

Lorena Silva Sampaio, Samira Haddad

**Análise e Implementação de Algoritmos de Busca  
de uma r-Arborescência Inversa de Custo Mínimo  
em Grafos Dirigidos com Aplicação Didática  
Interativa**

Brasil

2025

Lorena Silva Sampaio, Samira Haddad

**Análise e Implementação de Algoritmos de Busca de uma  
r-Arborescência Inversa de Custo Mínimo em Grafos  
Dirigidos com Aplicação Didática Interativa**

Dissertação apresentada à Universidade Federal do ABC como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação.

Universidade Federal do ABC

Orientador: Prof. Dr. Mário Leston

Brasil  
2025

*Dedicatória (opcional).*

# Agradecimentos

Agradecimentos (opcional).

*“Epígrafe (opcional).”*

# Resumo

Este trabalho apresenta uma análise e implementação de algoritmos de busca de uma r-arborescência inversa de custo mínimo em grafos dirigidos com aplicação didática interativa.

**Palavras-chave:** Grafos. Arborescência. Algoritmos. Visualização.

# Abstract

This work presents an analysis and implementation of algorithms for finding a minimum cost inverse r-arborescence in directed graphs with interactive didactic application.

**Keywords:** Graphs. Arborescence. Algorithms. Visualization.

# Lista de ilustrações

<p>Figura 1 – A figura ilustra a escolha gulosa quando esta produz uma <math>r</math>-arborescência. Os arcos em azul são os escolhidos; os cinza são os demais arcos do digrafo. . . . .</p> <p>Figura 2 – Os arcos azuis são os da escolha gulosa. . . . .</p> <p>Figura 3 – Os arcos azuis são os da escolha gulosa. . . . .</p> <p>Figura 4 – Os arcos azuis são os da escolha gulosa. . . . .</p> <p>Figura 5 – Os arcos azuis são os da escolha gulosa. . . . .</p> <p>Figura 6 – O caminho simples maximal <math>P</math> inicia em <math>u</math> e termina em <math>v</math>. A porção <math>S</math> de <math>P</math> entre <math>u</math> e <math>w</math> é indicada pelo arco ondulado azul; o caminho <math>S \cdot u</math> é um ciclo. . . . .</p> <p>Figura 7 – O digrafo <math>D</math> com o ciclo <math>C = (v_1, v_2, v_3, v_1)</math>. Os arcos azuis representam os arcos do ciclo, os tracejados representam os demais arcos do digrafo. . . . .</p> <p>Figura 8 – digrafo com custos <math>\lambda</math>-reduzidos. Os arcos internos do ciclo <math>C</math> têm custo zero (em azul). Os arcos da raiz para o ciclo têm custos 1, 0 e 3 (tracejados). . . . .</p> <p>Figura 9 – digrafo <math>D'</math> após a contração do ciclo <math>C</math>. O supervértice <math>x_C</math> substitui todos os vértices do ciclo. Originalmente, havia três arcos paralelos de <math>r</math> para o ciclo: <math>(r, v_1)</math>, <math>(r, v_2)</math> e <math>(r, v_3)</math> com custos reduzidos 1, 0 e 3; mantemos apenas o de menor custo 0. Os arcos que saíam do ciclo agora saem de <math>x_C</math>: <math>(x_C, u)</math> com custo 0 e <math>(x_C, w)</math> com custo 0. Note que havia dois arcos de vértices do ciclo para <math>u</math>; mantemos apenas o de menor custo. . . . .</p> <p>Figura 10 – Reexpansão da <math>r</math>-arborescência ótima <math>T'</math> em <math>D'</math> para obter a <math>r</math>-arborescência <math>T</math> em <math>D</math> . . . . .</p> <p>Figura 11 – Reexpansão da <math>r</math>-arborescência ótima <math>T'</math> em <math>D'</math> para obter a <math>r</math>-arborescência <math>T</math> em <math>D</math>. Os arcos selecionados em verde fazem parte de <math>T</math>. . . . .</p> <p>Figura 12 – digrafo <math>D</math> com custos originais. O ciclo <math>C = (v_1, v_2, v_3, v_1)</math> tem arcos com custos 5, 5 e 2. Existem dois arcos da raiz para o ciclo, ambos com custo 5: <math>(r, v_1)</math> e <math>(r, v_3)</math>. . . . .</p> <p>Figura 13 – digrafo <math>D</math> com custos <math>\lambda</math>-reduzidos. Todos os arcos do ciclo custo mínimo agora custo zero. . . . .</p> <p>Figura 14 – Arborescência <math>T'</math> no digrafo contraído <math>D'</math>. O arco <math>(r, x_C)</math> pode corresponder ou ao arco <math>(r, v_1)</math> ou ao <math>(r, v_3)</math> em <math>D</math> e o arco <math>(x_C, u)</math> corresponde ao arco normalizado <math>(v_2, u)</math> em <math>D</math>. . . . .</p>	<p>12</p> <p>12</p> <p>12</p> <p>13</p> <p>14</p> <p>15</p> <p>16</p> <p>16</p> <p>16</p> <p>16</p> <p>17</p> <p>17</p> <p>17</p> <p>18</p> <p>18</p>
---	---

Figura 15 – Duas  $r$ -arborescências ótimas distintas em  $D$  com custos  $c_\lambda$ -reduzidos.

$T_1$  usa o arco  $(r, v_1)$  e os arcos do ciclo  $(v_1, v_2)$  e  $(v_2, v_3)$ .  $T_2$  usa o arco  $(r, v_3)$  e os arcos do ciclo  $(v_3, v_1)$  e  $(v_1, v_2)$ . Ambas incluem o arco  $(v_2, u)$  e têm custo total zero. . . . .

19

Figura 16 – Exemplo de normalização de custos reduzidos. À esquerda, vértice  $v$  com três arcos de entrada (pesos 5, 3 e 7). À direita, após aplicar `reduce_weights(D, v)`: o menor peso  $y(v) = 3$  é subtraído de todas as entradas, resultando em custos reduzidos 2, 0 e 4. O arco  $(u_2, v)$  (em vermelho) tem custo zero e será selecionado para  $A_0$ . . . . .

26

Figura 17 – Exemplo de construção de  $A_0$  a partir de um digrafo normalizado. À esquerda, o digrafo  $D$  após normalização, onde cada vértice não-raiz possui ao menos um arco de entrada com custo zero (em vermelho). À direita, o afo  $A_0$  resultante contém apenas os arcos de custo zero selecionados, um por vértice. Note que  $A_0$  pode conter ciclos (como  $\{v_1, v_2\}$ ) que serão tratados nas etapas subsequentes. . . . .

28

Figura 18 – Exemplo de detecção de ciclo em  $A_0$ . À esquerda, o subdigrafo  $A_0$  contém um ciclo formado pelos vértices  $\{v_2, v_3, v_4\}$  (destacados em amarelo). A DFS percorre o digrafo e detecta o ciclo ao encontrar o arco  $(v_4, v_2)$ , onde  $v_2$  já está na pilha de recursão. À direita, a função devolve uma cópia do subdigrafo induzido pelos vértices do ciclo, contendo apenas os três vértices e os três arcos que formam o ciclo. .

30

Figura 19 – Exemplo de contração de ciclo. À esquerda, digrafo original  $D$  com ciclo  $C = \{v_2, v_3, v_4\}$  (em amarelo). Vértices externos  $r$ ,  $v_1$  e  $v_5$  têm arcos conectando ao ciclo:  $r$  envia arco para  $v_2$  (peso 2) e  $v_4$  (peso 5);  $v_4$  envia arco para  $v_5$  (peso 1). À direita, após a contração: o ciclo é substituído pelo supervértice  $x_C$  (vermelho). Os arcos de entrada são redirecionados:  $(r, x_C)$  recebe peso 2 (menor entre 2 e 5). O arco de saída  $(x_C, v_5)$  mantém peso 1. Os dicionários `in_to_cycle` e `out_from_cycle` armazenam os mapeamentos originais para posterior reexpansão. . . . .

33

Figura 20 – Remoção de arco interno durante reexpansão. À esquerda, ciclo  $C = \{v_2, v_3, v_4\}$  após adicionar arco externo  $(u, v_2)$  vindouro da arborescência  $T'$ : o vértice  $v_2$  tem grau de entrada 2 (arco externo vermelho de  $u$  e arco interno do ciclo vindo de  $v_4$ ), violando a propriedade de arborescência. À direita, após remover o arco interno  $(v_4, v_2)$ : o vértice  $v_2$  passa a ter grau de entrada 1, o ciclo é "quebrado" no ponto de entrada, transformando-se em um caminho que se integra corretamente à estrutura de árvore. O arco removido é mostrado tracejado em cinza. .

35



# Sumário

<b>1</b>	<b>ALGORITMO DE CHU–LIU–EDMONDS . . . . .</b>	<b>11</b>
1.1	O algoritmo . . . . .	11
1.2	Descrição do algoritmo . . . . .	20
1.2.1	Corretude . . . . .	21
1.2.2	Complexidade . . . . .	22
1.3	Implementação em Python . . . . .	22
1.3.1	Representação de digrafos e detecção de ciclos . . . . .	23
1.3.2	Remoção de arcos que entram na raiz: . . . . .	24
1.3.3	Redução de custos por vértice (normalização): . . . . .	25
1.3.4	Construção de $A_0$ : . . . . .	27
1.3.5	Detecção de ciclo: . . . . .	28
1.3.6	Contração de ciclo: . . . . .	30
1.3.7	Remoção de arco interno: . . . . .	34
1.3.8	Procedimento principal (recursivo): . . . . .	35
1.3.9	Correspondência entre teoria e implementação . . . . .	41
1.3.10	Transição para a abordagem primal-dual . . . . .	43
	<b>REFERÊNCIAS . . . . .</b>	<b>44</b>
	<b>ANEXOS</b>	<b>45</b>
	<b>ANEXO A – ANEXO A . . . . .</b>	<b>46</b>

# 1 Algoritmo de Chu–Liu–Edmonds

Neste capítulo, apresentaremos o algoritmo de Chu–Liu–Edmonds, que determina uma arborescência de custo mínimo em um digrafo ponderado. O algoritmo baseia-se em duas operações fundamentais: (i) a redução gulosa dos custos dos arcos e (ii) a contração de ciclos, de modo a resolver recursivamente uma instância menor do problema e, em seguida, estender a solução para o problema original. O propósito deste capítulo é fornecer uma descrição precisa tanto do algoritmo quanto da implementação desenvolvida neste trabalho.

## 1.1 O algoritmo

O algoritmo de Chu–Liu–Edmonds recebe uma tripla  $(D, c, r)$ , em que  $D = (V, A)$  é um digrafo,  $c: A \rightarrow \mathbb{R}$  é uma função custo e  $r \in V$  é a raiz, sob a hipótese de que  $D$  admite ao menos uma  $r$ -arborescência. O algoritmo devolve uma  $r$ -arborescência  $c$ -mínima de  $D$ .

Para evitar repetir essa hipótese, introduzimos a seguinte definição. Uma tripla  $(D, c, r)$  é um  **$r$ -digrafo ponderado** se  $(D, c)$  é um digrafo ponderado,  $r$  é um vértice de  $D$ ,  $\delta^-(r) = \emptyset$  e  $D$  possui uma  $r$ -arborescência. Note que a hipótese  $\delta^-(r) = \emptyset$  é uma trivialidade, pois uma  $r$ -arborescência não contém nenhum arco que entra em  $r$  e, portanto, tais arcos podem ser eliminados de  $D$  sem nenhum prejuízo.

Vamos tecer algumas considerações para motivar o algoritmo.

### Caráter Guloso

Suponha doravante que  $(D, c, r)$  é um  $r$ -digrafo ponderado. O algoritmo tem um caráter guloso. Note que, se  $T$  é uma  $r$ -arborescência de  $D$ , então, para cada vértice  $v \neq r$ , existe exatamente um arco de  $T$  que entra em  $v$ . Isso sugere a seguinte escolha gulosa: para cada vértice  $v \neq r$ , selecione um arco  $a_v$  de custo mínimo que entra em  $v$  e forme o conjunto  $T := \{a_v : v \in V \setminus \{r\}\}$ .

Suponha que  $T$  é uma  $r$ -arborescência. Não é difícil verificar que  $T$  tem custo mínimo. De fato, seja  $F$  uma  $r$ -arborescência de  $D$ . Para cada vértice  $v \neq r$ , escreva  $b_v$  para o único arco de  $F$  que entra em  $v$ . Pela escolha gulosa,

$$c(a_v) \leq c(b_v) \quad \text{para todo } v \neq r.$$

Logo,

$$c(F) = \sum_{v \in V \setminus \{r\}} c(b_v) \geq \sum_{v \in V \setminus \{r\}} c(a_v) = c(T).$$



Figura 1 – A figura ilustra a escolha gulosa quando esta produz uma  $r$ -arborescência. Os arcos em azul são os escolhidos; os cinza são os demais arcos do digrafo.

