[4] Shayan Oveis Gharan, Amin Saberi, and Mohit Singh. A randomized rounding approach to the traveling salesman problem. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 550–559. IEEE, 2011.

[5] M. C. Goldbarg and H. P. L. Luna. *Otimização combinatória e programação linear: modelos e algoritmos*, pages 332–333. Elsevier Ltda, second edition, 2005.

[6] Corinna Gottschalk and Jens Vygen. Better s-t-tours by gao trees. In *Proceedings of 18th International Conference on Integer Programming and Combinatorial Optimization (IPCO)*, pages 126–137, 2016.

[7] R. Kleinberg H.-C. An and D. B. Shmoys. Improving christofides' algorithm for the s-t path tsp. *Journal of the ACM*, pages 1–34, 2015.

[8] J.A. Hoogeveen. Analysis of christofides' heuristic: some paths are more difficult than cycles. *Operations Research Letters*, pages 291–295, 1991.

[9] Tobias Mömke and Ola Svensson. Approximating graphic tsp by matchings. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 560–569. IEEE, 2011.

[10] M. Murcha. $\frac{13}{9}$-approximation for graphic tsp. *Theory of Computing Systems*, pages 640–657, 2014.

[11] T. A. J. Nicholson. *A sequential method for discrete optimization problems and its application to the assignment, travelling salesman, and three machine scheduling problems*. J. Inst. Math Appl., 1967.

[12] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *Society for Industrial and Applied Mathematics*, 1977.

[13] S. Sahni and T. Gonzalez. P-Complete Approximation Problems. *Journal of the ACM*, 23(3):555–565, 1976.

[14] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[15] András Sebő. Eight-fifth approximation for the path tsp. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 362–374. Springer, 2013.

[16] András Sebo and Anke Van Zuylen. The salesman's improved paths: A 3/2+ 1/34 approximation. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 118–127. IEEE, 2016.

[17] Vera Traub and Jens Vygen. Approaching $\frac{3}{2}$ for the st-path tsp. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1854–1864. Society for Industrial and Applied Mathematics, 2018.

[18] V. V. Vazirani. *Approximation Algorithms*, pages 28–29. Springer Science and Business Media, 2001.

[19] J. Vygen. Reassembling trees for the traveling salesman. *Journal on Discrete Mathematics*, page 875–894, 2016.

[20] Rico Zenklusen. A 1.5-approximation for path tsp. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '19, pages 1539–1549, Philadelphia, PA, USA, 2019. Society for Industrial and Applied Mathematics.