Portanto,  $T$  é uma  $r$ -arborescência de custo mínimo.

A seguinte figura ilustra que podemos não ter tanta sorte.

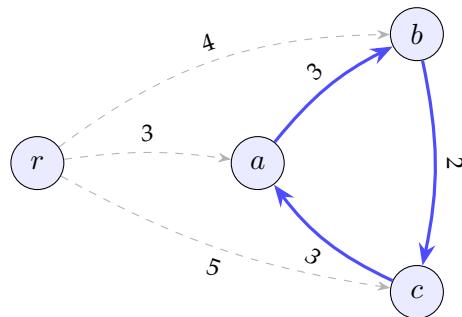


Figura 2 – Os arcos azuis são os da escolha gulosa.

Ora, se no lugar do arco  $(c, a)$  tivessemos escolhido o arco  $(r, a)$ , então  $r$ -arborescência resultante seria de custo mínimo.



Figura 3 – Os arcos azuis são os da escolha gulosa.

O exemplo acima sugere que devemos formar o subdigrafo  $H$  de  $D$  com  $V(H) = V(D)$  e

$$A(H) := \bigcup_{v \in V \setminus \{r\}} \arg \min \{ c(a) : a \in \delta^-(v) \}.$$

Ou seja, para cada  $v \neq r$  incluímos em  $H$  todos os arcos de custo mínimo que entram em  $v$ . Um argumento análogo ao anterior mostra que, se  $H$  contém uma  $r$ -arborescência, então ela é de custo mínimo.

Infelizmente, só isso não é suficiente, como mostra a próxima figura.



Figura 4 – Os arcos azuis são os da escolha gulosa.

O ideal, do ponto de vista algorítmico, é dispor de uma forma simples de identificar o subdigrafo  $H$ . Uma transformação nos custos fornece exatamente isso. Para tanto, introduzimos a noção de **custo  $q$ -reduzido**.

Seja  $q : V \setminus \{r\} \rightarrow \mathbb{R}$  (convencionamos  $q(r) = 0$ ). Definimos o **custo  $q$ -reduzido**  $c_q : A \rightarrow \mathbb{R}$  por

$$c_q(a) := c(a) - q(\text{head}(a)), \quad a \in A.$$

Para um conjunto  $X \subseteq V$ , escrevemos  $q(X) := \sum_{u \in X} q(u)$ .

A próxima proposição mostra que a transformação por custo  $q$ -reduzido preserva a optimalidade.

**Proposição 1.1.** *Para toda função  $q : V \setminus \{r\} \rightarrow \mathbb{R}$ , uma  $r$ -arborescência  $T$  é  $c$ -mínima em  $D$  se, e somente se,  $T$  é  $c_q$ -mínima em  $D$ .*

*Prova.* Seja  $F$  uma  $r$ -arborescência. Para cada  $u \in V \setminus \{r\}$ , seja  $a_u$  o único arco de  $F$  que entra em  $u$ . Então

$$\begin{aligned} c_q(F) &= \sum_{u \in V \setminus \{r\}} c_q(a_u) \\ &= \sum_{u \in V \setminus \{r\}} (c(a_u) - q(u)) \\ &= \sum_{u \in V \setminus \{r\}} c(a_u) - \sum_{u \in V \setminus \{r\}} q(u) \\ &= c(F) - q(V \setminus \{r\}). \end{aligned}$$

Assim, para quaisquer  $r$ -arborescências  $T$  e  $F$ ,

$$c(T) \leq c(F) \iff c'(T) = c(T) - q(V \setminus \{r\}) \leq c(F) - q(V \setminus \{r\}) = c_q(F),$$

o que prova a proposição.  $\square$

Para cada  $v \in V \setminus \{r\}$ , defina

$$\lambda(v) := \lambda_c(v) := \min\{c(a) : a \in \delta^-(v)\}.$$

Note que  $\lambda$  está bem definida uma vez que  $D$  possui uma  $r$ -arborescência e, portanto, existe ao menos um arco que entra em cada vértice diferente de  $r$ . Então, para todo  $v \in V \setminus \{r\}$ ,

$$\min\{c_\lambda(a) : a \in \delta^-(v)\} = 0,$$

isto é, precisamente os arcos de custo mínimo que entram em  $v$  passam a ter custo zero, e os demais ficam com custo positivo. Consequentemente, o subdigrafo  $H$  obtém-se simplesmente como o subdigrafo induzido pelos arcos de custo zero de  $c_\lambda$ :

$$V(H) = V(D) \quad \text{e} \quad A(H) = \{a \in A : c_\lambda(a) = 0\}.$$

A figura a seguir ilustra a redução de custos no digrafo da Figura 4.

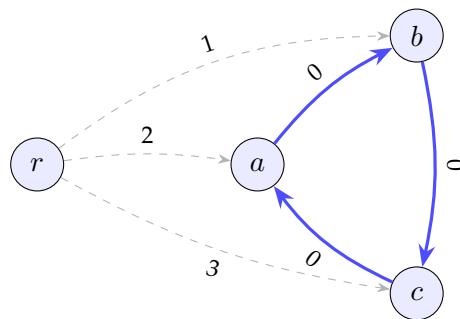


Figura 5 – Os arcos azuis são os da escolha gulosa.

Podemos agora retomar o caso no qual o subdigrafo gerador  $H$  de  $D$ , cujos arcos são aqueles em que o custo  $\lambda$ -reduzido é zero, não possui uma  $r$ -arborescência. Vamos mostrar que  $H$  possui um ciclo.

Seja  $v \neq r$  um vértice de  $V$  que *não* é alcançável a partir de  $r$  em  $H$ . Considere um caminho simples maximal<sup>1</sup> de  $H$  que termina em  $v$ . Seja  $u$  o início de  $P$ . Como  $v$  não é atingível a partir de  $r$ , temos que  $u \neq r$ . Logo, existe exatamente um arco, digamos  $wv$ , de  $H$  que entra em  $u$ . Pela maximalidade de  $P$ , o vértice  $w$  é um dos vértices de  $P$  (caso contrário,  $w \cdot P$  é um caminho simples, o que contradiz a escolha de  $P$ ). Como  $P$  é um caminho simples que começa em  $u$ , o vértice  $w$  aparece em  $P$  *após*  $u$ ; portanto,  $P$  contém um subcaminho  $S$  de  $u$  até  $w$ . Consequentemente,  $S \cdot u$  é um ciclo de  $H$ .

A solução consiste em *normalizar os custos por vértice*: para cada  $v \neq r$ , subtraímos de todo arco que entra em  $v$  o menor custo entre os arcos que chegam a  $v$ . Após esse

<sup>1</sup> Maximal aqui tem o seguinte sentido. Para cada vértice  $u$  de  $H$ , as sequências  $P \cdot u$  e  $u \cdot P$  não são caminhos simples.



Figura 6 – O caminho simples maximal  $P$  inicia em  $u$  e termina em  $v$ . A porção  $S$  de  $P$  entre  $u$  e  $w$  é indicada pelo arco ondulado azul; o caminho  $S \cdot u$  é um ciclo.

ajuste (custos reduzidos), cada  $v \neq r$  passa a ter ao menos um arco de custo reduzido zero. Se os arcos de custo zero forem acíclicos, já temos a  $r$ -arborescência ótima. Se formarem um ciclo  $C$ , *contraímos*  $C$  em um **supervértice**  $x_C$ , ajustamos os custos dos arcos externos e resolvemos recursivamente no digrafo menor.

A seguir, detalhamos essa operação de contração de ciclos.

### Contração de ciclos

Vamos agora formalizar a operação de contração de ciclos. Seja  $(D, c, r)$  um  $r$ -digrafo ponderado e seja  $C$  um ciclo dirigido de  $D$  tal que  $r \notin C$ . A **contração** de  $C$  consiste em formar um novo digrafo  $D' = (V', A')$  substituindo todos os vértices de  $C$  por um único **supervértice**  $x_C$  tal que  $x_C \notin V$ . Formalmente, o conjunto de vértices de  $D'$  é dado por

$$V' := (V \setminus C) \cup \{x_C\}.$$

O conjunto de arcos  $A'$  é construído a partir de  $A$  da seguinte forma: para cada arco  $a = (u, v) \in A$ , mantemos  $a$  inalterado em  $A'$  se ambos  $u$  e  $v$  estão fora de  $C$ ; descartamos  $a$  se ambos pertencem a  $C$ ; criamos um arco  $(u, x_C)$  se  $u \notin C$  e  $v \in C$ ; e criamos um arco  $(x_C, v)$  se  $u \in C$  e  $v \notin C$ .

Ajustamos os custos dos arcos que entram e saem do supervértice  $x_C$  em  $D'$  para refletir a contração do ciclo  $C$  da seguinte forma: para cada arco  $(u, v)$  com  $u \notin C$  e  $v \in C$ , o custo do arco contraído  $(u, x_C)$  é definido como  $c_\lambda(u, v)$ , onde  $\lambda(v) = \min\{c(a) : a \in \delta^-(v)\}$  é o custo mínimo de entrada em  $v$  e de forma semelhante, para cada arco  $(u, v)$  com  $u \in C$  e  $v \notin C$ , o custo do arco contraído  $(x_C, v)$  é definido como  $c_\lambda(u, v)$ , onde  $c_\lambda(u, v) = c(u, v) - \lambda(v)$  e  $\lambda(v) = \min\{c(a) : a \in \delta^-(v)\}$  é o custo mínimo de entrada em  $v$ .

Agora vamos ilustrar um exemplo de como essa contração é feita e os custos são ajustados.

Considere o digrafo  $D$  a seguir, com o ciclo  $C = (v_1, v_2, v_3, v_1)$ .

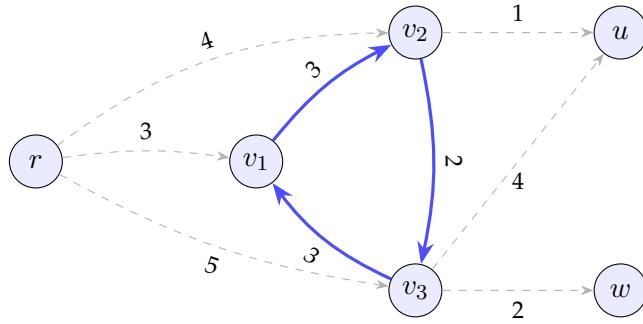


Figura 7 – O digrafo  $D$  com o ciclo  $C = (v_1, v_2, v_3, v_1)$ . Os arcos azuis representam os arcos do ciclo, os tracejados representam os demais arcos do digrafo.

Após a normalização dos custos, os arcos internos do ciclo passam a ter custo reduzido zero e os demais arcos são ajustados conforme a definição de custo  $\lambda$ -reduzido:

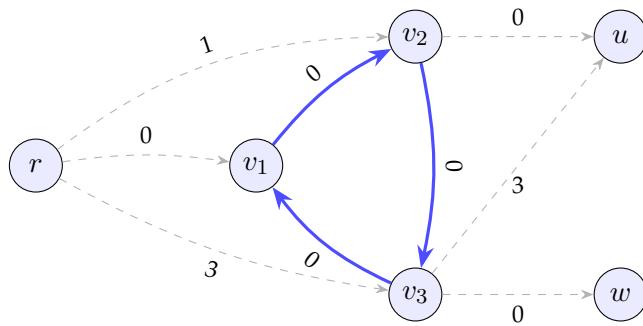


Figura 8 – digrafo com custos  $\lambda$ -reduzidos. Os arcos internos do ciclo  $C$  têm custo zero (em azul). Os arcos da raiz para o ciclo têm custos 1, 0 e 3 (tracejados).

Após a contração do ciclo  $C$ , obtemos o digrafo  $D'$  com o supervértice  $x_C$ .



Figura 9 – digrafo  $D'$  após a contração do ciclo  $C$ . O supervértice  $x_C$  substitui todos os vértices do ciclo. Originalmente, havia três arcos paralelos de  $r$  para o ciclo:  $(r, v_1)$ ,  $(r, v_2)$  e  $(r, v_3)$  com custos reduzidos 1, 0 e 3; mantemos apenas o de menor custo 0. Os arcos que saíram do ciclo agora saem de  $x_C$ :  $(x_C, u)$  com custo 0 e  $(x_C, w)$  com custo 0. Note que havia dois arcos de vértices do ciclo para  $u$ ; mantemos apenas o de menor custo.

Por definição não admitimos gerar arcos paralelos entre um mesmo par de vértices, mantemos apenas o arco de menor custo, conforme ilustrado do vértice  $r$  para

o supervértice  $x_C$  e de  $x_C$  para  $u$  e isso não afeta a optimidade, já que na reexpansão qualquer escolha entre arcos paralelos conduz à mesma solução ótima.

### Reexpansão de arborescências

Após resolver o problema no digrafo contraído  $D'$ , obtemos uma  $r$ -arborescência ótima  $T'$  em  $D'$ . Para reexpandir  $T'$  em uma  $r$ -arborescência  $T$  em  $D$ , substituímos o supervértice  $x_C$  pelo ciclo  $C$  e adicionamos os arcos do ciclo que formam a arborescência dentro de  $C$ . Especificamente, se o arco  $(u, x_C)$  em  $T'$  corresponde a um arco  $(u, v_i)$  em  $D$  (onde  $v_i \in C$ ), então incluímos esse arco em  $T$ . Em seguida, adicionamos os arcos do ciclo  $C$  que conectam os vértices de  $C$  de forma a manter a estrutura de arborescência. Note que, devemos escolher todos os arcos do ciclo  $C$  exceto aquele que entra em  $v_i$ , garantindo que cada vértice de  $C$  tenha grau de entrada igual a 1, exceto  $v_i$ .

Seguindo nosso exemplo anterior, ilustramos a reexpansão da  $r$ -arborescência, primeiramente adicionando novamente os vértices que pertenciam ao ciclo  $C$  e, em seguida, incluindo os arcos apropriados para formar a  $r$ -arborescência  $T$  em  $D$ :

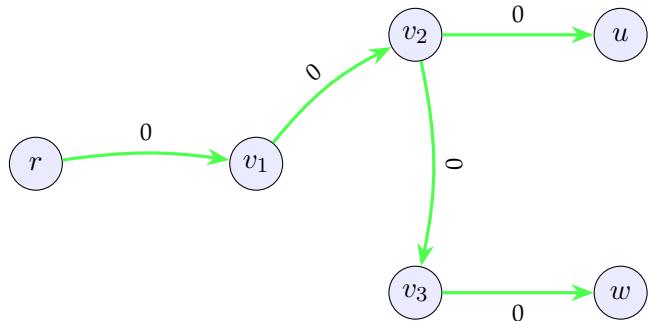


Figura 10 – Reexpansão da  $r$ -arborescência ótima  $T'$  em  $D'$  para obter a  $r$ -arborescência  $T$  em  $D$

Após isso reinserimos os pesos originais dos arcos. A figura a seguir ilustra esse processo:

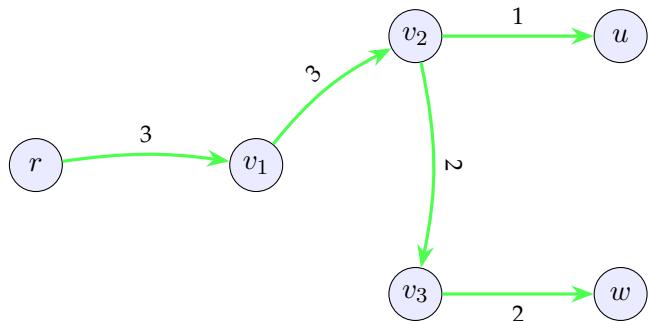


Figura 11 – Reexpansão da  $r$ -arborescência ótima  $T'$  em  $D'$  para obter a  $r$ -arborescência  $T$  em  $D$ . Os arcos selecionados em verde fazem parte de  $T$ .

É importante observar que, ao depender da forma com a qual extraímos a arborescência ótima de  $D'$  a partir do nosso digrafo original, podemos obter múltiplas arborescências ótimas em  $D$ , o exemplo a seguir ilustra essa situação:

Considere o digrafo a seguir com custos originais:



Figura 12 – digrafo  $D$  com custos originais. O ciclo  $C = (v_1, v_2, v_3, v_1)$  tem arcos com custos 5, 5 e 2. Existem dois arcos da raiz para o ciclo, ambos com custo 5:  $(r, v_1)$  e  $(r, v_3)$ .

Após a normalização dos custos (subtraindo  $\lambda(v_1) = 5$ ,  $\lambda(v_2) = 2$ ,  $\lambda(v_3) = 5$  e  $\lambda(u) = 4$ ), obtemos:



Figura 13 – digrafo  $D$  com custos  $\lambda$ -reduzidos. Todos os arcos do ciclo custo mínimo têm agora custo zero.

Após a contração do ciclo  $C$ , ambas as arborescências  $T_1$  e  $T_2$  mapeiam para a mesma arborescência  $T'$  no digrafo contraído  $D'$ :

#### Arborescência $T'$ em $D'$

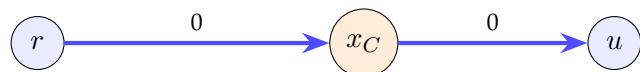


Figura 14 – Arborescência  $T'$  no digrafo contraído  $D'$ . O arco  $(r, x_C)$  pode corresponder ou ao arco  $(r, v_1)$  ou ao  $(r, v_3)$  em  $D$  e o arco  $(x_C, u)$  corresponde ao arco normalizado  $(v_2, u)$  em  $D$ .

No processo de reexpansão, existem duas  $r$ -arborescências ótimas distintas em  $D$ , ilustradas a seguir:



Figura 15 – Duas  $r$ -arborescências ótimas distintas em  $D$  com custos  $c_\lambda$ -reduzidos.  $T_1$  usa o arco  $(r, v_1)$  e os arcos do ciclo  $(v_1, v_2)$  e  $(v_2, v_3)$ .  $T_2$  usa o arco  $(r, v_3)$  e os arcos do ciclo  $(v_3, v_1)$  e  $(v_1, v_2)$ . Ambas incluem o arco  $(v_2, u)$  e têm custo total zero.

Assim, vemos que a correspondência entre as  $r$ -arborescências de  $D$  e  $D'$  não é bijetiva, pois duas arborescências distintas em  $D$  podem corresponder à mesma arborescência em  $D'$ . Isso ocorre porque ambos os arcos  $(r, v_1)$  e  $(r, v_3)$  têm o mesmo custo  $\lambda$ -reduzido (zero) e ambos entram no ciclo  $C$ ; após a contração, ambos são representados pelo único arco  $(r, x_C)$  no digrafo contraído.

A seguir, apresentamos a proposição que estabelece a correspondência entre as  $r$ -arborescências de  $D$  e  $D'$  após a contração do ciclo  $C$ .

**Proposição 1.2.** Seja  $C$  um ciclo dirigido em  $D$  com  $r \notin C$ , e seja  $D'$  o digrafo obtido pela contração de  $C$ . Para cada vértice  $v \in C$ , seja  $a_v$  o único arco de  $C$  que entra em  $v$ , e suponha que  $c_\lambda(a_v) = 0$  para todo  $v \in C$ , onde  $\lambda(v) = \min\{c(a) : a \in \delta^-(v)\}$ . Defina os custos  $c' : A' \rightarrow \mathbb{R}$  por

$$c'(a') := \begin{cases} c_\lambda(a) & \text{se } a' = a \text{ e } a \text{ não envolve } C, \\ c_\lambda(u, w) & \text{se } a' = (u, x_C) \text{ corresponde a } (u, w) \text{ com } w \in C, \\ c_\lambda(u, v) & \text{se } a' = (x_C, v) \text{ corresponde a } (u, v) \text{ com } u \in C. \end{cases}$$

Então existe uma correspondência bijetiva entre as  $r$ -arborescências de  $D'$  com custos  $c'$  e as  $r$ -arborescências de  $D$  com custos  $c_\lambda$  que contêm exatamente um arco entrando em  $C$ . Note que, em geral, podem haver múltiplas arborescências ótimas de  $D$  que são mapeadas para uma mesma arborescência ótima de  $D'$ , ou seja, a correspondência entre arborescências ótimas pode não ser bijetiva.

*Prova.* Seja  $T'$  uma  $r$ -arborescência de  $D'$ . Como  $x_C$  é um vértice de  $D'$  e  $x_C \neq r$ , existe exatamente um arco de  $T'$  que entra em  $x_C$ . Seja  $(u, x_C)$  esse arco. No digrafo original  $D$ , o arco  $(u, x_C)$  corresponde a um arco  $(u, w)$  para algum  $w \in C$ .

Definimos  $T \subseteq A$  da seguinte forma: para cada arco  $a' \in T'$  que não envolve  $x_C$ , incluímos o arco correspondente  $a \in A$  em  $T$ ; para o arco  $(u, x_C) \in T'$ , incluímos

$(u, w) \in A$  em  $T$ ; e incluímos todos os arcos de  $C$ , com exceção do arco  $a_w$  que entra em  $w$ .

Afirmamos que  $T$  é uma  $r$ -arborescência de  $D$ . De fato, para cada vértice  $v \in V \setminus \{r\}$ , se  $v \notin C$ , então  $v \in V'$  e há exatamente um arco de  $T'$  entrando em  $v$ , logo há exatamente um arco de  $T$  entrando em  $v$ . Se  $v \in C$  e  $v \neq w$ , então o único arco de  $T$  entrando em  $v$  é o arco  $a_v$  do ciclo  $C$ . Finalmente, se  $v = w$ , o único arco de  $T$  entrando em  $w$  é precisamente  $(u, w)$ .

Além disso, como  $T'$  é acíclico em  $D'$  e os arcos do ciclo  $C$  formam um caminho de  $w$  até seus predecessores em  $C$  (exceto o arco removido  $a_w$ ), o conjunto  $T$  permanece acíclico. Portanto,  $T$  é uma  $r$ -arborescência de  $D$ .

Reciprocamente, seja  $T$  uma  $r$ -arborescência de  $D$  que contém exatamente um arco entrando em  $C$ , digamos  $(u, w)$  com  $u \notin C$  e  $w \in C$ . Definimos  $T' \subseteq A'$  mantendo cada arco de  $T$  que não envolve vértices de  $C$ , substituindo o arco  $(u, w)$  por  $(u, x_C)$ , e descartando os arcos internos de  $C$  presentes em  $T$ . É direto verificar que  $T'$  é uma  $r$ -arborescência de  $D'$  e que essa correspondência é bijetiva.

Finalmente, como todos os arcos  $a_v$  do ciclo  $C$  têm custo  $c_\lambda$ -reduzido zero, temos

$$c_\lambda(T) = \sum_{a \in T \setminus C} c_\lambda(a) + c_\lambda(u, w) = c'(T'),$$

o que estabelece a correspondência entre custos.  $\square$

Essa proposição justifica a estratégia recursiva do algoritmo: resolver o problema no digrafo contraído  $D'$  com custos ajustados  $c'$  e, em seguida, expandir a solução para o digrafo original  $D$ .

A seguir, apresentamos a implementação completa do algoritmo de Chu–Liu e Edmonds para encontrar uma  $r$ -arborescência de custo mínimo em um  $r$ -digrafo ponderado  $(D, c, r)$ .

## 1.2 Descrição do algoritmo

A seguir apresentamos uma descrição formal do algoritmo de Chu–Liu/Edmonds. Detalhes de implementação serão discutidos na próxima seção.

### Algoritmo 1.1: Chu–Liu/Edmonds (visão operacional)

Entrada: digrafo  $D = (V, A)$ , custos  $c : A \rightarrow \mathbb{R}_{\geq 0}$ , raiz  $r$ .<sup>a</sup>

1. Para cada  $v \neq r$ , escolha  $a_v \in \operatorname{argmin}_{(u,v) \in A} c(u, v)$ . Defina  $y(v) := c(a_v)$  e  $A_0 := \{a_v : v \neq r\}$ .

2. Se  $(V, A_0)$  é acíclico, devolva  $A_0$ . Por (KLEINBERG; TARDOS, 2006, Obs. 4.36), trata-se de uma r-arborescência de custo mínimo.
3. Caso contrário, seja  $C$  um ciclo dirigido de  $A_0$  (com  $r \notin C$ ). **Contração:** contraia  $C$  em um supervértice  $x_C$  e defina custos  $c'$  por

$$\begin{aligned} c'(u, x_C) &:= c(u, w) - y(w) = c(u, w) - c(a_w) && \text{para } u \notin C, w \in C, \\ c'(x_C, v) &:= c(w, v) && \text{para } w \in C, v \notin C, \end{aligned}$$

descartando laços em  $x_C$  e permitindo paralelos. Denote o digrafo contraído por  $D' = (V', A')$ .

4. **Recursão:** compute uma r-arborescência ótima  $T'$  de  $D'$  com custos  $c'$ .
5. **Expansão:** seja  $(u, x_C) \in T'$  o único arco que entra em  $x_C$ . No digrafo original, ele corresponde a  $(u, w)$  com  $w \in C$ . Forme

$$T := (T' \setminus \{\text{arcos incidentes a } x_C\}) \cup \{(u, w)\} \cup ((A_0 \cap A(C)) \setminus \{a_w\}).$$

Então  $T$  tem grau de entrada 1 em cada  $v \neq r$ , é acíclico e tem o mesmo custo de  $T'$ ; logo, é uma r-arborescência ótima de  $D$  (KLEINBERG; TARDOS, 2006; SCHRIJVER, 2003, Sec. 4.9).

---

<sup>a</sup> Se algum  $v \neq r$  não possui arco de entrada, não existe r-arborescência.

### 1.2.1 Corretude

A corretude do algoritmo de Chu–Liu/Edmonds baseia-se em três pilares principais:

1. *Normalização por custos reduzidos:* para cada  $v \neq r$ , defina  $y(v) := \min\{c(u, v) : (u, v) \in A\}$  e  $c'(u, v) := c(u, v) - y(v)$ . Para qualquer r-arborescência  $T$ , vale

$$\sum_{a \in T} c'(a) = \sum_{a \in T} c(a) - \sum_{v \neq r} y(v),$$

pois há exatamente um arco de  $T$  entrando em cada  $v \neq r$ . O termo  $\sum_{v \neq r} y(v)$  é constante (independe de  $T$ ); assim, minimizar  $\sum c$  equivale a minimizar  $\sum c'$  (KLEINBERG; TARDOS, 2006, Obs. 4.37). Em particular, os arcos  $a_v$  de menor custo que entram em  $v$  têm custo reduzido zero e formam  $A_0$ .

2. *Caso acíclico:* se  $(V, A_0)$  é acíclico, então já é uma r-arborescência e, por realizar o mínimo custo de entrada em cada  $v \neq r$ , é ótima (KLEINBERG; TARDOS, 2006, Obs. 4.36).

3. *Caso com ciclo (contração/expansão)*: se  $A_0$  contém um ciclo dirigido  $C$ , todos os seus arcos têm custo reduzido zero.

Contraia  $C$  em  $x_C$  e ajuste apenas arcos que *entram* em  $C$ :  $c'(u, x_C) := c(u, w) - y(w) = c(u, w) - c(a_w)$ .

Resolva o problema no digrafo contraído  $D'$ , obtendo uma r-arborescência ótima  $T'$  sob  $c'$ . Na expansão, substitua o arco  $(u, x_C) \in T'$  pelo correspondente  $(u, w)$  (com  $w \in C$ ) e remova  $a_w$  de  $C$ .

Considerando que os arcos de  $C$  têm custo reduzido zero e  $c'(u, x_C) = c(u, w) - y(w)$ , a soma dos custos reduzidos é preservada na ida e na volta; logo,  $T'$  ótimo em  $D'$  mapeia para  $T$  ótimo em  $D$  para  $c'$ . Pela equivalência entre  $c$  e  $c'$ ,  $T$  também é ótimo para  $c$ . Repetindo o argumento a cada contração, obtemos a corretude por indução ([KLEINBERG; TARDOS, 2006](#); [SCHRIJVER, 2003](#), Sec. 4.9).

Em termos intuitivos,  $y$  funciona como um potencial nos vértices: torna “apertados” (custo reduzido zero) os candidatos corretos; ciclos de arcos apertados podem ser contraídos sem perder optimidade.

### 1.2.2 Complexidade

Na implementação direta, selecionar os  $a_v$ , detectar/contrair ciclos e atualizar estruturas custa  $O(m)$  por nível; como o número de vértices decresce a cada contração, temos no máximo  $O(n)$  níveis e tempo total  $O(mn)$ , com  $n = |V|$ ,  $m = |A|$ .

O uso de memória é  $O(m + n)$ , incluindo mapeamentos de contração/expansão e as filas de prioridade dos arcos de entrada. A implementação a seguir adota a versão  $O(mn)$  por simplicidade e está disponível no repositório do projeto ([<https://github.com/lorenypsum/GraphVisualizer>](https://github.com/lorenypsum/GraphVisualizer)).

## 1.3 Implementação em Python

Esta seção descreve a implementação do algoritmo de Chu–Liu–Edmonds em Python, estruturada para refletir com precisão as etapas formais discutidas anteriormente. Cada operação fundamental — normalização dos custos, construção do subdigrafo gerador, contração de ciclos e reexpansão — é traduzida em procedimentos sobre digrafos orientados, utilizando a biblioteca `networkx`.

A entrada consiste em um digrafo orientado  $D = (V, A)$ , com custos dos arcos registrados no atributo “`w`”, e uma raiz  $r \in V$ . As hipóteses adotadas são: (i) o digrafo é conexo a partir de  $r$ , isto é, todo vértice  $v \neq r$  é alcançável a partir da raiz; (ii) para todo subconjunto  $X \subseteq V \setminus \{r\}$ , existe ao menos um arco entrando em  $X$  (condições de Edmonds, cf. ([SCHRIJVER, 2003](#))); e (iii) todos os custos são não negativos.

A saída é um subdigrafo  $T$  de  $D$  com  $|A_T| = |V| - 1$  arcos, tal que cada vértice  $v \neq r$  possui grau de entrada igual a 1, todos os vértices são alcançáveis a partir de  $r$ , e o custo total  $\sum_{a \in A_T} c(a)$  é mínimo.

Por limitações da representação com `networkx.DiGraph`, a implementação elimina arcos paralelos durante a contração de ciclos.

A estrutura do código é modular: funções auxiliares tratam cada etapa do algoritmo — normalização dos custos, detecção e contração de ciclos, construção do subdigrafo gerador e reexpansão da solução. Todas operam sobre objetos `nx.DiGraph` e são coordenadas por uma função principal que gerencia o fluxo recursivo. As subseções seguintes detalham cada função auxiliar, abordando lógica, parâmetros, saídas e complexidade.

### 1.3.1 Representação de digrafos e detecção de ciclos

A implementação utiliza a biblioteca NetworkX<sup>2</sup>, especificamente a classe `nx.DiGraph` para representar digrafos  $D = (V, A)$ . Internamente, usa dicionários aninhados do Python para armazenar vértices, arcos e atributos, garantindo operações eficientes: adicionar/remover arco  $O(1)$  amortizado, iterar vizinhos  $O(\deg(u))$ , percorrer todos os arcos  $O(m)$ .

#### Métodos da API NetworkX

Os métodos da API NetworkX utilizados na implementação dividem-se em três categorias funcionais, cada uma correspondendo a uma fase específica do algoritmo:

#### Consulta de estrutura

- `D.nodes()`: devolve visão iterável sobre  $V$ , permitindo percorrer todos os vértices.
- `D.in_edges(v, data="w")`: devolve arcos entrantes em  $v$  com pesos, produzindo tuplas  $(u, v, w)$ .
- `D.out_edges(u, data="w")`: devolve arcos saíntes de  $u$  com pesos, análogo a `in_edges`.
- `D[u][v]["w"]`: acessa diretamente o peso do arco  $(u, v)$  para leitura ou modificação.

---

<sup>2</sup> NetworkX é uma biblioteca Python para criação, manipulação e estudo de redes. Disponível em <https://networkx.org/>.

## Modificação de estrutura

- `D.add_edge(u, v, w=peso)`: adiciona arco  $(u, v)$  com peso especificado, criando vértices automaticamente se não existirem.
- `D.remove_edges_from(edges)`: remove múltiplos arcos em lote.
- `D.remove_nodes_from(nodes)`: remove vértices e todos os seus arcos incidentes.

### 1.3.2 Remoção de arcos que entram na raiz:

Escrevemos essa função como uma etapa de pré-processamento para garantir que a raiz  $r_0$  não possua arcos de entrada antes de iniciar o algoritmo principal.

A remoção é necessária porque, por definição, uma r-arborescência é uma arborescência enraizada em  $r_0$  onde todo vértice  $v \neq r_0$  deve ser alcançável a partir de  $r_0$ , mas a própria raiz não pode ter predecessores (grau de entrada zero). Se o digrafo original contiver arcos entrando em  $r_0$ , esses arcos violariam a definição de arborescência enraizada e poderiam criar ciclos envolvendo a raiz, o que tornaria impossível obter uma estrutura válida. Além disso, a presença de arcos entrando na raiz interfere na normalização: ao tentar normalizar custos de entrada para  $r_0$ , criariam custos reduzidos artificiais que não fazem sentido no contexto do problema, já que nenhuma solução válida pode incluir tais arcos.

A escolha de implementar esta operação e as demais funções auxiliares fora do escopo da execução principal segue princípios de design de software: (1) *modularidade*, encapsulando uma responsabilidade específica e bem definida (remover entradas na raiz) em uma unidade testável independente; (2) *reutilização*, permitindo que outras partes do código ou implementações alternativas possam chamar esta operação quando necessário sem duplicar lógica; (3) *clareza semântica*, dando um nome descriptivo (`remove_edges_to_r0`) que documenta a intenção da operação no ponto de chamada, tornando a função principal mais legível ao abstrair detalhes de implementação; e (4) *facilidade de teste*, possibilitando escrever testes unitários focados exclusivamente nesta operação de pré-processamento, verificando casos extremos (como grafos onde a raiz já não tem predecessores ou onde todos os arcos entram na raiz) sem precisar testar toda a complexidade do algoritmo recursivo.

Em detalhes, ela recebe como entrada um digrafo  $D$  (objeto `nx.DiGraph`) e o raiz  $r_0$ . A implementação armazena em uma lista todos os arcos que entram em  $r_0$  usando o método `in.edges` (linha 2). Aqui precisamos armazenar os arcos em uma lista porque o método `in.edges` devolve um iterador, que quando sofre remoção direta sofre um erro devido à modificação da estrutura durante a iteração.

Em seguida, na linha 3, remove todos os arcos usando o método `remove_edges_from` da biblioteca NetworkX, o qual recebe como parâmetro uma lista de tuplas representando arcos na forma  $(u, v)$  e remove cada uma delas do digrafo. O método da NetworkX itera sobre a lista fornecida e, para cada tupla  $(u, v)$ , remove o arco correspondente da estrutura interna de adjacência. A complexidade de `remove_edges_from` é  $O(k)$ , onde  $k$  é o número de arcos na lista de entrada, pois cada remoção individual tem custo  $O(1)$  em média devido ao uso de dicionários aninhados para armazenar arcos.

Por fim, a função devolve o digrafo  $D$  atualizado no próprio objeto com todos os arcos de entrada em  $r_0$  removidos (linha 4). A complexidade total da função é  $O(\deg^-(r_0))$ , pois a operação coleta e remove cada arco de entrada uma única vez.

#### Remoção de arcos que entram na raiz

*Remove todos os arcos que entram na raiz  $r_0$ , modificando  $D$  ao invés de criar uma cópia e devolve o digrafo atualizado.*

```
1 def remove_edges_to_r0(D: nx.DiGraph, r0: str):
2     in_edges = list(D.in_edges(r0))
3     D.remove_edges_from(in_edges)
4     return D
```

### 1.3.3 Redução de custos por vértice (normalização):

Criamos uma função auxiliar para realizar a redução de custos por vértice - essa operação é chamada de normalização e calcula  $y(v) = \min\{w(u, v)\}$  e substitui cada peso  $w(u, v)$  por  $w(u, v) - y(v)$ , garantindo que ao menos um arco de entrada tenha custo zero. Como cada  $r$ -arborescência possui exatamente um arco entrando em cada vértice não-raiz, a soma total dos valores  $y(v)$  subtraídos é constante para qualquer solução, preservando assim a ordem de otimalidade entre diferentes arborescências.

Recebe como entrada um digrafo  $D$  (objeto `nx.DiGraph`) e o vértice `node` a ser normalizado. A implementação armazena em uma variável `incoming_edges` todos os arcos de entrada de `node` com seus pesos usando o método `D.in_edges(node, data="w")`, que devolve uma lista de tuplas  $(u, node, w)$  (linha 2) (fazemos isso para evitar repetição de código e deixar o código mais claro). Em seguida, calcula-se o peso mínimo  $yv$  através de uma compreensão de gerador que extrai o terceiro elemento de cada tupla (linha 3) e, para cada vértice  $u$ , se houver o atributo "w" subtrai  $yv$  do peso armazenado em `D[u][node]["w"]` (linha 7), caso contrário inicializa o peso como zero antes de subtrair (linha 6).

A função não devolve nenhum valor, pois a operação é realizada modificando

diretamente a estrutura: o digrafo  $D$  passado como parâmetro é modificado diretamente, e ao menos um arco de entrada de  $node$  terá custo reduzido zero após a execução. A complexidade é  $O(\deg^-(v))$ , pois cada operação percorre os arcos de entrada uma única vez.

### Redução de custos por vértice (normalização)

*Normaliza os pesos dos arcos que entram em node, subtraendo de cada uma o menor peso de entrada. Modifica o digrafo D no próprio objeto.*

```

1 def reduce_weights(D: nx.DiGraph, node: str):
2     incoming_edges = D.in_edges(node, data=True)
3     yv = min((data.get("w", 0) for _, _, data in incoming_edges))
4     for u, _, _ in incoming_edges:
5         if "w" not in D[u][node]:
6             D[u][node]["w"] = 0
7         D[u][node]["w"] -= yv

```

A Figura 16 ilustra o funcionamento da normalização:

**Antes:**  $y(v) = \min\{5, 3, 7\} = 3$  **Depois:** ao menos uma entrada tem custo 0

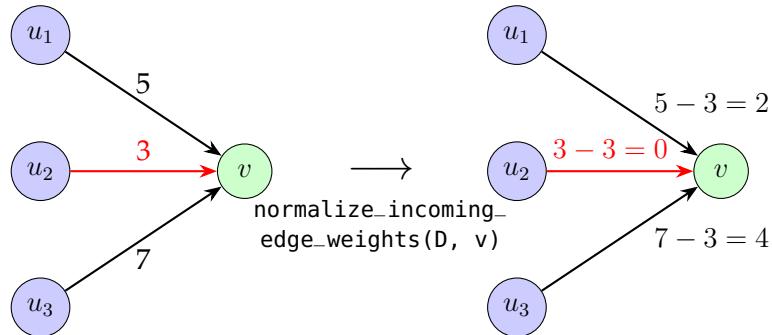


Figura 16 – Exemplo de normalização de custos reduzidos. À esquerda, vértice  $v$  com três arcos de entrada (pesos 5, 3 e 7). À direita, após aplicar  $reduce\_weights(D, v)$ : o menor peso  $y(v) = 3$  é subtraído de todas as entradas, resultando em custos reduzidos 2, 0 e 4. O arco  $(u_2, v)$  (em vermelho) tem custo zero e será selecionado para  $A_0$ .

Observe que as diferenças relativas são preservadas: o arco mais caro permanece 4 unidades acima da mais barata, e a intermediária mantém sua posição relativa.

Vale destacar que, quando o vértice recebe apenas um arco de entrada, trivialmente o custo desse arco é reduzido a zero.

Considerando que cada r-arborescência contém exatamente um arco entrando em cada vértice não-raiz, a soma  $\sum_{w \neq r} y(w)$  é constante para qualquer solução, garantindo

que a ordem de otimalidade seja preservada.

#### 1.3.4 Construção de $A_0$ :

Esta função constrói o subdigrafo  $A_0$  selecionando, para cada vértice  $v \neq r_0$ , um único arco de custo reduzido zero que entra em  $v$ .

Recebe como entrada um digrafo  $D$  e a raiz  $r_0$ . A implementação cria um novo digrafo vazio  $A\_zero$  (linha 2) em vez de modificar  $D$  diretamente; essa escolha de criar uma estrutura separada é fundamental porque  $A_0$  é um subdigrafo usado para detecção de ciclos, e preservar  $D$  inalterado permite que as operações subsequentes (como contração) trabalhem com o digrafo original completo, evitando perda de informação sobre arcos não selecionados que podem ser necessários após reexpansões.

Em seguida, para cada vértice  $v$  diferente de  $r_0$  (linha 3), utilizando o método  $D.nodes()$  para iterar sobre todos os vértices, se  $v$  for diferente de  $r_0$  (linha 4), obtém todos os arcos de entrada em  $v$  com seus pesos usando  $D.in_edges(v, data=True)$  (linha 5).

Em seguida, obtém o primeiro predecessor  $u$  cujo arco  $(u, v)$  tem peso zero, aramazenando-o na variável  $u$  (linha 6) utilizando uma compreensão de gerador combinada com `next`. A escolha de `next` com gerador em vez de uma busca exaustiva é eficiente porque interrompe a iteração assim que encontra o primeiro arco de custo zero, evitando processamento desnecessário dos arcos restantes (embora teoricamente todos os arcos de custo zero sejam equivalentes, na prática apenas um é necessário para  $A_0$ ). Finalmente, adiciona o arco  $(u, v)$  com peso zero a  $A\_zero$  (linha 7).

Então, devolve-se o digrafo  $A\_zero$  contendo exatamente um arco entrando em cada  $v \neq r_0$ , todos com custo reduzido zero. O digrafo original  $D$  não é modificado, preservando o estado para operações futuras. A complexidade é  $O(m)$ , onde  $m$  é o número de arcos, pois cada arco é considerado no máximo uma vez durante a iteração sobre todos os vértices: para cada um dos  $n - 1$  vértices não-raiz, examina-se seus arcos de entrada (totalizando no máximo  $m$  arcos ao longo de todas as iterações), e para cada vértice a busca por arco de peso zero é interrompida na primeira identificação, resultando em tempo linear no tamanho do digrafo.

#### Construção de $A$ zero

*Constrói o subdigrafo  $A_0$  a partir do digrafo  $D$ , selecionando para cada vértice (exceto a raiz  $r_0$ ) um arco de custo reduzido zero que entra nele.*

```
1 def get_Azero(D: nx.DiGraph, r0: str):
2     A_zero = nx.DiGraph()
```

```

3   for v in D.nodes():
4       if v != r0:
5           in_edges = D.in_edges(v, data=True)
6           u = next((u for u, _, data in in_edges if data.get("w") == 0))
7           A_zero.add_edge(u, v, w=0)
8   return A_zero

```

A Figura 17 ilustra a construção de  $A_0$ :

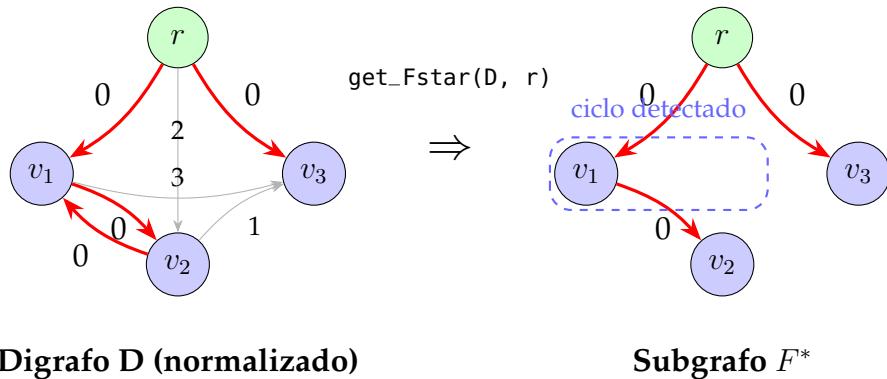


Figura 17 – Exemplo de construção de  $A_0$  a partir de um digrafo normalizado. À esquerda, o digrafo  $D$  após normalização, onde cada vértice não-raiz possui ao menos um arco de entrada com custo zero (em vermelho). À direita, o afó  $A_0$  resultante contém apenas os arcos de custo zero selecionados, um por vértice. Note que  $A_0$  pode conter ciclos (como  $\{v_1, v_2\}$ ) que serão tratados nas etapas subsequentes.

As funções de normalização por vértice e construção de  $A_0$  juntas implementam o passo 1 da descrição do algoritmo de Chu–Liu/Edmonds:

#### Passo 1 do Algoritmo de Chu–Liu/Edmonds

**Passo 1:** Para cada  $v \neq r$ , escolha  $a_v \in \arg \min_{(u,v) \in A} c(u, v)$ . Defina  $y(v) := c(a_v)$  e  $A_0 := \{a_v : v \neq r\}$ .

#### 1.3.5 Detecção de ciclo:

A detecção de ciclos é crucial, pois a presença de um ciclo em  $A_0$  indica que a escolha de arcos de custo reduzido zero não formou uma arborescência válida.

Logo, a função apresentada a seguir detecta a presença de um ciclo dirigido em  $A_0$  e devolve um subdigrafo que o contém; Não verificamos se o digrafo é acíclico, pois a função principal não deve sequer fazer essa verificação: se não houver ciclo, é porque uma arborecência já foi encontrada.

Recebe como entrada um digrafo  $A_{\text{zero}}$ . A função inicializa um conjunto vazio `nodes_in_cycle` (linha 2). O laço na linha 3 itera sobre os arcos devolvidos pela função `nx.find_cycle(A_zero, orientation="original")`, que utiliza uma busca em profundidade (DFS) para detectar ciclos dirigidos. Se um ciclo for encontrado, a função devolve um iterador sobre os arcos do ciclo, desempacotando cada uma na forma  $(u, v, \_)$  (ignorando o terceiro elemento com  $\_$ , que contém metadados de orientação), e na linha 4 para cada arco  $(u, v)$  adiciona ambos os vértices ao conjunto `nodes_in_cycle` (linha 4); a escolha de usar conjunto em vez de lista garante que cada vértice seja adicionado apenas uma vez mesmo que o ciclo tenha múltiplos arcos incidentes, e a operação de adição tem complexidade  $O(1)$ .

Após coletar todos os vértices do ciclo, constrói e devolve uma cópia do subgrafo induzido por eles (linha 7); a cópia é necessária porque o método `subgraph` devolve apenas uma visão dinâmica sobre o digrafo original.

No final, um subdigrafo contendo os vértices e arcos do ciclo detectado é devolvido. O digrafo original  $A_{\text{zero}}$  não é modificado. A complexidade é  $O(m)$ , onde  $m$  é o número de arcos, pois a DFS visita cada arco no máximo uma vez.

#### Detecção de ciclo dirigido em $A_0$

*Detecta um ciclo dirigido em  $A_0$  e devolve um subdigrafo contendo seus vértices e arcos, ou None se for acíclico.*

```

1 def find_cycle(A_zero: nx.DiGraph):
2     nodes_in_cycle = set()
3     for u, v, _ in nx.find_cycle(A_zero, orientation="original"):
4         nodes_in_cycle.update([u, v])
5     return A_zero.subgraph(nodes_in_cycle).copy()

```

A Figura 18 ilustra o processo de detecção de ciclo:

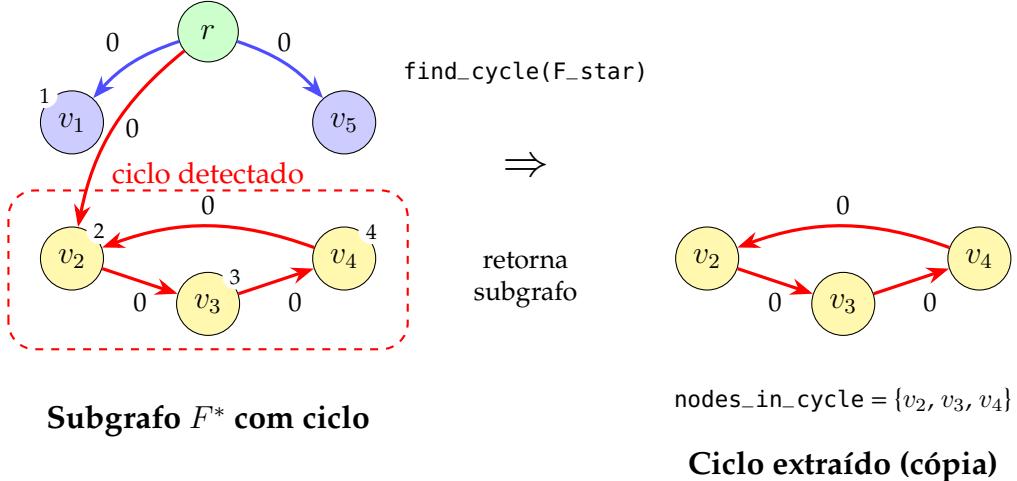


Figura 18 – Exemplo de detecção de ciclo em  $A_0$ . À esquerda, o subdigrafo  $A_0$  contém um ciclo formado pelos vértices  $\{v_2, v_3, v_4\}$  (destacados em amarelo). A DFS percorre o digrafo e detecta o ciclo ao encontrar o arco  $(v_4, v_2)$ , onde  $v_2$  já está na pilha de recursão. À direita, a função devolve uma cópia do subdigrafo induzido pelos vértices do ciclo, contendo apenas os três vértices e os três arcos que formam o ciclo.

Ao detectar um ciclo, o código avança para a etapa de contração executando operações concretas sobre o objeto Python: primeiro, coleta os vértices de  $C$  num conjunto para consultas em  $O(1)$ ; em seguida, para cada vértice externo  $u \notin C$  determina o arco de menor peso que vai de  $u$  para algum  $w \in C$  e guarda esse par em  $\text{exttint\_to\_cycle}[u]=(w,\text{weight})$ ; análogamente, para cada vértice externo  $v \notin C$  determina o arco de menor peso que vai de algum  $w \in C$  para  $v$  e guarda em  $\text{out\_from\_cycle}[v]=(w,\text{weight})$ . Depois de coletar estas informações (para evitar mutações durante iterações), o código cria no digrafo  $D$  arcos redirecionados  $(u, \text{label})$  e  $(\text{label}, v)$  com os pesos mínimos correspondentes. Por fim, a função devolve os dicionários  $\text{in\_to\_cycle}$  e  $\text{out\_from\_cycle}$ , que serão usados na reexpansão para reconstruir corretamente os arcos do ciclo original. Note que a substituição efetiva do ciclo pelo supervértice é realizada pelas operações de remoção/adaptação subsequentes no procedimento principal; aqui apenas são computadas e adicionadas as arestas de ligação e preservadas as informações necessárias para a reexpansão.

### 1.3.6 Contração de ciclo:

Escrevemos uma função responsável por contrair um ciclo dirigido  $C$  a um supervértice  $x_C$ , redirecionando arcos incidentes e ajustando custos segundo a regra de custos reduzidos. No final, a função devolve dois dicionários auxiliares com informações dos vértices que incidiam e ascendiam de  $C$ , essenciais para a etapa posterior de reexpansão da arborescência.

A função recebe como entrada um digrafo  $D$ , o ciclo  $C$  a ser contraído (que fora detectado pela função anterior) e parâmetro de rotulação do novo supervértice `label`. Inicialmente, coletam-se os vértices de  $C$  em um conjunto (linha 2) para permitir verificações de pertinência em tempo  $O(1)$ , essencial dado que essa operação é realizada repetidamente nos laços seguintes. Inicializa-se então `in_to_cycle` (linha 3), um dicionário que tem como chave vértices externos ao ciclo e cujo valor são tuplas  $(v, w)$ , onde  $v$  é o vértice do ciclo conectado a  $u$  e  $w$  é o peso do arco  $(u, v)$ ; essa estrutura preserva não apenas o peso mínimo, mas também o ponto exato de entrada no ciclo.

Para cada vértice  $u$  no digrafo  $D$  (linha 4), se  $u$  não pertence ao ciclo (linha 5), identifica-se o arco de menor peso que sai de  $u$  e entra em  $C$  (linhas 6–9) usando a função `min` combinada com uma compreensão de gerador: `((v, data.get("w", float("inf"))) for _, v, data in D.out_edges(u, data=True) if v in cycle_nodes)` a qual itera sobre todos os arcos que saem de  $u$ , desempacota cada arco na forma `(_, v, data)` (ignorando a origem com `_`, capturando o destino  $v$  e o peso  $data$ ), filtra apenas aquelas cujo destino  $v$  pertence ao ciclo, e produz tuplas  $(v, w)$ ; a função `min` (linha 6) então seleciona a tupla de menor peso usando `key=lambda x: x[1]` (linha 10) para comparar pelo segundo elemento (o peso), e devolve `None` se não houver arcos (linha 11). A escolha de selecionar apenas o arco de *menor peso* reflete a propriedade fundamental do algoritmo: qualquer solução ótima que conecta um vértice externo ao ciclo contraído usará necessariamente o arco de custo mínimo, pois todas as outras opções seriam subótimas. Se tal arco existir (linha 12), armazena em `in_to_cycle[u]` (linha 14).

Em seguida, a implementação itera sobre `in_to_cycle` usando o método `items()`, desempacotando cada entrada na forma  $(u, (v, w))$ , onde  $u$  é o vértice externo e  $(v, w)$  é a tupla com o vértice do ciclo e o peso (linha 14). Para cada par, cria um arco de  $u$  para `label` com peso  $w$ , efetivamente redirecionando os arcos de entrada para o supervértice (linha 15). A separação entre coleta (linhas 4–10) e criação (linhas 11–12) é necessária porque modificar o digrafo durante a iteração sobre seus vértices causaria comportamento indefinido; ao coletar primeiro todos os dados em estruturas auxiliares, garantimos que as modificações posteriores sejam seguras.

De forma análoga, constrói-se o dicionário `out_from_cycle` (linha 13) para mapear arcos que saem do ciclo.

Finalmente, dois dicionários são devolvidos: `in_to_cycle` mapeia vértices externos aos pontos de entrada no ciclo original, e `out_from_cycle` mapeia vértices externos aos pontos de saída. Esses dicionários são essenciais para a fase de reexpansão, onde será necessário determinar exatamente qual arco interno do ciclo deve ser removido a fim de manter um caminho que conecta todos os vértices. O digrafo  $D$  é modificado sem criar uma cópia: os vértices de  $C$  são removidos e substituídos por `label`. A escolha de modificação no próprio objeto (em vez de criar uma cópia) reduz significativamente

o uso de memória e o tempo de execução, especialmente em grafos grandes ou com múltiplos níveis de recursão, embora exija atenção cuidadosa para que informações originais sejam preservadas. A complexidade é  $O(m)$ , onde  $m$  é o número de arcos, pois cada arco incidente ao ciclo é processado uma vez: os laços nas linhas 4–10 e 14–19 examinam cada arco no máximo uma vez, e as operações de adição (linhas 11–12, 20–21) e remoção (linha 22) têm custo proporcional ao número de arcos afetados.

### Contração de ciclo

*Contraí o ciclo  $C$  em um supervértice `label`, redirecionando arcos incidentes e ajustando custos. Modifica  $D$  no próprio objeto e devolve dicionários para reexpansão.*

```

1 def contract_cycle(D: nx.DiGraph, C: nx.DiGraph, label: str):
2     cycle_nodes: set[str] = set(C.nodes())
3     in_to_cycle: dict[str, tuple[str, float]] = {}
4     for u in D.nodes:
5         if u not in cycle_nodes:
6             min_weight_edge_to_cycle = min(
7                 ((v, data.get("w", float("inf")))
8                  for _, v, data in D.out_edges(u, data=True)
9                  if v in cycle_nodes),
10                key=lambda x: x[1],
11                default=None,)
12         if min_weight_edge_to_cycle:
13             in_to_cycle[u] = min_weight_edge_to_cycle
14     for u, (v, w) in in_to_cycle.items():
15         D.add_edge(u, label, w=w)
16     out_from_cycle: dict[str, tuple[str, float]] = {}
17     for v in D.nodes:
18         if v not in cycle_nodes:
19             min_weight_edge_from_cycle = min(
20                 ((u2, data.get("w", float("inf")))
21                  for u2, _, data in D.in_edges(v, data=True)
22                  if u2 in cycle_nodes),
23                key=lambda x: x[1],
24                default=None,)
25         if min_weight_edge_from_cycle:
26             out_from_cycle[v] = min_weight_edge_from_cycle
27     for v, (u, w) in out_from_cycle.items():
28         D.add_edge(label, v, w=w)
29     return in_to_cycle, out_from_cycle

```

A Figura 19 ilustra o processo de contração de ciclo:

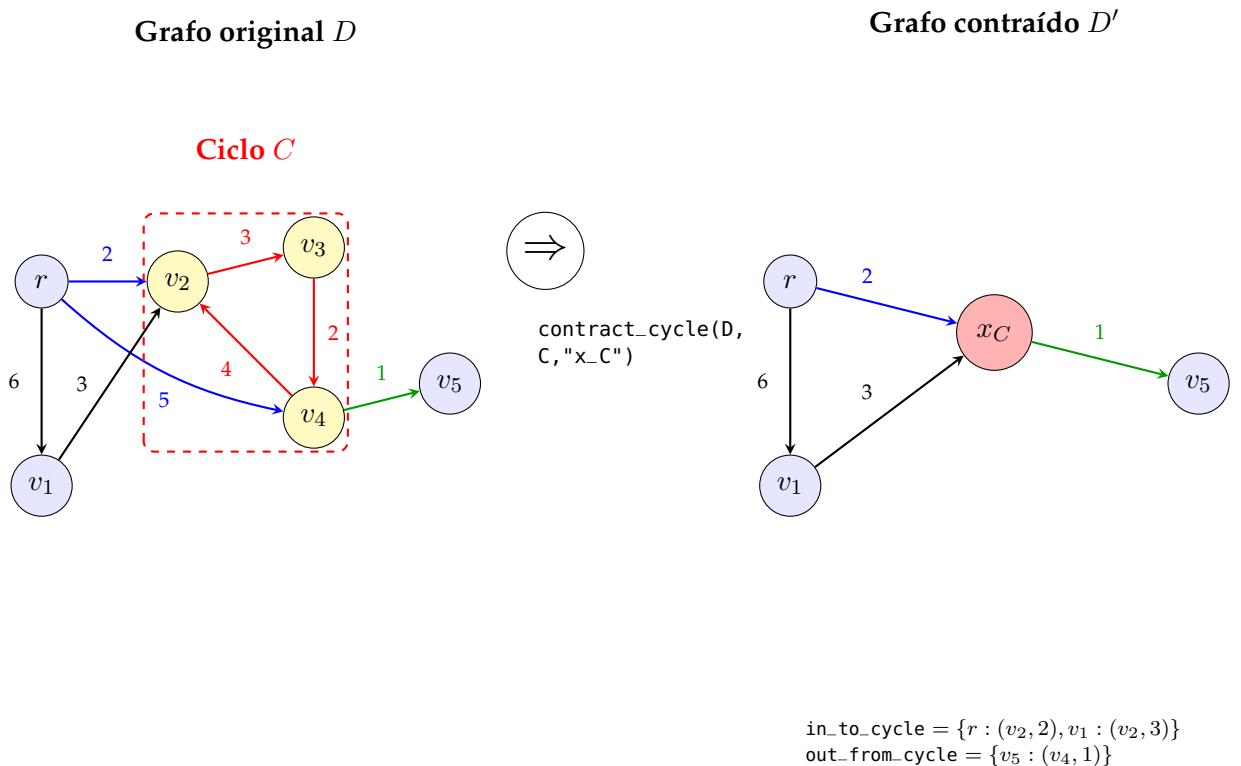


Figura 19 – Exemplo de contração de ciclo. À esquerda, digrafo original  $D$  com ciclo  $C = \{v_2, v_3, v_4\}$  (em amarelo). Vértices externos  $r$ ,  $v_1$  e  $v_5$  têm arcos conectando ao ciclo:  $r$  envia arco para  $v_2$  (peso 2) e  $v_4$  (peso 5);  $v_4$  envia arco para  $v_5$  (peso 1). À direita, após a contração: o ciclo é substituído pelo supervértice  $x_C$  (vermelho). Os arcos de entrada são redirecionados:  $(r, x_C)$  recebe peso 2 (menor entre 2 e 5). O arco de saída  $(x_C, v_5)$  mantém peso 1. Os dicionários  $in\_to\_cycle$  e  $out\_from\_cycle$  armazenam os mapeamentos originais para posterior reexpansão.

A função de deteção de ciclo e a de contração juntas implementam os passos 2 e 3 da descrição do algoritmo de Chu–Liu/Edmonds:

#### Passos 2 e 3 do Algoritmo de Chu–Liu/Edmonds

**Passo 2:** Se  $(V, A_0)$  é acíclico, devolva  $A_0$ . Por (KLEINBERG; TARDOS, 2006, Obs. 4.36), trata-se de uma  $r$ -arborescência de custo mínimo.

**Passo 3:** Caso contrário, seja  $C$  um ciclo dirigido de  $A_0$  (com  $r \notin C$ ). **Contração:** contraia  $C$  em um supervértice  $x_C$  e defina custos  $c'$  por

$$\begin{aligned} c'(u, x_C) &:= c(u, w) - y(w) = c(u, w) - c(a_w) && \text{para } u \notin C, w \in C, \\ c'(x_C, v) &:= c(w, v) && \text{para } w \in C, v \notin C, \end{aligned}$$

descartando laços em  $x_C$  e permitindo paralelos. Denote o digrafo contraído por  $D' = (V', A')$ .

### 1.3.7 Remoção de arco interno:

Esta função é invocada durante a fase de reexpansão do ciclo contraído, após a chamada recursiva devolver com a arborescência ótima  $T'$  do digrafo contraído. Quando o supervértice  $x_C$  é expandido de volta para o ciclo original  $C$ , um arco externo  $(u, v)$  é adicionado conectando um vértice externo  $u$  a um vértice  $v$  dentro do ciclo. Como o ciclo  $C$  originalmente continha exatamente um arco entrando em cada um de seus vértices (formando um ciclo fechado), e agora  $v$  recebe um arco adicional vindo do exterior, esse vértice teria grau de entrada 2, violando a propriedade fundamental de arborescência (cada vértice não-raiz deve ter exatamente uma entrada). Para restaurar essa propriedade, a função remove o arco interno que anteriormente entrava em  $v$ , mantendo apenas o novo arco externo. Essa remoção "quebra" o ciclo no ponto de entrada, transformando-o em um caminho que se integra corretamente à estrutura de árvore.

A função recebe como entrada o ciclo  $C$  e o vértice de entrada  $v$ . A implementação utiliza uma compreensão de gerador combinada com `next` para encontrar o predecessor de  $v$  dentro do ciclo (linha 2): a expressão `(u for u, _ in C.in_edges(v))` itera sobre os arcos de entrada de  $v$ , extraíndo apenas o vértice origem  $u$  (ignorando metadados com `_`), e `next` devolve o primeiro (e teoricamente único) predecessor. Em seguida, remove o arco `(predecessor, v)` do ciclo usando o método `remove_edge` (linha 3).

A função modifica diretamente o subdigrafo  $C$  e não devolve valor. A complexidade é  $O(\deg^-(v))$ , dominada pela operação de busca dos arcos de entrada, embora em ciclos simples isso seja tipicamente  $O(1)$  pois cada vértice tem exatamente um predecessor.

#### Remover arco interno na reexpansão

*Remove o arco interno que entra no vértice de entrada  $v$  do ciclo  $C$  durante a reexpansão, pois  $v$  passa a receber um arco externo, e manter ambos violaria a propriedade de arborescência.*

```

1 def remove_edge_cycle(C: nx.DiGraph, v):
2     predecessor = next((u for u, _ in C.in_edges(v)))
3     C.remove_edge(predecessor, v)

```

A Figura 20 ilustra o objetivo da função:

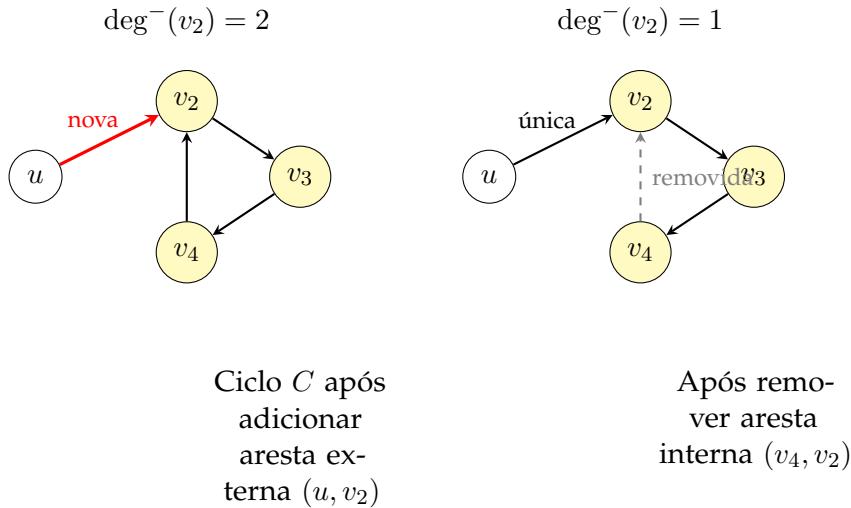


Figura 20 – Remoção de arco interno durante reexpansão. À esquerda, ciclo  $C = \{v_2, v_3, v_4\}$  após adicionar arco externo  $(u, v_2)$  vindouro da arborescência  $T'$ : o vértice  $v_2$  tem grau de entrada 2 (arco externo vermelho de  $u$  e arco interno do ciclo vindo de  $v_4$ ), violando a propriedade de arborescência. À direita, após remover o arco interno  $(v_4, v_2)$ : o vértice  $v_2$  passa a ter grau de entrada 1, o ciclo é "quebrado" no ponto de entrada, transformando-se em um caminho que se integra corretamente à estrutura de árvore. O arco removido é mostrado tracejado em cinza.

### 1.3.8 Procedimento principal (recursivo):

Esta função implementa o algoritmo de Chu–Liu/Edmonds de forma recursiva, orquestrando todas as funções auxiliares descritas anteriormente. Recebe como entrada um digrafo ponderado  $D$  (objeto `nx.DiGraph`), o vértice raiz  $r_0$ , e um parâmetro `level` (padrão 0) usado para rotular supervértices em níveis recursivos distintos.

A implementação segue a estrutura do algoritmo:

Preservação do digrafo original (linha 2):

Cria uma cópia  $D\_copy = D.copy()$  para preservar os pesos originais. Como as operações de normalização e contração modificam os pesos dos arcos diretamente, a cópia é necessária para restaurar os custos corretos na arborescência final. Complexidade:  $O(m + n)$ .

Normalização e construção de  $A_0$  (linhas 3–6):

Itera sobre todos os vértices não-raiz (linhas 3–5), chamando `reduce_weights(D_copy, v)` para cada um. Após normalizar todos os vértices, constrói  $A_0$  (linha 6) chamando `get_Azero(D_copy, r0)`, que seleciona um arco de custo reduzido zero entrando em cada vértice não-raiz.

Verificação de aciclicidade — caso base (linhas 7–10):

Verifica se  $A_0$  é uma arborescência válida usando `nx.is_arborescence(A_zero)` (linha 7). Se sim, restaura os pesos originais de  $D$  para cada arco de  $A_0$  (linhas 8–9) e devolve  $A_{\text{zero}}$  como solução (linha 10). A função `nx.is_arborescence` testa conectividade, aciclicidade e grau de entrada correto simultaneamente.

Contração e resolução recursiva — caso recursivo (linhas 11–16):

Caso  $A_0$  contenha um ciclo, detecta  $C$  chamando `find_cycle(A_zero)` (linha 12). Cria um rótulo único `contracted_label = f"contracted_{level}"` para o supervértice (linha 13). Contraí o ciclo chamando `contract_cycle(D_copy, C, contracted_label)` (linhas 14–15), que modifica  $D_{\text{copy}}$  diretamente criando o digrafo contraído  $D'$  e devolve os dicionários `in_to_cycle` e `out_from_cycle`. Chama-se recursivamente (linha 16) com `chuli(D_copy, r0, level+1)`, obtendo  $F'$ .

Reexpansão do ciclo contraído (linhas 17–30):

Identifica o arco externo que entra no supervértice em  $F'$  (linha 17) e extrai o vértice externo  $u$  (linha 18). Consulta  $v = \text{in\_to\_cycle}[u]$  para determinar o vértice do ciclo que recebe a conexão (linha 19). Remove o arco interno que entrava em  $v$  chamando `remove_internal_edge_to_cycle_entry(C, v)` (linha 20), quebrando o ciclo no ponto de entrada. Adiciona o arco externo  $(u, v)$  a  $F'$  (linha 21) e reintegra os demais arcos do ciclo (linhas 22–23). Processa os arcos de saída (linhas 24–26): para cada  $(\text{contracted\_label}, w)$  em  $F'$ , adiciona  $(v_{\text{out}}, w)$  usando `out_from_cycle[w]`. Remove o supervértice (linha 27), restaura os pesos originais (linhas 28–29) e devolve  $F_{\text{prime}}$  (linha 30).

A função devolve um digrafo contendo exatamente  $|V| - 1$  arcos onde cada vértice  $v \neq r_0$  tem grau de entrada 1, todos os vértices são alcançáveis a partir de  $r_0$ , e o custo total é mínimo. O digrafo original  $D$  não é modificado devido à cópia (linha 2). A complexidade é  $O(mn)$  no pior caso, onde cada nível de recursão (até  $O(n)$  níveis) processa  $O(m)$  arcos durante a normalização, detecção de ciclos e contração/expansão.

O código completo da função principal é apresentado a seguir:

### Procedimento principal (recursivo)

Implementa o algoritmo de Chu–Liu/Edmonds de forma recursiva para encontrar a arborescência de custo mínimo em um digrafo ponderado  $D$  com raiz  $r\theta$ . Normaliza custos, constrói  $A_0$ , detecta ciclos  $e$ , se houver, contrai em supervértice, resolve recursivamente no digrafo reduzido e reexpande, restaurando a arborescência ótima no digrafo original. Devolve um  $nx.DiGraph$  contendo exatamente  $|V| - 1$  arcos com grau de entrada 1 para cada vértice exceto a raiz.

```

1 def chuliu(D: nx.DiGraph, rθ: str, level=0,):
2     D_copy = D.copy()
3     for v in D_copy.nodes:
4         if v != rθ:
5             reduce_weights(D_copy, v)
6     A_zero = get_Azero(D_copy, rθ)
7     if nx.is_arborescence(A_zero):
8         for u, v in A_zero.edges:
9             A_zero[u][v]["w"] = D[u][v]["w"]
10    return A_zero
11 else:
12     C: nx.DiGraph = find_cycle(A_zero)
13     contracted_label = f"\n {level}"
14     in_to_cycle, out_from_cycle = contract_cycle(
15         D_copy, C, contracted_label)
16     F_prime = chuliu(D_copy, rθ, level + 1)
17     in_edge = next(iterator(F_prime.in_edges(contracted_label, data="w")),
18                    None)
19     u, _, _ = in_edge
20     v, _ = in_to_cycle[u]
21     remove_internal_edge_to_cycle_entry(C, v)
22     F_prime.add_edge(u, v)
23     for u_c, v_c in C.edges:
24         F_prime.add_edge(u_c, v_c)
25     for _, z, _ in F_prime.out_edges(contracted_label, data=True):
26         u_cycle, _ = out_from_cycle[z]
27         F_prime.add_edge(u_cycle, z)
28     F_prime.remove_node(contracted_label)
29     for u, v in F_prime.edges:
30         F_prime[u][v]["w"] = D[u][v]["w"]
31     return F_prime

```

Os passos do algoritmo implementados nesta função em conjunto com a função de remover arco interno do ciclo na reexpansão correspondem diretamente à descrição formal do algoritmo de Chu–Liu/Edmonds da seguinte forma:

### Passos 4 e 5 do Algoritmo de Chu–Liu/Edmonds

#### **Passo 4 — Resolução recursiva:**

- Para resolver o digrafo contraído  $D'$  aplica-se uma chamada recursiva: `F_prime = chuli(D_copy, r0, level+1)` (linha 16 na implementação).
- Essa chamada executa novamente a normalização, construção de  $A_0$ , detecção/contração de ciclos e prossegue até que o caso base (arborescência em  $D'$ ) seja atingido.

#### **Passo 5 — Reexpansão:**

- Após obter  $F'$  em  $D'$ , identifica-se o arco  $(u, \text{contracted\_label})$  que entra no supervértice; no digrafo original esse arco corresponde a  $(u, v)$  com  $v = \text{in\_to\_cycle}[u]$ .
- Remove-se o arco interno que entrava em  $v$  (quebrando o ciclo) — função `remove_internal_edge_to_cycle_entry(C, v)` —, adiciona-se o arco externo  $(u, v)$  e reintegram-se os demais arcos de  $C$ ; saídas do ciclo são tratadas via `out_from_cycle` (implementado nas linhas 17–30).
- O resultado é uma arborescência em  $D$  com pesos originais restaurados.

A seguir, ilustramos o funcionamento do algoritmo de Chu–Liu/Edmonds em um digrafo de teste. Mostramos o digrafo original, os principais passos do algoritmo e a arborescência final encontrada.

### Exemplo de execução do algoritmo

A seguir, demonstramos a execução completa do algoritmo de Chu–Liu/Edmonds em um digrafo exemplo, ilustrando cada fase do processo: normalização, construção de  $A_0$ , detecção de ciclos, contração, resolução recursiva e reexpansão.

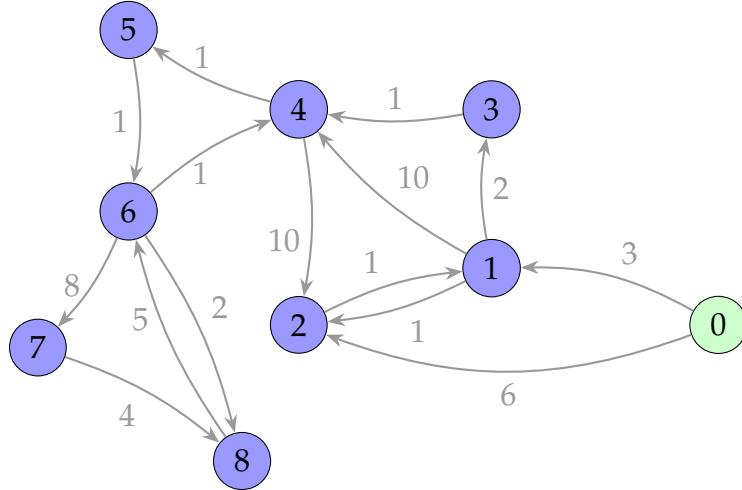


Figura 21 – dígrafo direcionado ponderado inicial com raiz no vértice 0. O dígrafo contém 9 vértices e múltiplos arcos com pesos variados. O primeiro passo do algoritmo seria remover arcos que entram na raiz, porém não há nenhum neste caso, logo não existe necessidade de alterar o dígrafo.

O primeiro passo do nosso algoritmo seria remover os arcos que entram na raiz (vértice 0), porém não há nenhum nesse caso, logo não existe a necessidade de alterar o dígrafo.

O próximo passo é normalizar os pesos dos arcos de entrada para cada vértice. Nessa etapa, para cada vértice  $v$  (exceto a raiz), o algoritmo encontra o arco de menor peso que entra em  $v$  e subtrai esse menor peso de todos os arcos que entram em  $v$  (isso serve para zerar o peso do arco mínimo de entrada em cada vértice).

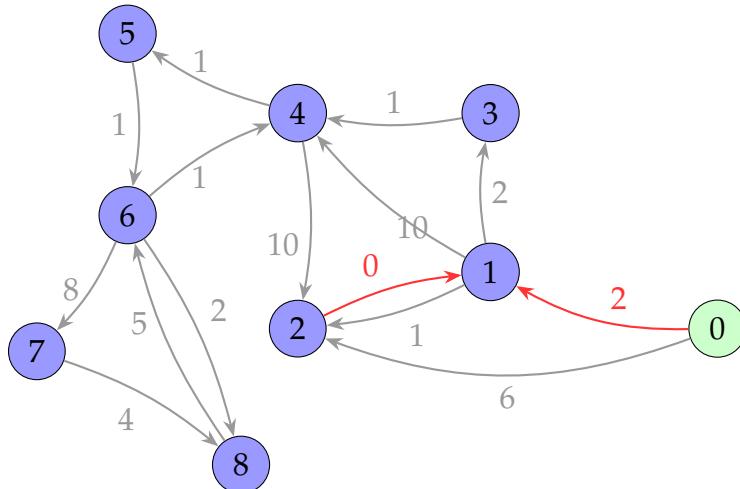


Figura 22 – Normalização parcial dos arcos de entrada para o vértice 1. Os arcos de entrada são  $(0 \rightarrow 1)$  com peso original 3 e  $(2 \rightarrow 1)$  com peso original 1. Elegendo o arco  $(2 \rightarrow 1)$  como o de menor peso (peso mínimo = 1), subtraímos este valor de todos os arcos de entrada:  $(0 \rightarrow 1)$  passa de peso 3 para 2, e  $(2 \rightarrow 1)$  passa de peso 1 para 0 (destacadas em vermelho). Esse processo é repetido para todos os demais vértices.

Com os pesos normalizados, o próximo passo é construir  $A_0$ : para isso, selecionamos para cada vértice o arco de custo reduzido zero de entrada. Detectamos um ciclo em  $A_0$ , formado pelos vértices  $\{1, 2\}$ . Portanto, precisamos contrair esse ciclo em um supervértice  $n * 0$ .

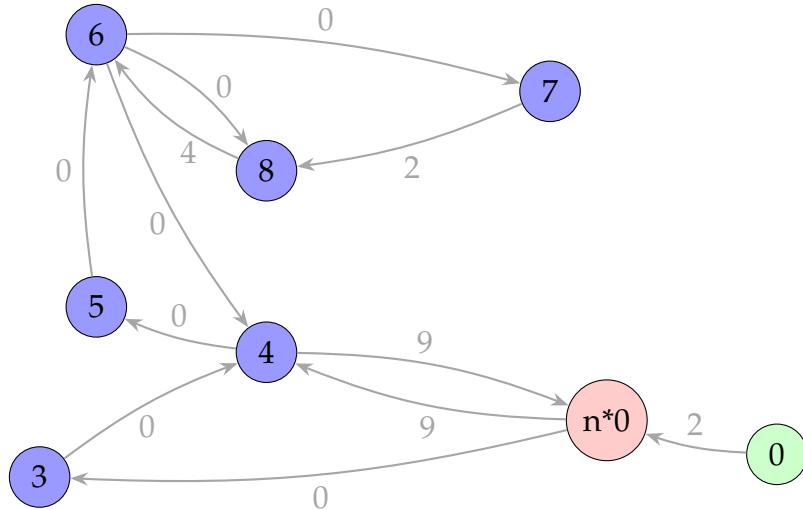


Figura 23 – digrafo contraído apóis detecção do ciclo  $C = \{1, 2\}$  em  $A_0$ . O ciclo foi contraído no supervértice  $n * 0$  (destacado em vermelho). Os arcos que entravam ou saíam do ciclo foram redirecionados para o supervértice, com custos ajustados segundo as fórmulas  $c'(u, x_C) := c(u, w) - y(w)$  para arcos de entrada e  $c'(x_C, v) := c(w, v)$  para arcos de saída.

Agora, repetimos o processo recursivamente no digrafo contraído até obter uma arborescência válida.

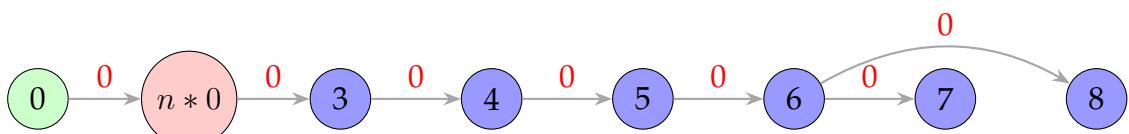


Figura 24 – Arborescência ótima  $F'$  obtida no digrafo contraído. todos os arcos selecionados têm custo reduzido 0 (destacados em vermelho), e o digrafo forma uma arborescência válida enraizada em 0: cada vértice (exceto a raiz) tem exatamente um arco de entrada, não há ciclos, e todos os vértices são alcançáveis a partir da raiz. Como  $F'$  é acíclico, alcançamos o caso base da recursão.

Após validarmos que  $A_0$  não possui mais ciclos e forma uma arborescência, iniciamos o processo de reexpansão do ciclo contraído para obter a arborescência final no digrafo original. Adicionamos o arco de entrada ao ciclo  $(0, 1)$ , os arcos internos do ciclo modificado  $(1, 2)$ , e os arcos de saída  $(1, 3)$ , chegando a uma arborescência válida.

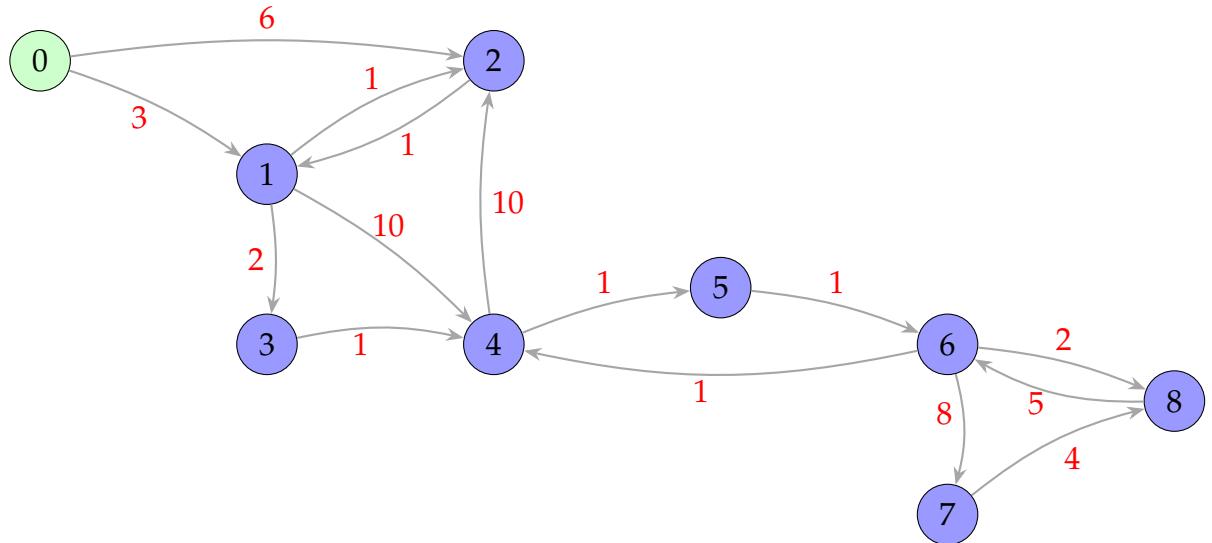


Figura 25 – Arborescência ótima final no digrafo original com pesos restaurados. O supervértice  $n * 0$  foi expandido de volta para os vértices 1 e 2, com o arco externo  $(0, 1)$  escolhido pela solução recursiva conectando ao ciclo. O arco interno  $(2, 1)$  do ciclo original foi removido para manter a propriedade de arborescência ( $\deg^-(v) = 1$ ). O resultado é uma 0-arborescência de custo mínimo com exatamente 8 arcos, onde cada vértice não-raiz tem grau de entrada 1 e todos são alcançáveis a partir da raiz 0.

### 1.3.9 Correspondência entre teoria e implementação

A implementação em Python segue fielmente os cinco passos da descrição teórica do algoritmo de Chu–Liu/Edmonds apresentada na Seção anterior. A tabela abaixo estabelece o paralelo direto entre cada passo teórico e sua realização no código:

Descrição Teórica	Implementação Python
<p><b>Passo 1:</b> Normalização e construção de <math>A_0</math>. Para cada <math>v \neq r</math>, escolha <math>a_v \in \arg \min_{(u,v) \in A} c(u, v)</math>. Defina <math>y(v) := c(a_v)</math> e <math>A_0 := \{a_v : v \neq r\}</math>.</p>	<p><b>Linhas 3–6:</b>  <code>for v in D_copy.nodes:     reduce_weights(D_copy, v) A_zero = get_Azero(D_copy, r0)</code>          Calcula <math>y(v)</math> e cria custos reduzidos, depois constrói <math>A_0</math> selecionando arcos de custo zero.</p>
<p><b>Passo 2:</b> Verificação de aciclicidade (caso base). Se <math>(V, A_0)</math> é acíclico, devolva <math>A_0</math>. Por Obs. 4.36 de (KLEINBERG; TARDOS, 2006), trata-se de uma r-arborescência de custo mínimo.</p>	<p><b>Linhas 7–10:</b>  <code>if nx.is_arborescence(A_zero):     [restaura pesos originais]     return A_zero</code>          Testa conectividade, aciclicidade e grau de entrada correto simultaneamente.</p>
<p><b>Passo 3:</b> Contração de ciclo. Caso contrário, seja <math>C</math> um ciclo dirigido de <math>A_0</math> (com <math>r \notin C</math>). Contraia <math>C</math> em supervértice <math>x_C</math> e defina custos <math>c'</math> por:  <math>c'(u, x_C) := c(u, w) - y(w)</math>  <math>c'(x_C, v) := c(w, v)</math>          Denote o digrafo contraído por <math>D' = (V', A')</math>.</p>	<p><b>Linhas 11–15:</b>  <code>C = find_cycle(A_zero) label = f"contracted_{level}" in_to_cycle, out_from_cycle =     contract_cycle(D_copy, C, label)</code>          Implementa as fórmulas de ajuste de custos e modifica <math>D_{copy}</math> para criar <math>D'</math>.</p>
<p><b>Passo 4:</b> Resolução recursiva. Resolva recursivamente em <math>D'</math>, obtendo arborescência ótima <math>F'</math>.</p>	<p><b>Linha 16:</b>  <code>F_prime = chuliu(     D_copy, r0, level+1)</code>          Chamada recursiva resolve o problema no digrafo contraído.</p>
<p><b>Passo 5:</b> Reexpansão. Expanda <math>x_C</math> para o ciclo original <math>C</math>. Se <math>(u, x_C) \in F'</math>, adicione <math>(u, v)</math> onde <math>v</math> é o vértice do ciclo mapeado por <math>u</math>, remova o arco interno entrando em <math>v</math>, e reintegre demais arcos de <math>C</math>. Restaure custos originais.</p>	<p><b>Linhas 17–30:</b>  <code>v = in_to_cycle[u] remove_internal_edge_to_cycle_entry(C, v) F_prime.add_edge(u, v) F_prime.add_edges_from(C.edges) [processa saídas, remove supervértice] [restaura pesos originais]</code></p>

Tabela 1 – Correspondência entre os cinco passos teóricos do algoritmo de Chu–Liu/Edmonds e sua implementação em Python. Cada linha da coluna direita mostra a tradução direta dos conceitos matemáticos da coluna esquerda em operações concretas sobre grafos.

Esta correspondência demonstra que a implementação não é uma aproximação ou interpretação livre da teoria, mas uma tentativa de traduzir fielmente a descrição teórica. As funções auxiliares (`reduce_weights`, `get_Azero`, `find_cycle`, `contract_cycle`, `remove_internal_edge_to_cycle_entry`) encapsulam exatamente as operações descritas na formulação teórica, preservando as propriedades de correção e complexidade do algoritmo original.

### 1.3.10 Transição para a abordagem primal-dual

Embora o algoritmo de Chu–Liu/Edmonds seja elegante e eficiente, sua mecânica operacional — normalizar custos, selecionar mínimos, contrair ciclos — pode parecer um conjunto de heurísticas bem-sucedidas sem uma justificativa teórica unificadora aparente. Por que escolher a melhor entrada para cada vértice garante otimalidade global após o tratamento de ciclos? A resposta reside na *dualidade em programação linear*.

No capítulo seguinte, revisitaremos o mesmo problema sob uma ótica primal–dual em duas fases, proposta por András Frank. Essa perspectiva organiza a normalização via potenciais<sup>3</sup>  $y(\cdot)$ , explica os custos reduzidos e introduz a noção de cortes apertados (família laminar) como guias das contrações. Veremos como a mesma mecânica operacional (normalizar → contrair → expandir) emerge de condições duais que também sugerem otimizações e generalizações.

---

<sup>3</sup> No contexto primal–dual, “potenciais” são valores escalares  $y(v)$  atribuídos aos vértices para definir custos reduzidos  $c'(u, v) = c(u, v) - y(v)$ . Ajustar  $y$  desloca uniformemente os custos dos arcos que entram em  $v$ , sem mudar a otimalidade global: preserva a ordem relativa entre entradas e torna “apertadas” (custo reduzido zero) as candidatas corretas, habilitando contrações e uma prova de corretude via cortes apertados.

## Referências

KLEINBERG, J.; TARDOS, É. *Algorithm Design*. [S.l.]: Addison-Wesley, 2006. Citado 4 vezes nas páginas [21](#), [22](#), [33](#) e [42](#).

SCHRIJVER, A. *Combinatorial Optimization: Polyhedra and Efficiency*. [S.l.]: Springer, 2003. Citado 2 vezes nas páginas [21](#) e [22](#).

## Anexos

## ANEXO A – Anexo A

Conteúdo do anexo A.